SAPIENZA
UNIVERSITÀ DI ROMA

MASTER'S DEGREE COURSE IN MANAGEMENT ENGINEERING

LABORATORY PROJECT 1 – SERVICE SYSTEMS

NETWORK RESOURCE MANAGEMENT

Work Team: M.A.T.

Members:

Alberto Magro 1935939

Alessandro Rossi 1941142

Andrea Di Biase 1939038

Giuseppe Di Tommaso 1936118

Francesco Sanetti 1933718

Francesco Ventura 1841504

Maria Carla Fiorella 1837520

Professor: Andrea Baiocchi

Academic Year 2023/24 - II semester

Summary

1. Introduction

2. Data Metric Evaluation

3. JIQ – FCFS Performance Analysis

3.1 - JIQ Dispatcher Analysis

3.2 - FCFS Scheduling Analysis

3.3 Measuring the Variability of a Process

3.4 - JIQ-FCFS Perfomance Evaluation

4. Design, Implementation and Analysis of System Management Models

4.1 – LAS Implementations

4.1.1 – Preliminary Considerations for the Design Phase

4.1.2 – Comparison with Other Policies

4.1.3 – First Implementation LAS Policy with Tables

4.1.4 - B-LAS and SSRE-LAS Implementations

4.1.5 - DC-LAS Implementation

4.1.6 - Performance Analysis

4.2 – Hybrid JSQ with dynamic server allocation

4.2.1 - Algorithm description

4.2.2 - Servers' dynamic allocation

4.2.3 - Benefits and drawbacks

4.2.4 - Future Improvements

4.2.5 - Implementation

4.2.6 - Result analysis

5. Conclusions

1 - Introduction

Consider a computational cluster composed of a dispatcher and NNN servers, each with its own memory for storing processes and its own CPU for executing them, subjected to a workload described by a dataset obtained from measurements taken at a Google data center (https://github.com/MertYILDIZ19/Google_cluster_usage_traces_v3_single_cell). The objective of the project lab is to evaluate the balances achievable between the parallelism of the servers and the computational power of each, defining scheduling and dispatching algorithms for this purpose and assessing the quality of possible solutions in terms of the average job response time.

The workload, that is, the dataset extracted from Google's public data, consists of a CSV file with the following columns:

- **JID** : job identifier;
- **TID** : task identifier within the job;
- **A** : task arrival time in the system, expressed in milliseconds;
- **C** : running time required to execute the task with a Google Normalized Computing Unit (GNCU), expressed in seconds;
- **M** : memory space required to execute the task, expressed in Google Normalized Memory Unit (GNMU).

We will always work under the following assumptions:

- Maximum overall capacity of the computational cluster: μ,tot = 10 GNCU;
- Service time of a task: X = CPU/ μ;
- The total available computational budget, that is Nμ, must be such that the average utilization coefficient is: $\varrho = 0.6$;
- All tasks must be processed, but they can be processed and managed independently of each other;

- Time horizon for evaluating metrics : time elapsed between the arrival of the first task and the arrival of the last task.

The following metrics will be evaluated as performance indicators of the system:

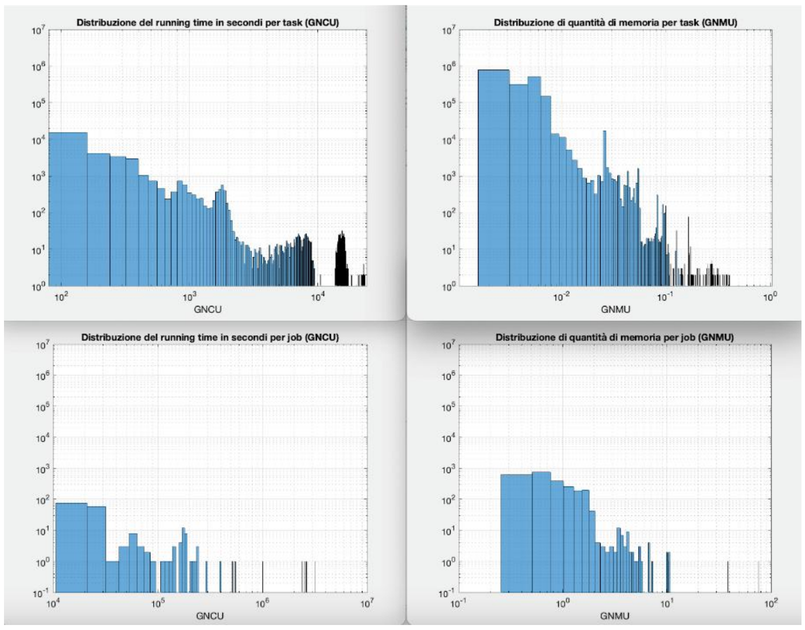
- Job response time (R) : the time interval between the arrival of the first task of the job and the exit of the last task of the job;
- Job slowdown (S) : the ratio between the response time RRR of the job and the sum of the service times of the tasks belonging to the job;
- Utilization coefficient (ϱ) : the fraction of time during which tasks are in execution.

2 - Data Metric Evaluation

The workload extracted from Google's public data starts on 1st May 2019 at 00:00:00 PDT and ends on 31st May 2019.

First of all, with function *GWLM_data_cleaner* all the unit of measurement are converted in seconds, all the tasks having null CPU are removed and jobs and tasks are numbered with integer values starting from 1.

In order to get a better sense of the distribution of the main characteristics of the jobs and the tasks, a simple statistical analysis was performed. By analyzing the distribution of the running time and of the used memory for jobs and tasks, the following histogram is made.



Due to the large diversity in running time and used memory, the x-axis is reported in log scale. The same was done for the y-axis to make the few demanding jobs more visible.

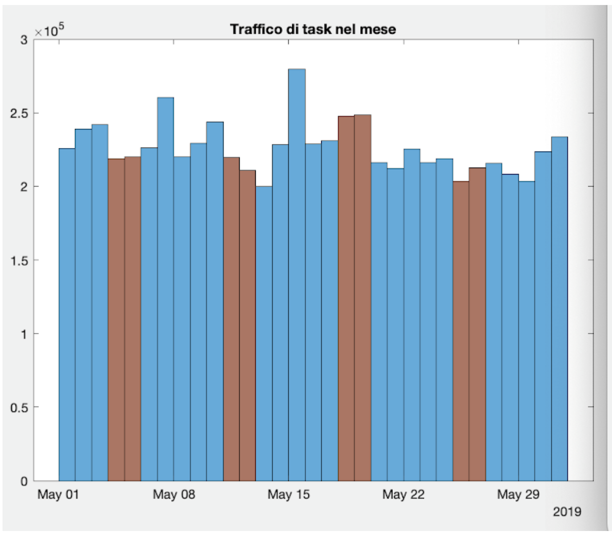
These histograms show that there is more presence of tasks and jobs having a low value of running time and used memory than tasks and jobs having a high value of running time and used memory.

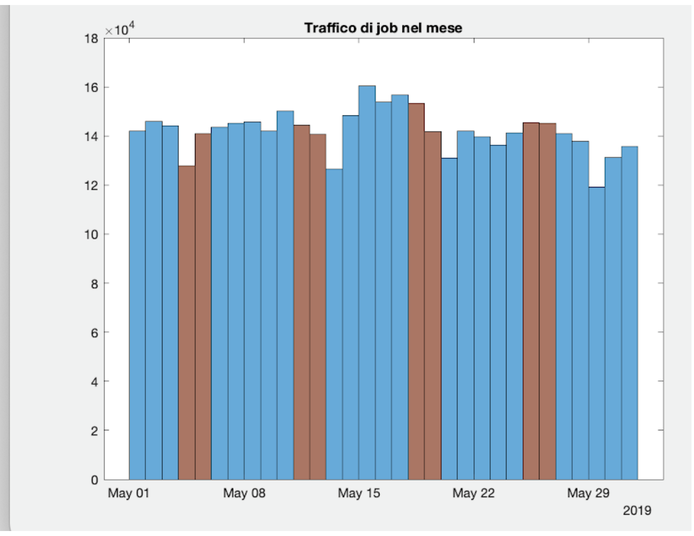
In particular, an additional analysis was performed to assess the distribution of the requested running time and memory, by grouping the resource with respect to a given percentage:

- considering tasks running time:

    - 99% are lower than 39.22 s
    - 75% are lower than 0.71 s
    - 50% are lower than 0.11 s
    - 25% are lower than 0.02 s
- considering tasks used memory:
    - 99% are lower than 0.00789 GNMU
    - 75% are lower than 0.00164 GNMU
    - 50% are lower than 0.00100 GNMU
    - 25% are lower than 0.00038 GNMU
- considering jobs running time:
    - 99% are lower than 32.26 s
    - 75% are lower than 0.45 s
    - 50% are lower than 0.09 s
    - 25% are lower than 0.02 s
- considering jobs used memory
    - 99% are lower than 0.01010 GNMU
    - 75% are lower than 0.00249 GNMU
    - 50% are lower than 0.00094 GNMU
    - 25% are lower than 0.00051 GNMU

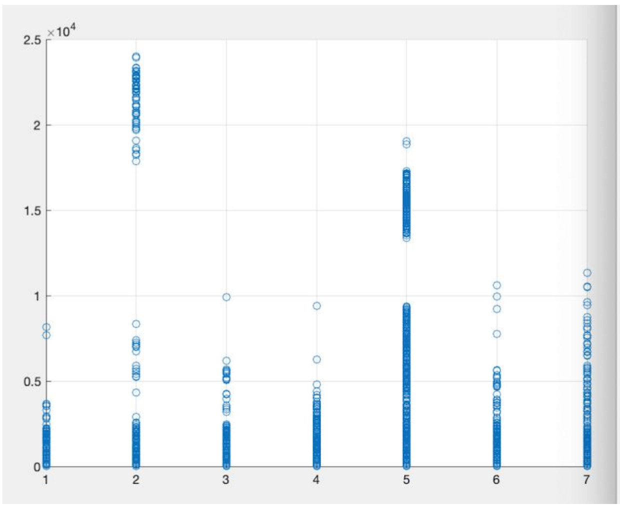
Analyzing the tasks and jobs traffic during the month, the major peak for both is on 15th May 2019 while the minor tasks traffic peak is on 13th May of 2019 and the minor jobs traffic peak is on 29th May 2019 as it is shown in the following graph.

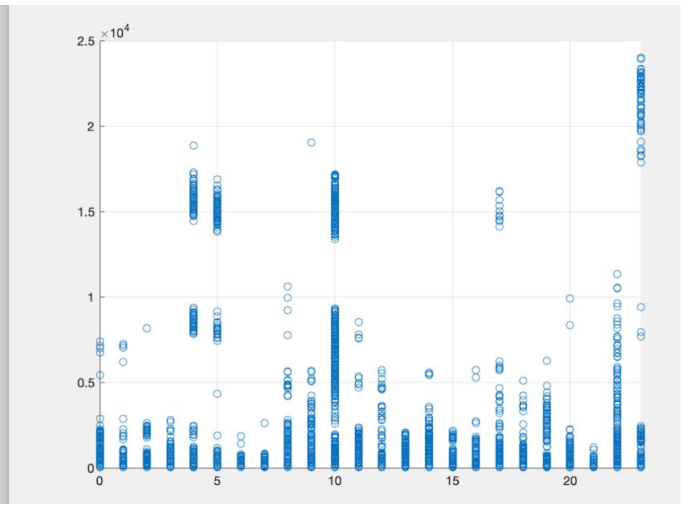




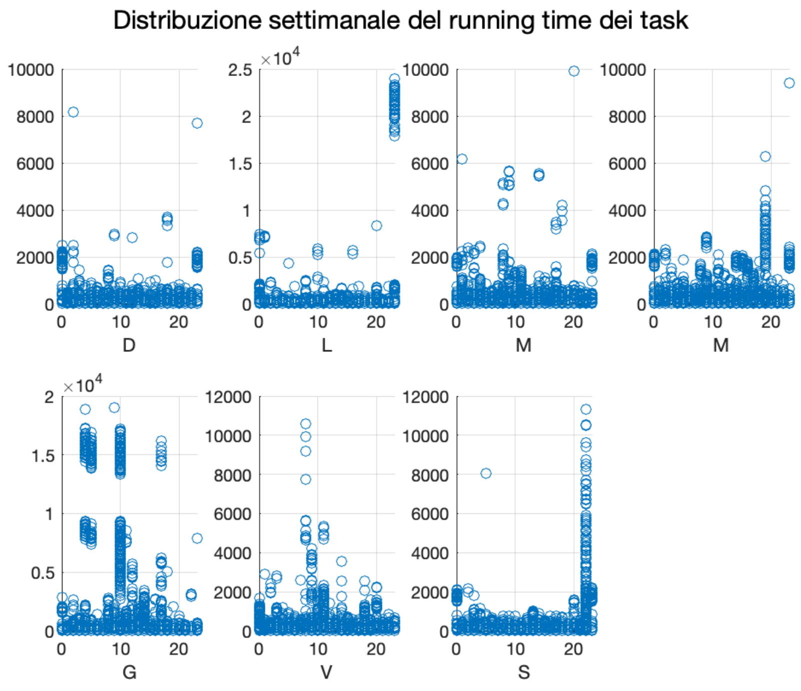
This shows that there is little to no correlation between a particular day of the month and the traffic.

From a weekdays point of view the peaks are on Monday (x = 2 in the left graph) and on Thursday (x = 5 in the left graph), but again there seems to be little correlation between the weekday and the particular traffic. Same goes for the hour of the day (right graph), where it is shown that the running time is marginally higher during late morning at 9 a.m. and at the end of the day at 11 p.m.

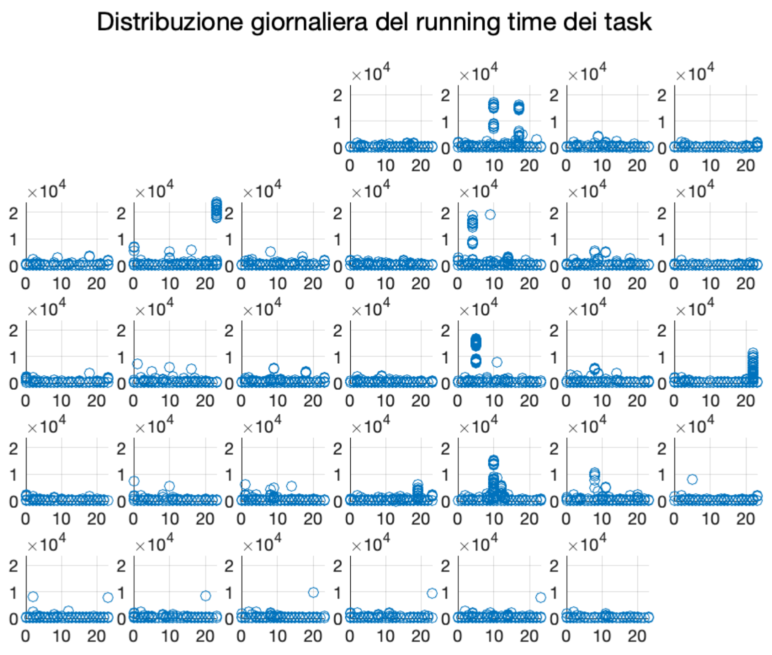




To have more accurate data analysis and to understand how the running time varies during the day, a day by day running time analysis is made. As it is shown in the following graph, the peaks are at the end of Monday, on Thursday and on Saturday night.



The day by day running time task distribution is reported starting from 1st May 2019. The graph shows that on Thursday there are weekly peaks starting from 2nd May 2019. Other values above the average occur on 6th May 2019 and on 18th May 2019.



In conclusion, jobs and tasks have a mainly homogeneous distribution throughout the month. There is no significant difference between weekdays, apart from Thursday which shows some edge cases that can be useful for testing. There is little difference in the workloads between different hours of the day. The majority of the jobs and tasks have low time and memory usage.

3 - JIQ – FCFS Performance Analysis

3.1 - JIQ Dispatcher Analysis

The functioning of a system that uses JIQ as a dispatching policy is as follows:

- A dispatcher with the Join Idle Queue policy sends job requests to the queue of the server that is found to be eligible. This strategy indeed uses "Idle tokens" indicators contained within an N-bit vector where N is the number of servers. When a job request arrives, the dispatcher randomly selects one of the servers whose corresponding bit is equal to 1, which means that the server will be IDLE, and the dispatcher can direct the task. If there are no idle servers, it will randomly select among the servers present in the system. Once the dispatcher sends the job request and the server is no longer IDLE, its token is set to 0 so that when the server completes the task, it can send a new Idle token this time with the bit equal to 1 to accept new job requests.

Advantages of Join Idle Queue:

- Being a pull-type policy, dispatching is instantaneous, and there will be no delays in routing tasks.
- Minimization of waiting time as job requests are preferably directed to inactive servers.
- Reduction of balancing load since the dispatcher does not need to maintain detailed information on each server (1 bit per server).

Disadvantages of Join Idle Queue:

- Performance substantially degrades in the case of heavy traffic.

- The accuracy of the system strongly depends on the accuracy of the indicators (idle tokens).

3.2 - FCFS Scheduling Analysis

The fundamental principle of the FCFS (First-Come, First-Served) policy is that tasks in server queues are handled in the order they arrive without prioritizing larger tasks or those that require less time.

Advantages of FCFS:

- Extremely easy to implement.
- Fairness in resource distribution, every task is treated the same way without following other metrics than "first-come, first-served" to prioritize processes.

Disadvantages of FCFS:

- Does not dynamically adapt the order of processes based on the current conditions of the system.
- The average waiting time can be very high, especially for processes with high variability. In fact, the average waiting time for FCFS is directly proportional to the variability of service times.

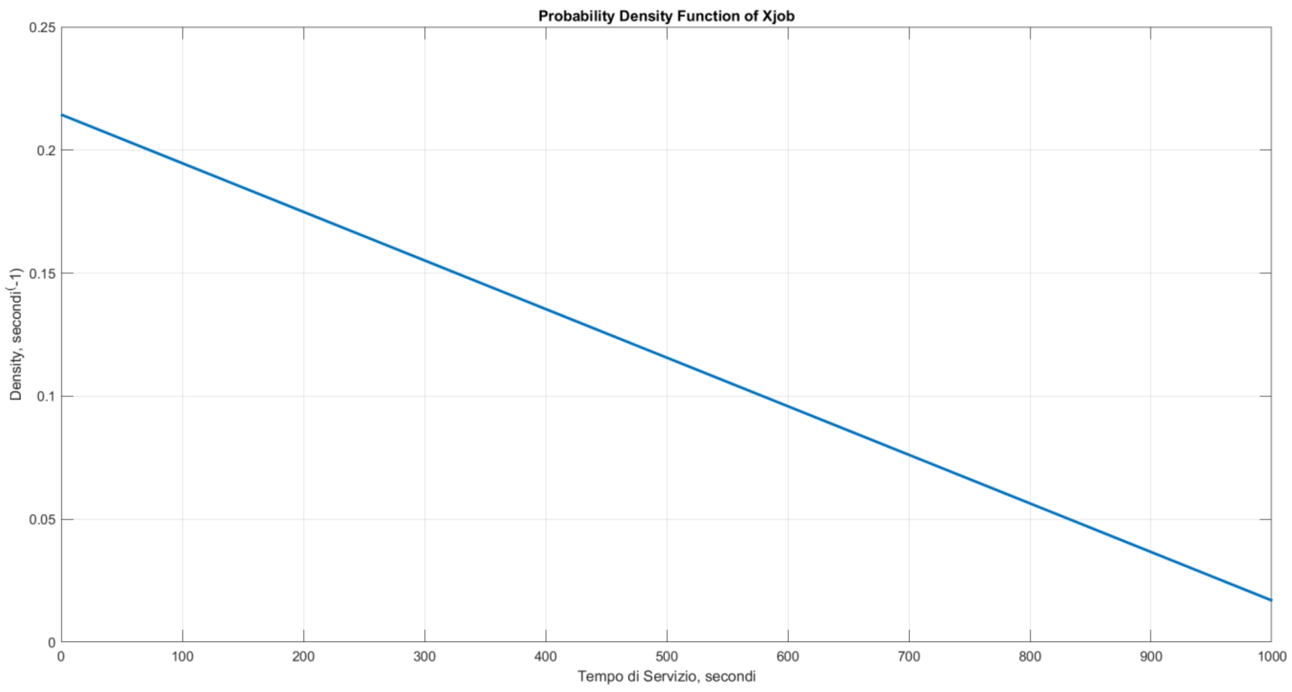
What happens if we have a high variability process and an FCFS policy?

3.3 Measuring the Variability of a Process

We define the "coefficient of variation" and "square coefficient of variation" as measures of the variability of a process.

The COV is defined as the ratio between the standard deviation of the service times "σ" and the mean of the service times "μ" $COV= \sigma/\mu$ while the SCOV is the square of the former. By calculating the coefficient of variation (COV), we found values equal to 276.3460 and consequently the squared coefficient of variation (SCOV) of 76,367.111716.

These very high values indicate a high variability in service times, comparable to a heavy-tailed Pareto distribution.



Indeed, looking at the probability density function (PDF) of the xjob, we notice that our expectations are met. Under these assumptions, we expect our system to perform poorly, suffering from all this variability without doing anything to mitigate it.

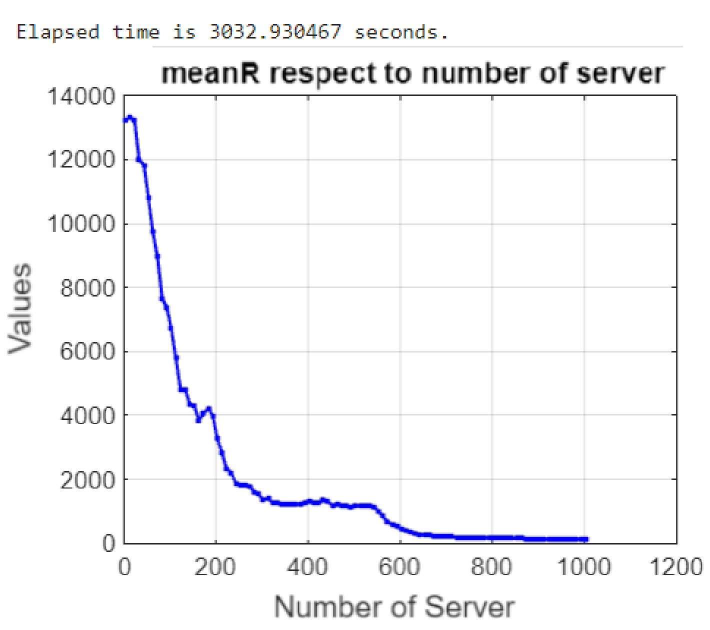
3.4 - JIQ-FCFS Perfomance Evaluation

To evaluate the metrics of our interest, the first step was to modify the dataset "cell_c.csv" by converting the arrival times to the unit of measure that was most convenient for us (seconds) and filtering the rows where CPU values are non-zero. Subsequently, after extracting the columns Job_ID, Arrival_Time, and CPU, we initialized the parameters (Nmin, T, Nmax) where T is the number of jumps between one cycle and another thus moving to the next step of simulation for the calculation of parameters, namely:

Response Time

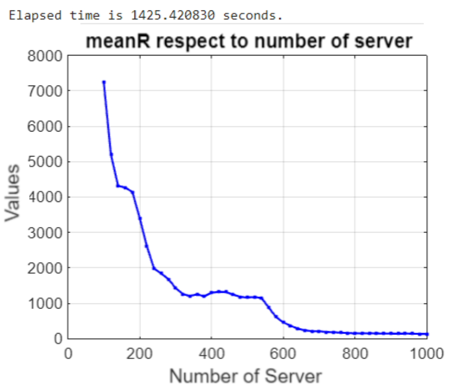
To evaluate the response time, we used a "splitapply" function to determine the first arrival time for each Job_ID and the last completion time for each Job_ID. We then defined the response time "R" as the difference between the last completion time and the first arrival time.

Analyzing the graphs, a general trend can be observed: as the number of servers increases, the average response time (mean R) tends to decrease. This is an expected behavior, as more servers allow for handling more requests simultaneously, thereby reducing the average response time.

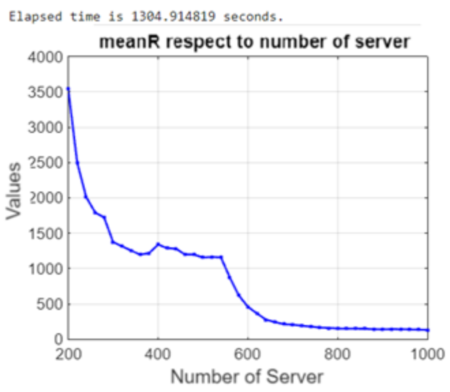


```
Elapsed time is 3032.930467 seconds.
```

In detail, the graph plotted with the following parameters (Nmin=3, T=10, Nmax=1003) shows a significant decrease in the average response time up to around 200 servers. However, let's analyze in more detail what happens as the number of servers increases.



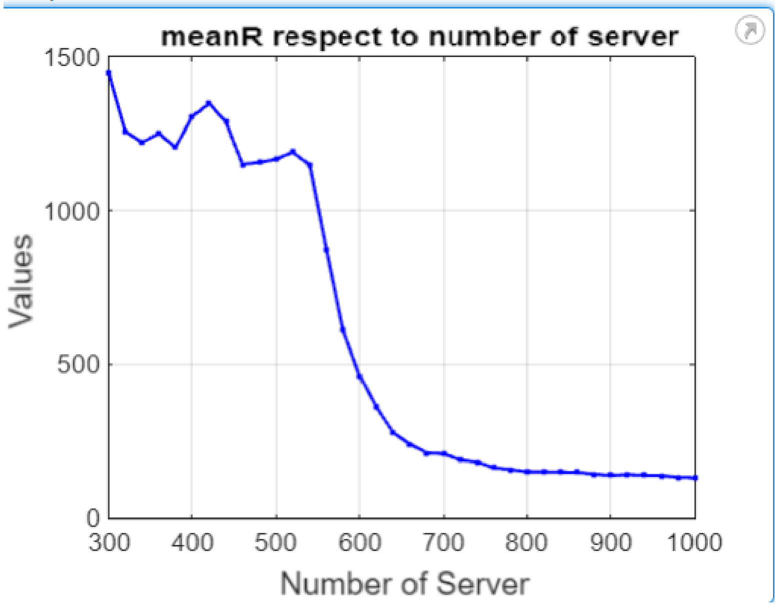
```
Elapsed time is 1425.420830 seconds.
```



```
Elapsed time is 1304.914819 seconds.
```

As we can see from the two graphs, as the number of servers increases, the response time decreases. This is because the dispatcher will find more and more free servers to assign new tasks to, thus reducing the queues for each server. However, this does not eliminate the problem of high data variability, as a large job can still cause problems for smaller jobs that arrive after it and have to wait for its entire service time plus own.

```
Elapsed time is 1175.652890 seconds.
```

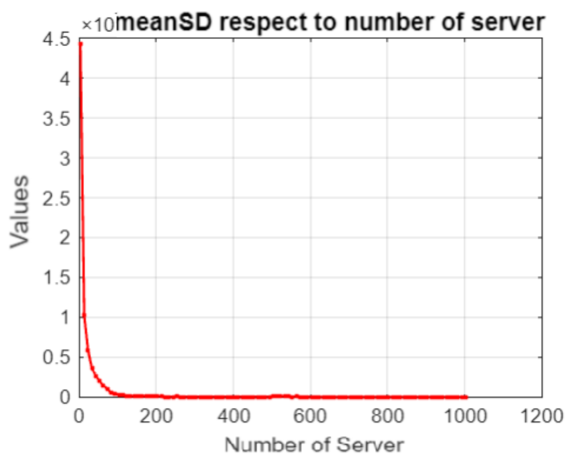


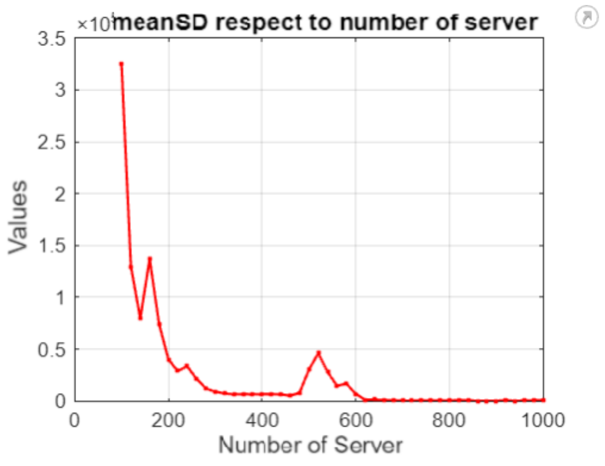
.

At the end of this inspection of the results regarding the response time metric, we can say that the response time decreases significantly for the first 200 servers, then continues to decrease more marginally beyond the threshold of 600 servers, reaching an optimum at 750 servers. Ultimately, this graph helps us understand how a simple implementation system like JIQ-FCFS can have negative implications. Although we managed to achieve "acceptable" response time values, the implementation cost would still be too high given the excessive number of servers required. Moreover, to eliminate the issue of high arrival variability, we would need an offered traffic with an infinite number of servers, which would not be realistic.

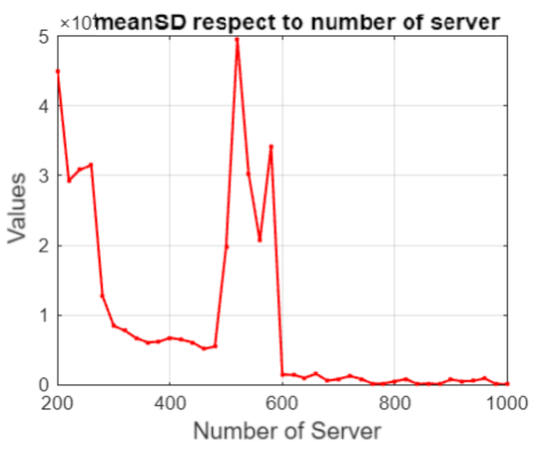
SlowDown

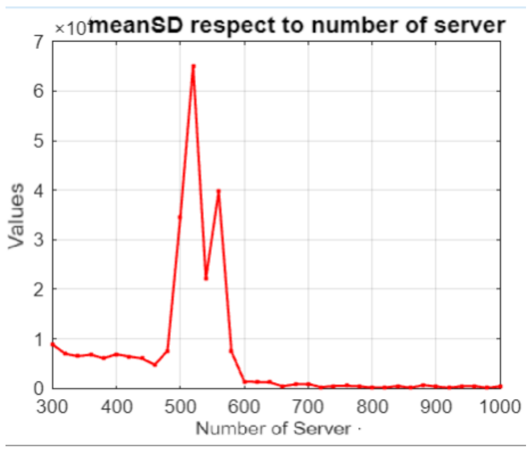
The slowdown is the measure of how much extra time a job takes to complete compared to the ideal execution time without interference. In our case, we have defined this metric as the ratio between the response time (R) and the total execution time of the job, calculated as the sum of the normalized execution times of the tasks for each Job_ID.





By analyzing the slowdown through the obtained graphs, we observe that starting from a macroscopic focus and gradually highlighting the significant aspects of the metric of interest, there is a notable peak in the values when the number of servers reaches around 500.





After this peak, there is a rapid decrease in values as the number of servers increases, which could suggest that there is an optimal point near 500, beyond which, for a short period, performance decreases dramatically. (Optimal slowdown values are in a range of 1). Ideally, I provide all users with the same slowdown, ensuring that those who "have more tasks to do" will wait longer.

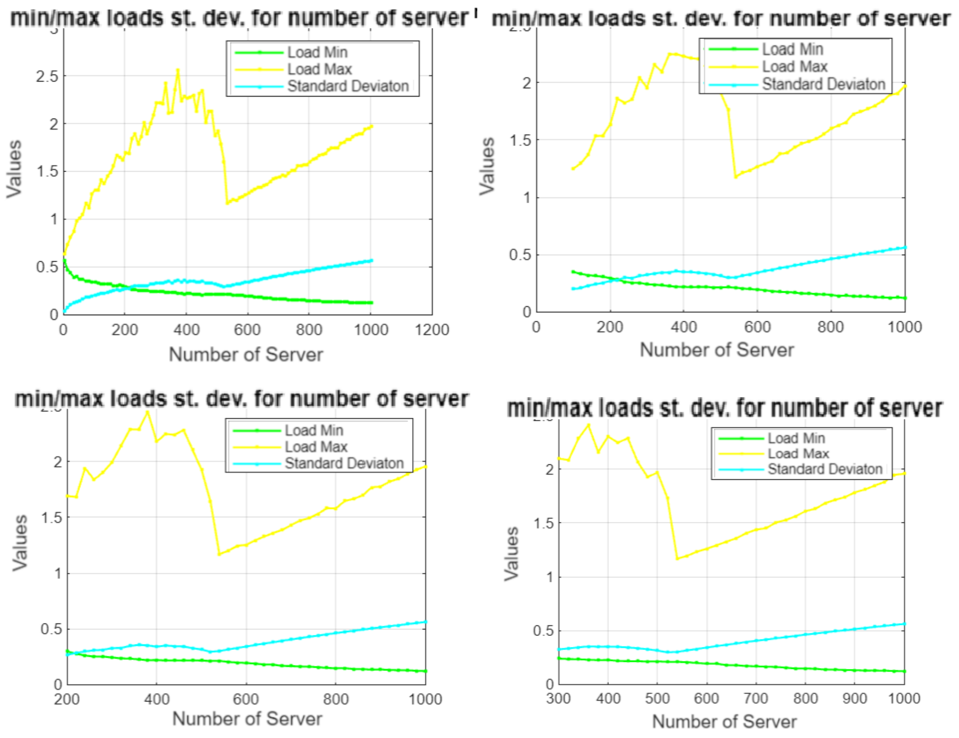
But what’s the meaning of different values of Slowdown?

Slowdown = 1: The job is executed without delays, which means that the system has enough resources to run the job immediately upon its arrival.

Slowdown > 1: The job has experienced delays due to competition for resources or other causes, which means that the system is overloaded and jobs must wait to acquire resources.

In our case, with a magnitude of around 10^4, the slowdown is much greater than 1. This does not surprise us, as we expected that the various tasks for a realistic number of servers would compete for resources, and it is likely that some smaller tasks had to wait longer than necessary.

Min/Max loads and Standard deviation



Looking at the graph we have for:

Maximum Load (Yellow Line):

- **Initial Increase:** The maximum load starts low but increases sharply as the number of servers approaches approximately 500. This indicates that initially, as more servers are added, some servers experience significantly higher loads due to uneven distribution or contention for resources.

- **Peak Around 500 Servers:** The maximum load reaches a peak around 500 servers, suggesting a critical point where the system's performance is most imbalanced. This could be due to the limitations in the load distribution mechanism of the JIQ dispatcher, where some servers may become overloaded while others are underutilized.
- **Subsequent Decrease and Stabilization:** Beyond 500 servers, the maximum load decreases and then stabilizes. This trend suggests that after a certain threshold, adding more servers helps balance the load across the system, reducing the maximum load experienced by any single server.

Minimum Load (Green Line):

- **Initial Increase:** The minimum load also increases as the number of servers increases, but at a much slower rate compared to the maximum load. This indicates that initially, even the least loaded servers are seeing a gradual increase in their load.
- **Flattening Beyond 500 Servers:** Beyond the 500-server mark, the minimum load plateaus and remains relatively stable. This suggests that additional servers do not significantly reduce the load on the least loaded servers, implying a more balanced load distribution across the system.

Standard Deviation (Cyan Line):

- **Initial Increase and Peak:** The standard deviation of the loads increases with the number of servers, peaking around 500 servers. This indicates a growing disparity in load distribution, where some servers are much more heavily loaded than others.
- **Decrease and Stabilization:** After the peak, the standard deviation decreases and stabilizes, suggesting that beyond a certain number of servers, the load distribution becomes more even, and the system reaches a more stable state.

Important Points

- **Critical Point at 500 Servers:** The graph highlights a critical point around 500 servers where the system experiences the highest load imbalance. This is evident from the peak in both the maximum load and standard deviation. At this point, the JIQ dispatcher and FCFS scheduler combination appears to be most challenged in balancing the load effectively.

- **Balancing Effect of Additional Servers:** Beyond 500 servers, the addition of more servers appears to help in balancing the load, as seen by the decreasing maximum load and standard deviation. This suggests that there is a threshold beyond which the system can better distribute tasks and manage loads more evenly.

Conclusion

As we have seen in the previous metrics, in this case as well, there is a threshold beyond which the system tends to stabilize. However, this is difficult to apply in reality given the high number of servers required.

4 - Design, Implementation, and Analysis of System Management Models

4.1 – LAS Implentations

4.1.1 – Preliminary Considerations for the Design Phase

Our goal is to understand and size a system that provides a service: we know that the system to provide the service needs resources (which can be in terms of channel capacity, energy, space, etc.) that in our case are GNCU (Google Normalized Computing Unit). Our aim is to understand how to optimally manage the resources that provide the service.

Remember that we refer to the concept of a service system as an abstraction that represents different types of systems using resources to provide a service to a population of users/customers. In our project, we will simultaneously perform:

- Analysis, based on the input system (demand) and the system's operating mechanisms (number of servers, amount of resources available to each server, policy with which the service is provided to users). We need to provide system performance metrics (average job response time, slowdown, etc.)

- Sizing, starting from the input system (demand) and performance constraints (the average utilization coefficient cannot exceed 60%), we need to size the amount of resource required to operate the system within the required performance limits.

We want to put together these three system dimensions:

- Demand (comes from the users)

- System capacity (what the system can do, which depends on the resources it uses, the way it uses them, and the amount it has)

- Performance metrics (directly related to the quality of the system)

Typically, these three dimensions make the problem non-trivial. To put together not only these three but also arrivals and quantitative assessments of the system (understanding how to build it to meet quality requirements), we use models.

The concept of a model is fundamental for an engineering approach for two basic purposes:

- Gaining insight, deep understanding of the system. If I define a model that matches experimental data or that fits what I get from field trials, it means that I deeply understood how the system is working: I gained a deep understanding of how the system runs.

- Striking the best possible trade-off between competing quantities. The typical task of an engineer is to quantitatively solve problems, which means finding the best trade-off.

We will therefore build a mathematical model that includes:

- A structure of the service system, the resources used, and the service policies

- A (statistical) description of the demand

- A (statistical) description of the service capacity

The type of model we use to study this system is an analytical model because it allows us to understand the phenomena driving the system while requiring little time, but at the same time, it is very simplified. Thanks to this choice of model type, we can ask "WHAT IF?" questions, such as "What if I change the capacity/number of servers/service policy/etc..." without having to build the real system and make the appropriate changes. We just need to build this simple theoretical model and try to understand which direction the system moves once the chosen changes are made.

The mathematical model that we refer to has a name, “the queuing system.” This mathematical model has a center (anything that provides the service) that usually, in software, is translated into hardware facilities set up to provide the kind of service we need (at least in the ICT realm). There are customers who represent service requests that come (they might be packets, requests to transfer files, requests for processing), and customers can be admitted or rejected into the system. Once they are admitted, they can possibly wait before getting into the server. Once the customer finally gets into the server, they receive the service and then leave the system.

A service system has essentially three components:

- Arrivals, which usually take place according to what we call a “stochastic process,” so arrivals take place randomly.

- Service time, which is the time it takes to complete the customer's request. It can also be described with stochastic variables.

- Structure of the system, specifying how many servers there are, if they are equivalent, if they are accessible or not, what the policy of use is, for example, if there are priorities or if servers can be shared or not.

We want to characterize in particular the variability related to stochastic arrivals because, beyond the models, what interests us is the role of variability in service systems: if the processes that feed and determine the operation of a service system are random, then this negatively impacts the quality of the service provided.

Therefore, we are interested in the role of variability in:

- Inter-arrival times

- Service times

Our goal is to try to balance the load to achieve the best possible overall performance: distributing the load wisely to achieve the best possible overall delay performance. We do not let users access the various servers in an uncontrolled way, and we also try to reduce the variability of arrivals (the distribution of arrivals to the dispatcher might be Poissonian, but the distribution of arrivals to one of the various servers might be entirely different).

Within service systems, there are two contrasting aspects:

- Quality of service

- System utilization

The user looks at the quality of service, which is better the less crowded the system is (if I am the only one using the system, then for me, the user, it would be perfect). The system provider wants a heavily utilized system because the system has a cost that must be amortized over the largest number of consumers to reduce service costs. We need to find a good compromise imposed by external constraints or by competition in which I try to improve myself; otherwise, I lose customers.

We begin by discussing how we chose scheduling for our project: the way resources are managed in a system to combat the randomness of arrival times and service times. Remember that the randomness of inter-arrival times and service times negatively impacts performance, particularly system response delays. The difference that service policy can make can be very significant when service times are highly variable. If service times are regular, then the impact of the scheduling policy is much lower than when service times are highly variable. However, when we implement it, we must also evaluate the difficulty of realization, for example, if we can do preemption. Whenever we know that service times can have a wide variability, we remember that the scheduling policy is fundamental within the limits of what we can achieve.

Scheduling, sometimes also called discipline/service policy, is a set of rules that determine how the available service capacity (resource) in a server is used (allocated) to serve one or more users waiting for service. So scheduling answers questions like "who gets served next?"

The response time is a random variable; if I submit a task at two different time points, it does not mean that it will take the same time to be executed. The execution time of that task will be the same given the same resources, but if I do it at two different time points, the total time = system response time might not be the same because it depends, for example, on traffic, waiting time, etc. Since response times can be characterized as random variables, it makes sense to talk about the average value. Another reason we talk about the average response time is that since the average response time is one of the system's performance indices, we can use the average system response time as a benchmark for comparison among various policies under the same conditions. Therefore, for us, the best policy will be the one that, under the same conditions, has a lower average response time.

Criteria for Classifying Scheduling Policies

Scheduling policies (on a single server) can be classified based on different criteria. We can define three dimensions for characterizing possible classifications:

- Classification based on the possibility of preemption (depends on the context, for example, in the cloud, this is possible, while in the case of a machine on an industrial automation line that has started working on a piece, it cannot be stopped).
  - A system is said to be non-preemptive if I cannot interrupt the service once it has been taken over (if I interrupt the service, it means I lose all the work I have done, and I will have to start over from scratch).
  - A system is said to be preemptive when I can interrupt an already started and not yet completed service and later resume it from where it was left off (e.g., reading a book).
- Classification based on the server's sharing capacity (in the case of TCP, the uplink channel of a mobile radio network is typically simultaneously shared among multiple users' flows. This is the case where the entire radio resource is simultaneously used by multiple users, whereas in the case of Wi-Fi, the entire radio channel is given to one user for a very short time each).
  - The policy is of the “head-of-line” / HOL type if I serve one user at a time.

- Server sharing, the server is divided and can serve multiple users in parallel, if not all at once. However, the more there are, the slower I transmit because I give an increasingly smaller capacity to each one.

- Classification based on knowledge of job size.

  - A system is non-anticipative if the job size is not known a priori: when the request arrives, I cannot immediately say, "this request can be fulfilled, having the entire server's capacity, in X seconds."

  - A system is anticipative if one can have an idea of how long a task will last with a fair degree of approximation (having an idea of how long a task lasts is very useful in scheduling), so a rough estimate but still an estimate of the job size. I can know the average value/distribution of times but not the exact value of that particular job.

We can have two types of systems:

- Concentrated and centralized system, that is, in each system (server), there is only the capacity to provide the service (no queues), so the waiting line is only at the dispatcher.

- System in which each server is a complete system, but instead of receiving all arrivals, it receives only the arrivals that the dispatcher directs to it.

4.1.2 – Comparison with Other Policies

What would be the disadvantages of an FCFS policy in this context?

FCFS is very easy to implement on systems and machines (like a buffer, you just need two pointers such that one points to the memory address of the first in the queue and another pointer points to the first free space; each time the first packet in the queue is taken, the pointer associated with the first in the queue is moved to the next packet in the queue, so the pointers move on the buffer in a circular way). Additionally, it is easy to explain and understand, making it a frequently chosen option.

If we adopt a FIFO policy, variability is "completely endured" because nothing has been done to combat it: the effect of variability is entirely endured, which is the price paid for the simplicity of the FIFO policy. This is a factor we considered, especially because our data analysis showed that service times are extremely variable.

What would be the disadvantages of an LCFS policy in this context?

In the case of LCFS, as ϱ increases, the variability of the waiting time increases more rapidly than the average waiting time. In LCFS, I serve the last one to arrive; in the case of a high load, the probability that someone is constantly bypassed by someone who enters the system after them becomes very high, leading to potentially enormous waiting times.

What would be the disadvantages of an SJF policy in this context?

SJF is non-preemptive, so if a user with a very short execution time arrives in a queue, the scheduler will choose this user because they have the shortest service time. However, if the server is currently serving another user with a very long service time, the new user must wait the entire time it takes for the server to complete the service of the user currently being served.

What would be the disadvantages of a POS policy in this context?

POS (Process of Sharing) is a policy where the server equally divides its capacity among all the users present in the system. If I have a service capacity of μ, I give a fraction of capacity μ/n to each of the n users. The fraction of capacity given to each user in the system changes based on the number of users, and we must also keep in mind that the number of users in the system is not fixed but changes over time. Therefore, depending on the moment in time considered, each user is given a different fraction of capacity. This POS scheduling scheme is considered a benchmark for scheduling policies because it is an idealized scheduling scheme (we cannot always implement the perfect division of capacity so fluidly) and it is also absolutely fair.

Which performs better overall, POS or LAS?

The answer is not straightforward (we cannot say that one scheduling policy always performs better than another) because it depends on the distribution of service times. We can use the hazard function to define classes of probability densities:

- DFR Class: All those probability densities where the hazard function is monotonically decreasing (if the device survives at the beginning, it is likely to last over time, a typical case for electronic devices). In this case, on average, LAS performs better than POS (provides a lower average response time).

- IFR Class: All those probability densities where the hazard function is monotonically increasing (the older the object gets, the more likely it is to break). In this case, on average, LAS performs worse than POS, but the expected value of the slowdown averaged over all possible LAS service times is better (we calculate R(x) / x and average this over all possible service times).

- Many distributions that are neither DFR nor IFR.

Therefore, for some classes of service time probability densities, LAS seems to be a good policy.

In our scheduling method, what is the trade-off in choosing a large or small Δ?

In choosing Δ, there is a trade-off between two opposing needs:

- As the size of the service quantum decreases, there is less waste of server capacity, but users will be in the queue for a longer time (thus increasing the waiting time and consequently the response time).

- As the size of the service quantum increases, users wait less time in the various queues, especially in the last queue which tends to become saturated, but it is difficult to reuse part of the quantum that has not been used by someone: if I assign Δ but the user requires Δ/2, the remaining part of Δ is lost.

What would have been the best policy?

In the case of LAS, the label is the obtained service time: how many seconds of service have you already received up to now? Each user in the system has a label indicating how many seconds of service they had received until that moment. The server prefers to serve the user with the smallest label at any given moment.

In the case of SRPT, being a preemptive policy, we assume that the service time is known (whereas in LAS it is not known). In SRPT, the label given to each user is x-a = how much service time do you require - how much service have you already received. We recall that the required service time - the service time already received = the remaining service time. So how does this policy work? At any given moment, the server prefers to serve the user who has the least remaining time to complete their service.

It can be demonstrated that this policy is the best: it provides the minimum average response time (also known as the average system time) among all policies that allow preemption and are also anticipative (know the service time). It is obvious that this policy performs better compared to policies that do not know the average service time because it has additional knowledge. In the paradoxical case where it performs poorly, it means we are using the additional information improperly compared to policies that do not know the average service time.

Therefore, with SRPT, we offer a lower average system time compared to all the policies we have seen (not only those that allow preemption but also those that do not). While it is not surprising that SRPT is better than FIFO (because FIFO does not know the service times and does not exploit preemption), it is not trivial that it is the best even among all non-preemptive policies.

SRPT is, however, a discriminatory policy because it prefers shorter jobs. Imagine two jobs entering the system: one requires a short time and the other requires a long time. The job requiring a short service time is served first. We know that preferring shorter jobs is the "correct" direction because it aims to improve the overall system performance. However, it is also true that processes requiring a lot of service are not as discriminated against as in other situations like SJF because as a process receives some service, its remaining processing time decreases, which is like saying that the process gains priority: the more a user is served, the more priority they gain.

We can imagine a user in this policy as an avalanche: starting slow and then going faster and faster. A client requiring a lot of service time is very penalized initially, but as they receive service quanta, they become increasingly favored; that is, the more service time they receive, the more service time they will get. This acceleration is precisely what makes this policy better than others, like LAS, for example.

Why didn't we adopt this optimal policy?

Because it is very difficult to know the users' service times in advance in a service system, whereas we want to implement a sufficiently general scheduling policy.

What are the disadvantages of adopting a JSQ dispatching policy in this context?

The JSQ policy is a non-anticipative policy but with the knowledge of state information. When a task arrives, the dispatcher sends a message to all servers asking how many tasks they have in the queue. This means that the servers must be able to monitor the number of tasks present.

However, we mentioned that JSQ is not an anticipative policy, so the servers cannot know how much time will still be needed to complete the tasks in the queue, but they know the number of jobs in the queue and send this number to the dispatcher. Suppose the dispatcher can gather information from all servers. In that case, it chooses the server with the smallest queue (but this does not necessarily mean it is the best solution, especially if the service times have high variability). If two servers have the same minimum queue value, the dispatcher randomly chooses between these two: the choice is made uniformly and randomly, ensuring a good load distribution.

4.1.3 – First Implementation LAS Policy with Tables

The assumptions we make are that:

- Our server is non-sharing, works on serving, preemptive, non-size-based.

- Infinite waiting line (no loss).

- Age of information is not relevant, for instance, we are not in a context of sensing where the age of the information collected by sensors is very important.

- Ours is an open system, meaning that users arrive from outside, stay inside the system, and then return outside.

- The type of preemption we use, called preemption resume, does not involve resource dissipation, meaning if I have performed part of the user's work but interrupt it, I then resume from the point where it was interrupted, so the work done is not lost: normally the preemption process is not free, this preemption work requires dead times necessary to swap processes within memory, copy the state to memory, reload it into the system caches (maybe these dead times are small compared to the service time quanta but they are still present), and memory space to allocate processes on standby.

- Servers are all capable of doing what I need (servers are interchangeable); what can vary from one server to another is its speed of execution/service but not their ability to perform different tasks, meaning all servers can do the same things. However, it is obviously not always the case in real scenarios that servers are always interchangeable (because they may have different specialized tasks), but if servers have different tasks, then the dispatcher is limited.

We don't consider possible variants like:

- Impatient customers who leave the system before getting their service request completed.

- Strategic customers (we mix economic concepts with queuing so that customers evaluate if it is convenient for them to enter the system).

Suppose the computing resource assigned is not a physical machine but a virtual machine: an environment that emulates an operating system in all respects, so it is like a physical machine except it is software. The utility of a virtual machine is twofold because we can allocate resources assigned to a virtual machine much more flexibly (I buy a physical machine with a given number of cores, RAM, SSD, etc., and make larger or smaller slices of the physical machine to then assign them to tasks of various sizes). Additionally, I can move a virtual machine between multiple physical machines (I only need to move a large file), so I can reconfigure the physical machine (by expanding it) or shut it down to save energy/do maintenance. Therefore, having the freedom to have objects (servers) that appear to tasks in every way like physical machines but are not physical machines is very convenient because they can be reconfigured and moved.

From the outset, our idea was to immediately remove from the system users who have a relatively short service time (the typical task with a service time several orders of magnitude below 0 in the table provided as project material). However, since we decided to adopt a head-of-line, preemptive, non-size-based policy, we are not aware of the processing time of tasks arriving in the system, and for this reason, we have divided the service provision into service quanta.

All tasks initially enter the same queue and, according to a FIFO scheduling policy, are served by a first server that provides each task with a service quantum. Tasks that do not complete their service enter the end of the queue of the second server, and the entire process iterates until the task is served by the last server. If a task does not complete its service even after receiving a service quantum from the N-th server, it is put back at the end of the queue of the latter.

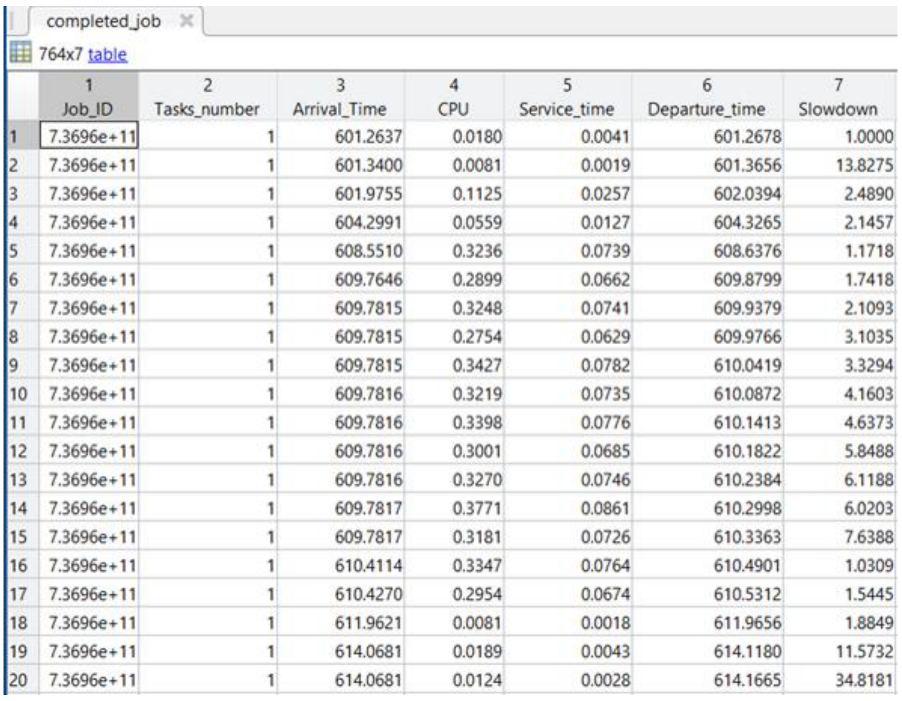
According to this dispatching policy, customers are served at regular intervals by the servers: all servers have the same capacity and operate according to the same time quantum. During a quantum, a server can be idle (if no tasks are in the queue or only one task is present but has already been served by the previous server) or provide a service quantum.

In the actual MATLAB code, it was decided to create a "servers" structure containing all the N servers, each with the following attributes:

- Capacity
- Utilization time
- A physical queue defined by a table with the following fields:
  - Job ID of the task
  - Unique task ID in the job
  - Task arrival time in the system
  - Task CPU time
  - Remaining task CPU time (at a given point in time)
  - Task service time within our system
  - Task age

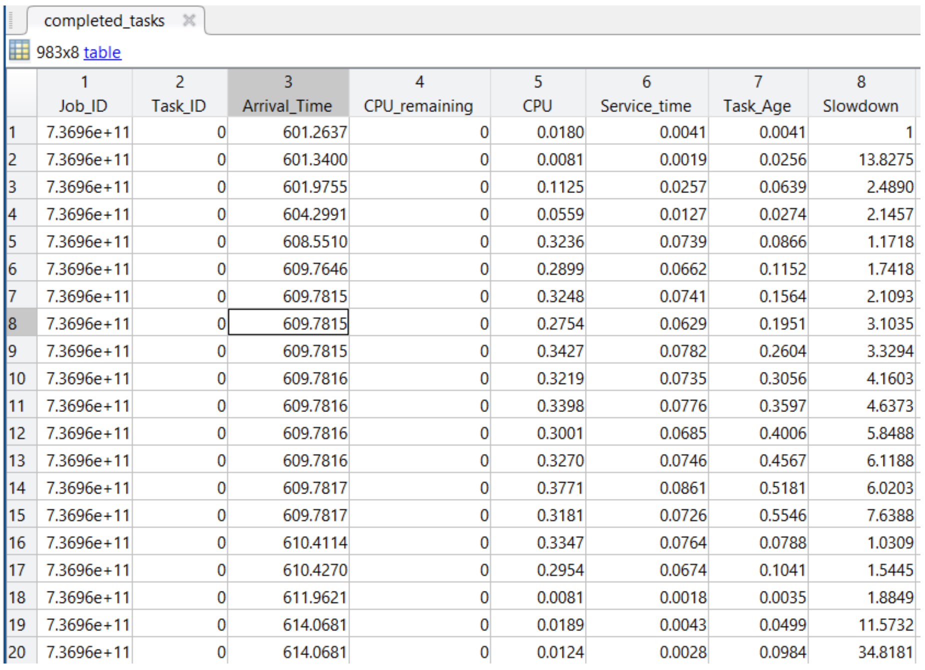
The advantage of this table is that we can track, step by step, the state of the various tasks in the system and if some of them cause an overflow of the entire system: very large tasks that go through all the servers and are reprocessed by the last server numerous times, causing a non-negligible slowdown of the entire system. Once a task is completed, its information is transferred to the completed tasks table, and once all tasks leave the system, the completed jobs table is built row by row by retracing the entire completed tasks table, considering that each job can be composed of various tasks and that the age of a job is given by the time interval between the first task of that job arriving in the system and the moment the last task pertaining to that job exits the system.

Completed Job Table:



	1 Job_ID	2 Tasks_number	3 Arrival_Time	4 CPU	5 Service_time	6 Departure_time	7 Slowdown
1	7.3696e+11	1	601.2637	0.0180	0.0041	601.2678	1.0000
2	7.3696e+11	1	601.3400	0.0081	0.0019	601.3656	13.8275
3	7.3696e+11	1	601.9755	0.1125	0.0257	602.0394	2.4890
4	7.3696e+11	1	604.2991	0.0559	0.0127	604.3265	2.1457
5	7.3696e+11	1	608.5510	0.3236	0.0739	608.6376	1.1718
6	7.3696e+11	1	609.7646	0.2899	0.0662	609.8799	1.7418
7	7.3696e+11	1	609.7815	0.3248	0.0741	609.9379	2.1093
8	7.3696e+11	1	609.7815	0.2754	0.0629	609.9766	3.1035
9	7.3696e+11	1	609.7815	0.3427	0.0782	610.0419	3.3294
10	7.3696e+11	1	609.7816	0.3219	0.0735	610.0872	4.1603
11	7.3696e+11	1	609.7816	0.3398	0.0776	610.1413	4.6373
12	7.3696e+11	1	609.7816	0.3001	0.0685	610.1822	5.8488
13	7.3696e+11	1	609.7816	0.3270	0.0746	610.2384	6.1188
14	7.3696e+11	1	609.7817	0.3771	0.0861	610.2998	6.0203
15	7.3696e+11	1	609.7817	0.3181	0.0726	610.3363	7.6388
16	7.3696e+11	1	610.4114	0.3347	0.0764	610.4901	1.0309
17	7.3696e+11	1	610.4270	0.2954	0.0674	610.5312	1.5445
18	7.3696e+11	1	611.9621	0.0081	0.0018	611.9656	1.8849
19	7.3696e+11	1	614.0681	0.0189	0.0043	614.1180	11.5732
20	7.3696e+11	1	614.0681	0.0124	0.0028	614.1665	34.8181

Completed Tasks Table:



completed_tasks

983x8 table

	1	2	3	4	5	6	7	8
	Job_ID	Task_ID	Arrival_Time	CPU_remaining	CPU	Service_time	Task_Age	Slowdown
1	7.3696e+11	0	601.2637	0	0.0180	0.0041	0.0041	1
2	7.3696e+11	0	601.3400	0	0.0081	0.0019	0.0256	13.8275
3	7.3696e+11	0	601.9755	0	0.1125	0.0257	0.0639	2.4890
4	7.3696e+11	0	604.2991	0	0.0559	0.0127	0.0274	2.1457
5	7.3696e+11	0	608.5510	0	0.3236	0.0739	0.0866	1.1718
6	7.3696e+11	0	609.7646	0	0.2899	0.0662	0.1152	1.7418
7	7.3696e+11	0	609.7815	0	0.3248	0.0741	0.1564	2.1093
8	7.3696e+11	0	609.7815	0	0.2754	0.0629	0.1951	3.1035
9	7.3696e+11	0	609.7815	0	0.3427	0.0782	0.2604	3.3294
10	7.3696e+11	0	609.7816	0	0.3219	0.0735	0.3056	4.1603
11	7.3696e+11	0	609.7816	0	0.3398	0.0776	0.3597	4.6373
12	7.3696e+11	0	609.7816	0	0.3001	0.0685	0.4006	5.8488
13	7.3696e+11	0	609.7816	0	0.3270	0.0746	0.4567	6.1188
14	7.3696e+11	0	609.7817	0	0.3771	0.0861	0.5181	6.0203
15	7.3696e+11	0	609.7817	0	0.3181	0.0726	0.5546	7.6388
16	7.3696e+11	0	610.4114	0	0.3347	0.0764	0.0788	1.0309
17	7.3696e+11	0	610.4270	0	0.2954	0.0674	0.1041	1.5445
18	7.3696e+11	0	611.9621	0	0.0081	0.0018	0.0035	1.8849
19	7.3696e+11	0	614.0681	0	0.0189	0.0043	0.0499	11.5732
20	7.3696e+11	0	614.0681	0	0.0124	0.0028	0.0984	34.8181

4.1.4 - B-LAS and SSRE-LAS Implementations

The queues

Since the goal of the LAS algorithm was to have a simulation that uses physical queues and can process the entire dataset, it was necessary to find an efficient way to manage the queues.

Our requirement was to manage simple FCFS queues, but with a high number of insertions and removals.

Managing a queue as an array by adding and removing elements with each task arrival or departure is inefficient because *MATLAB* shifts all array elements by one position with each removal.

For this reason, we implemented a Matlab class that we called ***Deque*** with the following features:

1. Efficient head insertion and tail removal.
2. Dynamic resizing.

3. Simplicity of implementation and modularity.
4. Efficient memory management.
5. Basic operations (head element access, queue length, is empty).

In summary, we can say that the removal operation has the highest computational complexity and, therefore, we have optimized it.

The method code is provided below, but only an intuitive explanation of its functionality will be given.

```
lassdef Deque < handle
    properties
        Data       % Array per memorizzare gli elementi della coda
        Head       % Indice della testa
        Tail       % Indice della coda
        Capacity   % Capacità massima dell'array Data
    end

    methods
        % Costruttore per inizializzare un oggetto Deque con una capacità specifica
        function obj = Deque(capacity)
            if nargin < 1      % Se non viene data come input una capacità
                capacity = 30; % Dimensione iniziale predefinita
            end
            obj.Data = cell(1, capacity); % Inizializza l'array Data con celle vuote
            obj.Head = 1;      % Imposta l'indice Head all'inizio
            obj.Tail = 0;      % Imposta l'indice Tail a 0 (indica che la coda è vuota)
            obj.Capacity = capacity; % Imposta la capacità
        end

        % Metodo per aggiungere un elemento alla fine della coda
        function enqueue(obj, element)
            % Se la coda è piena, ridimensiona l'array Data
            if obj.Tail - obj.Head + 1 == obj.Capacity
                obj.resize();
            end
            obj.Tail = obj.Tail + 1;        % Incrementa l'indice Tail
            obj.Data{obj.Tail} = element;   % Aggiunge l'elemento alla coda
        end

        % Metodo per rimuovere e restituire l'elemento in testa alla coda
        function element = dequeue(obj)
            % Se la coda è vuota, genera un errore
            if obj.isEmpty()
                error('La coda è vuota');
            end
            element = obj.Data{obj.Head};   % Assegna l'elemento in testa a element
            obj.Data{obj.Head} = []; % Rimuove il riferimento per liberare memoria
            obj.Head = obj.Head + 1; % Incrementa l'indice Head
        end

        % Metodo per verificare se la coda è vuota
        function isEmpty = isEmpty(obj)
            isEmpty = obj.Head > obj.Tail; % La coda è vuota se Head supera Tail
        end

        % Metodo per ridimensionare l'array Data quando la coda è piena
        function resize(obj)
            newCapacity = obj.Capacity * 2; % Nuova capacità
            newData = cell(1, newCapacity); % Crea un nuovo array Data più grande
            % Copia gli elementi dalla vecchia coda al nuovo array
            for i = 1:obj.Tail - obj.Head + 1
                newData{i} = obj.Data{obj.Head + i - 1};
            end
            obj.Tail = obj.Tail - obj.Head + 1; % Aggiorna l'indice Tail
            obj.Head = 1; % Resetta l'indice Head
            obj.Data = newData; % Assegna il nuovo array Data
            obj.Capacity = newCapacity; % Aggiorna la capacità
        end

        % Metodo per visualizzare l'elemento in testa alla coda senza rimuoverlo
        function element = peek(obj)
            % Se la coda è vuota, genera un errore
            if obj.isEmpty()
                error('La coda è vuota');
            end
            element = obj.Data{obj.Head}; % Restituisce l'elemento in testa
        end

        % Metodo per ottenere la lunghezza attuale della coda
        function len = getLength(obj)
            len = obj.Tail - obj.Head + 1; % Calcola la lunghezza della coda
        end
    end
end
```

Initially, an empty cell is initialized. With each task arrival, the task is added to the queue, and the Tail index is incremented.

With each departure, only the Head index is incremented.

When the queue reaches its maximum capacity, a new array of double the size is created, where all the elements of the old queue are copied, and the indices are reinitialized.

Below is an example of the functionality described.

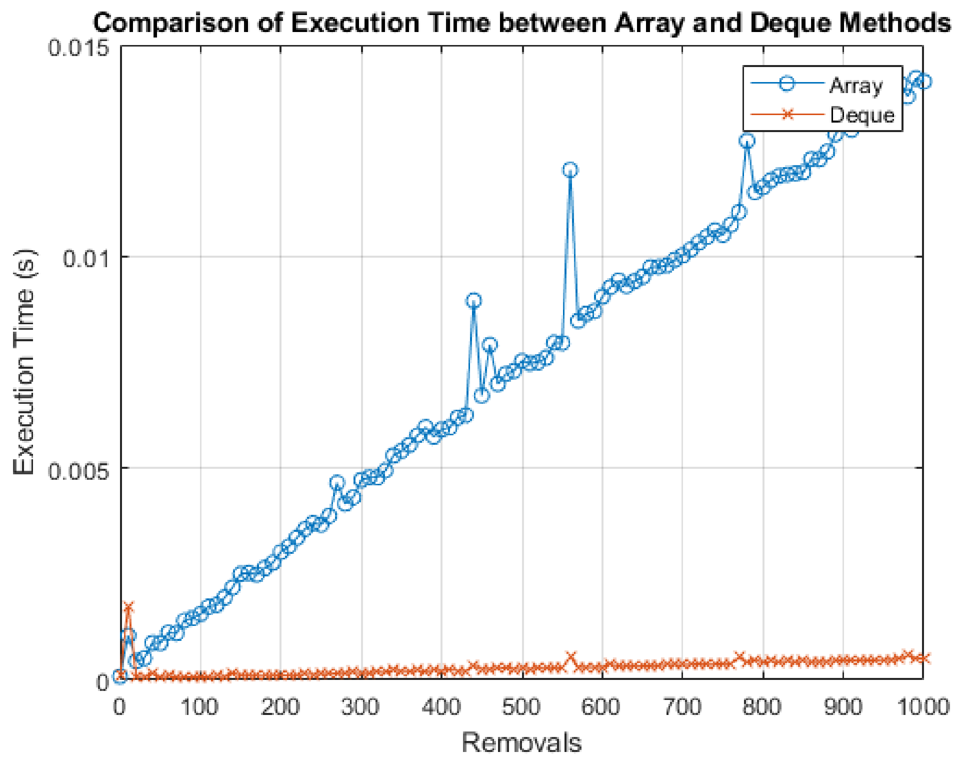
```
% Inizializzazione della coda con capacità iniziale di 3
dq = Deque(3);

% Aggiunta di elementi
disp('Aggiunta di elementi:');
dq.enqueue(10);
dq.enqueue(20);
dq.enqueue(30);
disp(dq.Data);                           % Output: {10, 20, 30}

% Visualizzazione dell'elemento in testa
disp('Elemento in testa (peek):');
disp(dq.peek());                         % Output: 10
```

```
% Rimozione di alcuni elementi
disp('Rimozione di elementi:');
dq.dequeue(); % Rimuove 10
dq.dequeue(); % Rimuove 20
disp(dq.Data);                              % Output: {[], [], 30}

% Verifica se la coda è vuota
disp('La coda è vuota?');
disp(dq.isEmpty());                         % Output: false

% Aggiunta di nuovi elementi per attivare il ridimensionamento
disp('Aggiunta di nuovi elementi per attivare il ridimensionamento:');
dq.enqueue(40);
dq.enqueue(50);
dq.enqueue(60); % Questa aggiunta attiverà il ridimensionamento
disp(dq.Data);                              % Output: {30, 40, 50, 60, [], [], [], []}
```

The advantage of this approach is that the only operation with O(n) (resizing) is performed infrequently, while all other operations have complexity O(1).

Operation	Classic Method	Deque
Insertion	O(1)	O(1)
Removal	O(n)	amortized O(1)

As we can see, the execution time in the classic case grows linearly, while with Deque it is almost constant.



The code: preliminary optimization

The program takes as input a CSV file with the following columns: Job_ID, Arrival_Time, CPU.

The Job_IDs each represent a task, and tasks of the same job have the same ID.

To save memory:

- The Job_IDs have been ordered from 1 onwards.
- The Arrival_Times have been ordered from 0 onwards while maintaining the same interarrival times.

All columns of the CSV are extracted, a log file is opened, and the total number of tasks is computed.

The technique of *extraction and access* (extracting a column from the CSV and accessing the column element instead of directly accessing the element *ij* of the table) allows for a reduction in the time needed to scan a column by 3,4 orders of magnitude.

In the end, the data table is cleared.

```
% Lettura csv e apertura log
fileID = fopen('LOG.txt', 'w');
Dati = readtable("Num_Sec_DaZero_No_Inter.csv");

% Estrazione colonne ID, CPU, Arrival Time, calcolo numero di task totali.
ID = Dati.Job_ID;        CPU = Dati.CPU;      ArrivalTime = Dati.Arrival_Time;
Ntask = height(Dati);    clear Dati;
```

Initialization of variables

The total computational budget, the computing power per server, and the service times for tasks and jobs are calculated, and the number of servers is chosen.

```
% Calcolo Budget Computazionale Totale (Nmi)
SommaCPU = sum(CPU);    BudgetComputazionale_Nmi = SommaCPU / (0.6 * (max(ArrivalTime)
- min(ArrivalTime)));
mi = BudgetComputazionale_Nmi / N;              % Potenza di calcolo di ogni server
N = 5; % Numero di server

% Calcolo dei tempi di servizio per task e per job, definizione Quanto
X = CPU / mi;                              % Vettore dei tempi servizio di tutte le task
Xjob = splitapply(@sum,X,findgroups(ID)); % Vettore dei tempi servizio di ogni job
```

```
quanto = 0.1;
```

Given that short tasks are predominant in the dataset, the **service quantum** has been chosen to be sufficient to complete between 40% and 70% of the tasks.

Data structures are initialized to manage queues and servers.

Empty queues are created for each server, and the initial idle state is set for all servers. The first arrival time and subsequent departure times for each server are initialized. Additionally, variables are prepared to monitor server occupancy times and job start and end metrics. Finally, the maximum arrival time is precalculated to save processing time.

```
% Code e Server
queue = cell(1, N);

for i = 1:N
    queue{i} = Deque(500);
end

server_idle = true(1, N);            % Stato di inattività dei server
next_arrival_time = ArrivalTime(1);  % Valore del primo tempo di arrivo
next_departure_time = inf(1, N);     % Prossima partenza di ogni server
queues_not_empty = false;            % True se esiste una coda non vuota

% Tempo Occupazione Server e metriche job
server_start_times = zeros(1, N);
server_busy_times = zeros(1, N);
job_start_times = zeros(max(ID), 1);
job_end_times = zeros(max(ID), 1);

max_arrival_time = max(ArrivalTime);
```

Simulation

```
while current_time < max_arrival_time || queues_not_empty || ~all(server_idle)
```

The simulation continues as long as the current time is less than the maximum arrival time, at least one queue contains tasks, or at least one server is occupied.

The code is composed of two blocks: **arrival management** and **departure management**.

Il codice è composto da due blocchi: gestione dell'arrivo e della partenza.

```
if task_index <= Ntask && next_arrival_time <= min(next_departure_time)
```

If the task index is valid and the next arrival time is less than or equal to the nearest departure time, the **arrival** of a task is managed; otherwise, the **departure** of a task is managed.

Arrival management

```
        % Assegna il job al server 1
        selected_server = 1;

        % Aggiungi il job alla coda del server selezionato
        queue{selected_server}.enqueue(task_index);
        queues_not_empty = true;

        % Registra il tempo di arrivo del job se è la prima task del job
        if job_start_times(ID(task_index)) == 0
            job_start_times(ID(task_index)) = next_arrival_time;
        end

        % Aggiorna il tempo corrente e l'indice del job
        current_time = next_arrival_time;
        task_index = task_index + 1;

        % Aggiorna il next_arrival_time
        if task_index <= Ntask
            next_arrival_time = ArrivalTime(task_index);
        else
            next_arrival_time = inf;
        end

        % Se il server era inattivo, avvia il job per un quanto di servizio
        if server_idle(selected_server) == true
            server_idle(selected_server) = false;
            server_start_times(selected_server) = current_time;
            next_departure_time(selected_server) = current_time + quanto;
        end
else
```

- The arriving task is assigned to server 1 and added to its queue.
- If it is the first task of the job, the job's arrival time is recorded.
- The current time and the task index are updated, and the next arrival time is calculated.
- If the server was idle, the job is started for a specified service quantum, and the server's next departure time is updated.

It is noted that the job start time can be calculated in this way because job start time is initialized to zero and is not updated in other parts of the code. Therefore, upon the arrival of the first task of the job, the job start time is updated, and since it is no longer zero, it will not be updated again. This way, the job start time is calculated only upon the arrival of the first task of the job.

Departure management

```
else
        %%% Gestione partenza %%%
        [min_departure_time, departing_server] = min(next_departure_time);
        current_time = min_departure_time;

        if queue{departing_server}.isEmpty()
            next_departure_time(departing_server) = inf;
            server_idle(departing_server) = true;
            queues_not_empty = false;
            continue;
        end

        % Seleziona il primo job dalla coda del server
        completed_task = queue{departing_server}.peek();

        % Riduci il tempo rimanente del task
        remaining_time = X(completed_task) - quanto;
```

The next minimum departure time and the corresponding server are found, the current time is updated.

If the queue of the departing server is empty, the server is marked as inactive, and the next departure time is set to infinity and the variable `queues_not_empty` is set to false. Otherwise, the first job in the server's queue is selected, and the task's remaining time is reduced based on the service quantum.

Now, two situations are distinguished: if the remaining time is negative, the task is completed and exits the system; otherwise, it is moved to the next server.

```
if remaining_time > 0

            % Task non completato, aggiorna il tempo rimanente
            X(completed_task) = remaining_time;

            % Rimuovi il task dalla coda corrente
```

```
        completed_task = queue{departing_server}.dequeue();

        % Trova la coda successiva

        % Variante Base
        next_server = min(N,departing_server+1);

        % Variante Feedback Inverso
        if departing_server == N
            next_server = 2;
        else
            next_server = departing_server + 1;
        end

        % Reinserisci il task nella coda del prossimo server
        queue{next_server}.enqueue(completed_task);
        queues_not_empty = true;

        % Se il prossimo server era inattivo,
         % avvia il job per un quanto di servizio
        if server_idle(next_server) == true
            server_idle(next_server) = false;
            server_start_times(next_server) = current_time;
            next_departure_time(next_server) = current_time + quanto;
        end

        % Aggiorna il tempo di occupazione del server e il tempo corrente
        server_busy_times(departing_server) = server_busy_times(departing_server) +
          (current_time - server_start_times(departing_server));

        % Aggiorna lo stato del server
        if queue{departing_server}.isEmpty()
            server_idle(departing_server) = true;
            next_departure_time(departing_server) = inf;
            queues_not_empty = false;
        else
            next_departure_time(departing_server) = current_time + quanto;
        end

    else
```

If the task is not completed, the task's service time is updated to the remaining time, and it is removed from the current server's queue.

The next queue is determined using two variants: in the **base variant**, the task is moved to the next server, while in the **reverse feedback variant**, if the current server is the last one, the task is sent to the second server; otherwise, it is sent to the next one.

The task is reinserted into the queue of the next server and, if the server was idle, it is started for a service quantum.

The server's occupancy time and the current time are updated.

If the current server's queue is empty, the server is marked as inactive and the next departure time is set to infinity; otherwise, the next departure time is updated to the current time plus the service quantum.

```
% Remaining Time < 0 (task completed)
Else
            % Task completato, rimuovilo dalla coda
            completed_task = queue{departing_server}.dequeue();

            % Calcola il tempo esatto di completamento della task
            actual_completion_time = current_time + remaining_time;

            % Registra il tempo di completamento del job
            job_end_times(ID(completed_task)) = actual_completion_time;

            % Aggiorna il tempo di utilizzazione del server
            server_busy_times(departing_server) = server_busy_times(departing_server) +
             (actual_completion_time - server_start_times(departing_server));

            % Aggiorna lo stato del server
            if queue{departing_server}.isEmpty()
                server_idle(departing_server) = true;
                next_departure_time(departing_server) = inf;
                queues_not_empty = false;
            else
                next_departure_time(departing_server) = current_time + quanto;
            end

        end
```

If the task is completed, it is removed from the server's queue. The exact completion time of the task is calculated and recorded. The server's usage time is updated with the task's actual completion time. If the server's queue is empty, the server is marked as inactive and the next departure time is set to infinity; otherwise, the next departure time is updated to the current time plus the service quantum.

```
        % Aggiorna il tempo di occupazione del server
        server_start_times(departing_server) = current_time;
    end
end
```

After managing a task, the code updates the server's start time to the current time, ensuring that the server's usage times are correctly recorded for the next task.

Metrics

```
% Calcolo delle metriche
job_response_times = job_end_times - job_start_times;
job_slowdowns = job_response_times ./ Xjob;
mean_job_response_time = mean(job_response_times);
mean_job_slowdown = mean(job_slowdowns);
utilization_coefficients = server_busy_times / current_time;
mean_utilization_coefficient = mean(utilization_coefficients);
```

The job response times are calculated as the difference between the job end times and job start times. The job slowdowns are then calculated by dividing the response times by the total service times of each job.

Next, the average job response times and slowdowns are calculated.

Finally, the server utilization coefficients are determined as the ratio of server occupancy times to the current time, and the average of these utilization coefficients is calculated.

Logs

It is important to note that this code is a complete simulation that tracks the progression of each task.

The version presented is not the most comprehensive one available; in fact, there is a version with a detailed log that tracks the entire system's evolution and stores the queue values. That version is not used for the calculation of metrics because it is computationally expensive.

Example of a complete Log

```
Numero totale di task: 10000
Numero di server: 5
Quanto di Servizio: 0.1500
Budget Computazionale Totale (Nmi): 16.5189
Potenza di calcolo di ogni server (mi): 3.30
```

```
-------------------------------------------------------------------------------------
Inizio Job: 1 - Al tempo: 0.000000
(current_time: 0.000000)
   Aggiornamento  |   Task: 2 - Prossimo Arrivo: 0.076269
(current_time: 0.000000)

Avvio Server: 1 per un quanto di servizio
   Inizio Occupazione: 0.000000 - Prossima Partenza: 0.150000

Tempo corrente: 0.000000 - Arrivo Task: 1 al server 1
Code: 1 0 0 0 0
-------------------------------------------------------------------------------------

-------------------------------------------------------------------------------------
Inizio Job: 2 - Al tempo: 0.076269
(current_time: 0.000000)
   Aggiornamento  |   Task: 3 - Prossimo Arrivo: 0.711769
(current_time: 0.076269)

Tempo corrente: 0.076269 - Arrivo Task: 2 al server 1
Code: 2 0 0 0 0
-------------------------------------------------------------------------------------
Task: 1 rimossa dal server 1 - Remaining Time: -0.144555
(current_time: 0.150000)
Tempo di quanto corrente: 0.150000 - Code: 1 0 0 0 0
Al Tempo: 0.005445 - Task: 1 completato dal server 1
           Server Busy Time 0.005445
           Server Busy Time 0.000000
           Server Busy Time 0.000000
           Server Busy Time 0.000000
           Server Busy Time 0.000000
Aggiorno Prossima Partenza Server 1 a 0.300000
-------------------------------------------------------------------------------------
Task: 2 rimossa dal server 1 - Remaining Time: -0.147547
(current_time: 0.300000)
Tempo di quanto corrente: 0.300000 - Code: 0 0 0 0 0
Al Tempo: 0.152453 - Task: 2 completato dal server 1
           Server Busy Time 0.007898
           Server Busy Time 0.000000
           Server Busy Time 0.000000
           Server Busy Time 0.000000
           Server Busy Time 0.000000
-------------------------------------------------------------------------------------
```

After many tasks

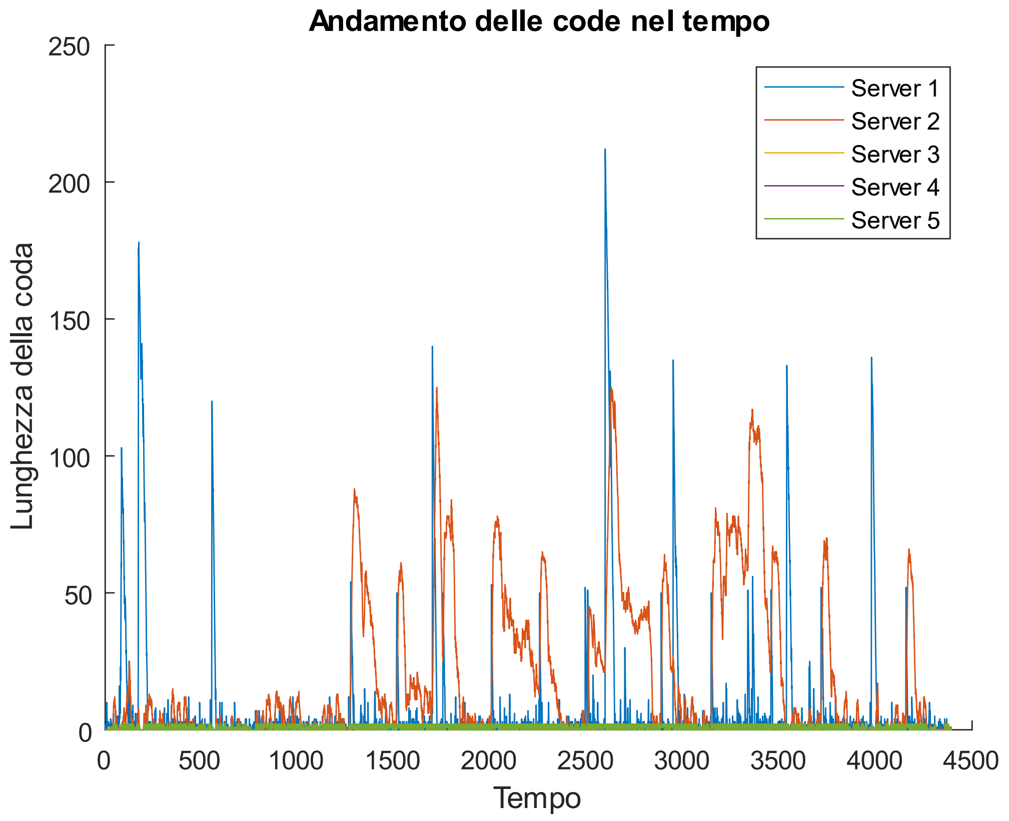
```
-------------------------------------------------------------------------------------
Task: 3276 rimossa dal server 1 - Remaining Time: -0.144382
(current_time: 1398.455023)
Tempo di quanto corrente: 1398.455023 - Code: 3 26 1 1 1
Al Tempo: 1398.310641 - Task: 2690 completato dal server 1
           Server Busy Time 209.872930
           Server Busy Time 845.718088
           Server Busy Time 810.981049
           Server Busy Time 780.427084
           Server Busy Time 750.748070
Aggiorno Prossima Partenza Server 1 a 1398.605023
-------------------------------------------------------------------------------------
Task: 3261 rimossa dal server 2 - Remaining Time: 0.356457
(current_time: 1398.477507)
Aggiorno Prossima Partenza Server 2 a 1398.627507
```

```
Tempo corrente: 1398.477507 - Code: 3 25 2 1 1
    Task: 2675 spostato dal server 2 al server 3
           Server Busy Time 209.872930
           Server Busy Time 845.868088
           Server Busy Time 810.981049
           Server Busy Time 780.427084
           Server Busy Time 750.748070
-------------------------------------------------------------------------------------
Task: 3253 rimossa dal server 3 - Remaining Time: 3.420298
(current_time: 1398.477507)
Aggiorno Prossima Partenza Server 3 a 1398.627507
Tempo corrente: 1398.477507 - Code: 3 25 1 2 1
    Task: 2667 spostato dal server 3 al server 4
           Server Busy Time 209.872930
           Server Busy Time 845.868088
           Server Busy Time 811.131049
           Server Busy Time 780.427084
           Server Busy Time 750.748070
-------------------------------------------------------------------------------------
```

At the end

```
-------------------------------------------------------------------------------------
Mean job response time: 8.907581
Mean job slowdown: 142.203690
Utilization coefficients: 0.159992 0.750457 0.720068 0.691861 0.665065
Mean utilization coefficient: 0.597489
Simulazione completata
Il tempo trascorso è: 0 ore, 0 minuti e 40.63 secondi
```

Example of a queue graph over time in the case of LAS reverse feedback



For the selection of the service quantum provided by each server, a balance was sought between two opposing tendencies:

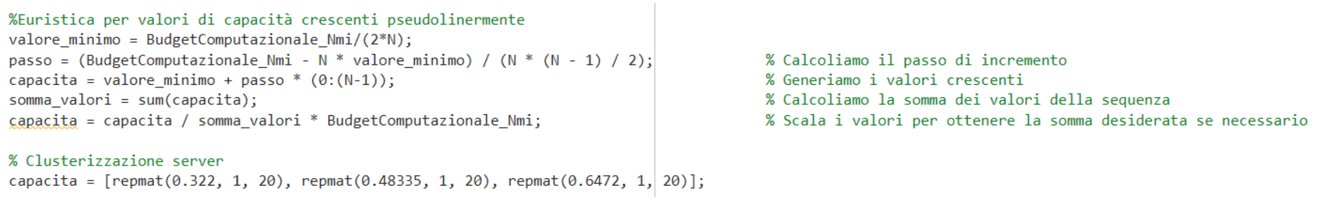
- Maximizing server utilization with small quanta: This approach aims to avoid wasting any potential time between the end of processing a task and the end of the quantum. This is important because, in the design phase of the policy, it was decided not to reallocate any remaining quantum to the next task.
- Considering the existence of a synchronization time at the start of the execution of each quantum: This synchronization time is negligible only for quanta that are not excessively small.

The search for this trade-off, driven by these theoretical foundations and evaluated through practical simulations, led us to consider it reasonable to set the service quantum of our servers to 0.1 seconds.

4.1.5 - DC-LAS Implementation

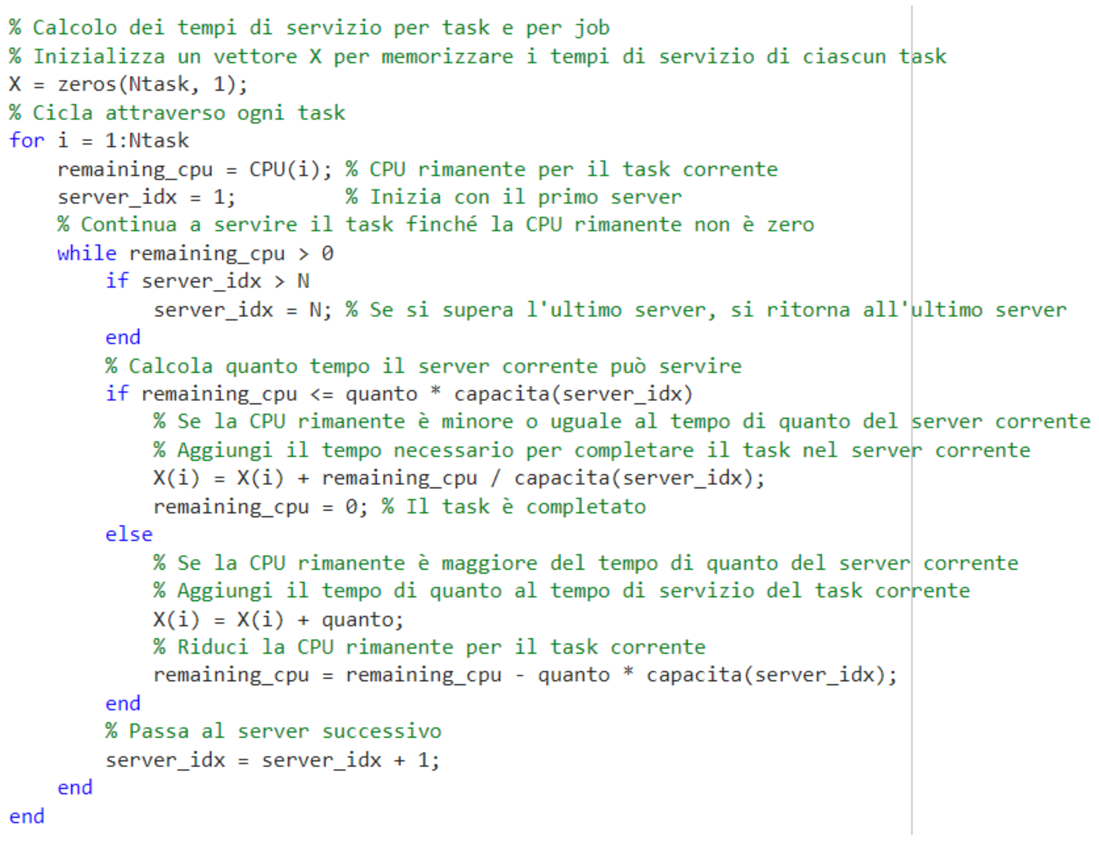
In this case, too, the core of the code remains substantially unchanged compared to B-LAS. The main adjustments are as follows:

- Pseudolinear increasing assignment of server capacity values



```
%Euristica per valori di capacità crescenti pseudolinermente
valore_minimo = BudgetComputazionale_Nmi/(2*N);
passo = (BudgetComputazionale_Nmi - N * valore_minimo) / (N * (N - 1) / 2);           % Calcoliamo il passo di incremento
capacita = valore_minimo + passo * (0:(N-1));                                          % Generiamo i valori crescenti
somma_valori = sum(capacita);                                                          % Calcoliamo la somma dei valori della sequenza
capacita = capacita / somma_valori * BudgetComputazionale_Nmi;                         % Scala i valori per ottenere la somma desiderata se necessario

% Clusterizzazione server
capacita = [repmat(0.322, 1, 20), repmat(0.48335, 1, 20), repmat(0.6472, 1, 20)];
```

- Calculation of job service times,
  this represents the main additional difficulty and involves scaling down the initial service time required for the task (in units of 1 GNCU) by the service provided by each server the task passes through (in units of 1 GNCU, weighted by the server's own capacity) until the remaining time of the task reaches 0.

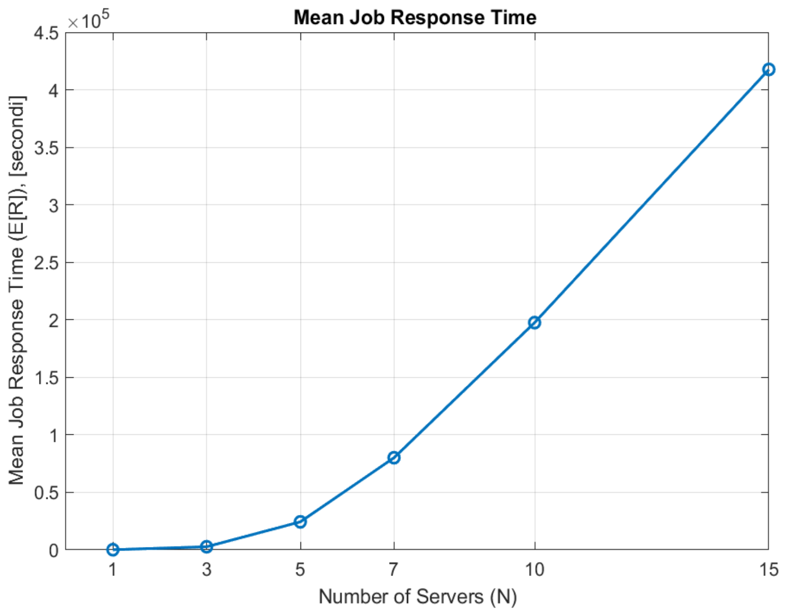


```matlab
% Calcolo dei tempi di servizio per task e per job
% Inizializza un vettore X per memorizzare i tempi di servizio di ciascun task
X = zeros(Ntask, 1);
% Cicla attraverso ogni task
for i = 1:Ntask
    remaining_cpu = CPU(i); % CPU rimanente per il task corrente
    server_idx = 1;         % Inizia con il primo server
    % Continua a servire il task finché la CPU rimanente non è zero
    while remaining_cpu > 0
        if server_idx > N
            server_idx = N; % Se si supera l'ultimo server, si ritorna all'ultimo server
        end
        % Calcola quanto tempo il server corrente può servire
        if remaining_cpu <= quanto * capacita(server_idx)
            % Se la CPU rimanente è minore o uguale al tempo di quanto del server corrente
            % Aggiungi il tempo necessario per completare il task nel server corrente
            X(i) = X(i) + remaining_cpu / capacita(server_idx);
            remaining_cpu = 0; % Il task è completato
        else
            % Se la CPU rimanente è maggiore del tempo di quanto del server corrente
            % Aggiungi il tempo di quanto al tempo di servizio del task corrente
            X(i) = X(i) + quanto;
            % Riduci la CPU rimanente per il task corrente
            remaining_cpu = remaining_cpu - quanto * capacita(server_idx);
        end
        % Passa al server successivo
        server_idx = server_idx + 1;
    end
end
```

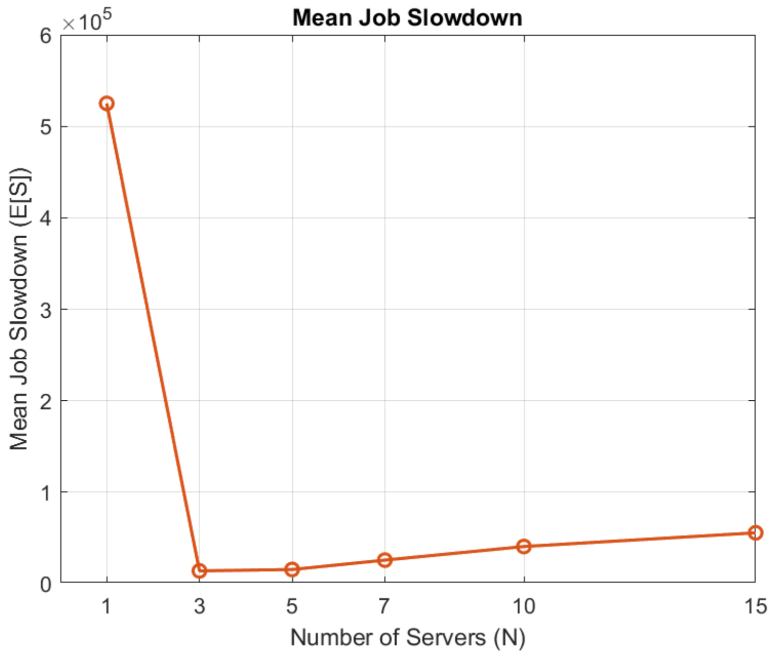
4.1.6 - Performance Analysis

B-LAS

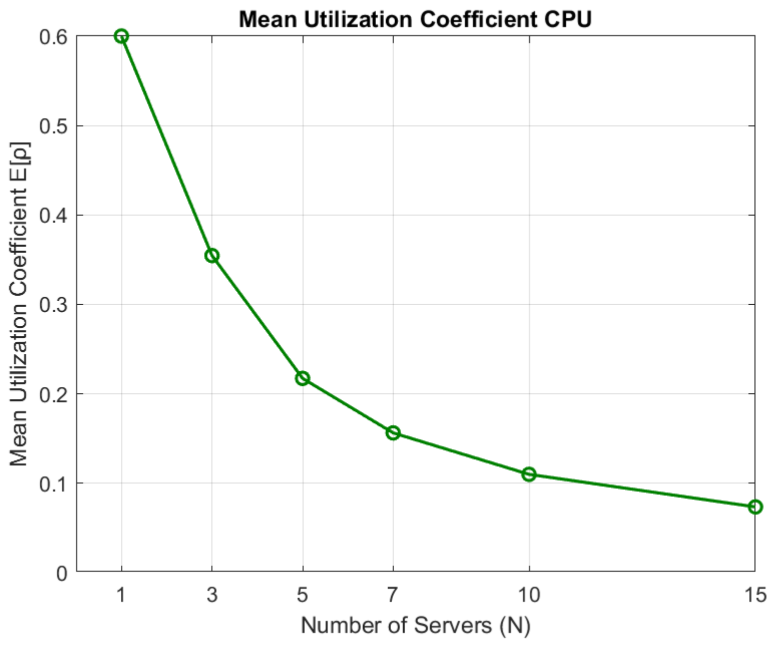
First, let's analyze the trend of the metrics of interest as the number of servers varies, all with the same capacity, derived from the simulation performed with our basic implementation (B-LAS). The following are the graphs of these trends, briefly commented, and a more exhaustive overall discussion in conclusion.



It can be observed that there is a polynomial increase in the average response time of the system as the number of servers increases. This divergence towards undesirable values led us to halt our simulation (burdensome in terms of execution time); however, generality was not lost in graphing its qualitative trend.



A drastic reduction in the average slowdown is observed when moving from 1 server to 3 servers, followed by a pseudo-linear increase as the number of servers increases.



A polynomial reduction in the average utilization coefficient of the system is observed as the number of servers increases. Specifically, it is important to report the utilization coefficients obtained in the simulation to highlight the causes of the results:

N = 1 : [0.600];

N = 2 : [0.051, 0.030, 0.981];

N = 5 : [0.036, 0.022, 0.019, 0.017, 0.992];

N = 7 : [0.029, 0.018, 0.015, 0.014, 0.013, 0.012, 0.994];

N = 10 : [0.024, 0.013, 0.012, 0.011, 0.010, 0.009, 0.009, 0.009, 0.009, 0.996];

N = 15 : [0.018, 0.010, 0.009, 0.008, 0.008, 0.007, 0.007, 0.006, 0.006, 0.006, 0.006, 0.006, 0.006, 0.005, 0.997].

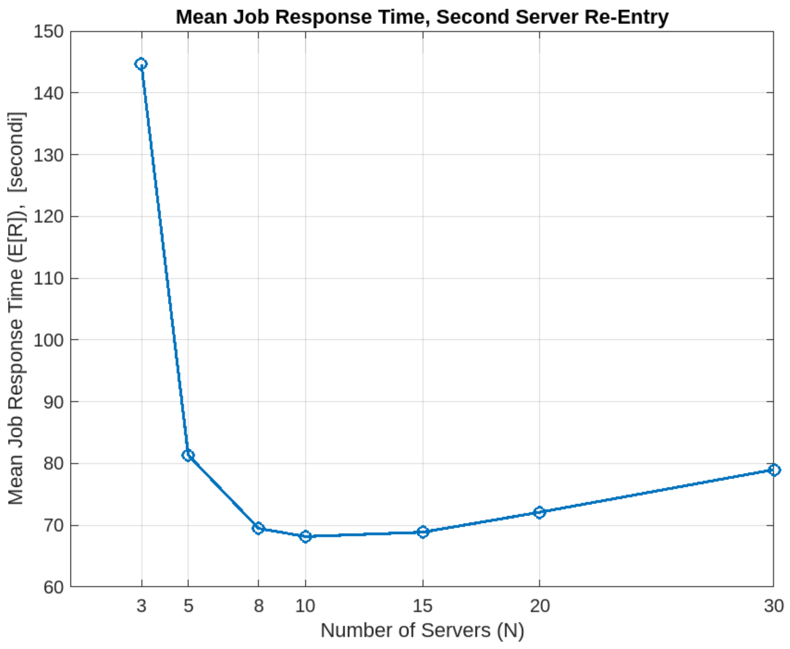
It is evident that there is underutilization of the first N−1 servers and an excessive load on the last server. This phenomenon becomes more pronounced as the number of servers increases, and this is responsible for the negative consequences on the metrics of our interest: the excessive load on the last server, which is asymptotically always in execution, indicates that there is always a queue at the last server. Specifically, the extremely high number of users in the queue outside of it (also evidenced by the simulations performed) leads to a huge response time for traversing this server. This causal sequence, which amplifies as the number of servers increases, is what we have identified as the main justification for the trends mentioned above. Therefore, the overload of the last server represents an extremely limiting bottleneck for the system in this situation. These considerations immediately justify the observed conclusions (increasing average response time, increasing average slowdown, and decreasing utilization coefficient).

For completeness of the analysis, it remains to explain the jump in the trend of the average slowdown between 1 and 3 servers before it starts to increase as mentioned above: with a single server, large tasks always fall into the first queue, limiting its ability to quickly process small tasks without causing them to endure a significant wait. As the number of servers increases, large tasks will move down to servers with higher indices until the last one, clogging that server, with the negative consequences mentioned above, but allowing the others to more quickly process the small tasks. In conclusion, the transition from the limiting single-server system (for various reasons explored later) to the more desirable multi-server system results in significantly worse overall performance, albeit more equitable.

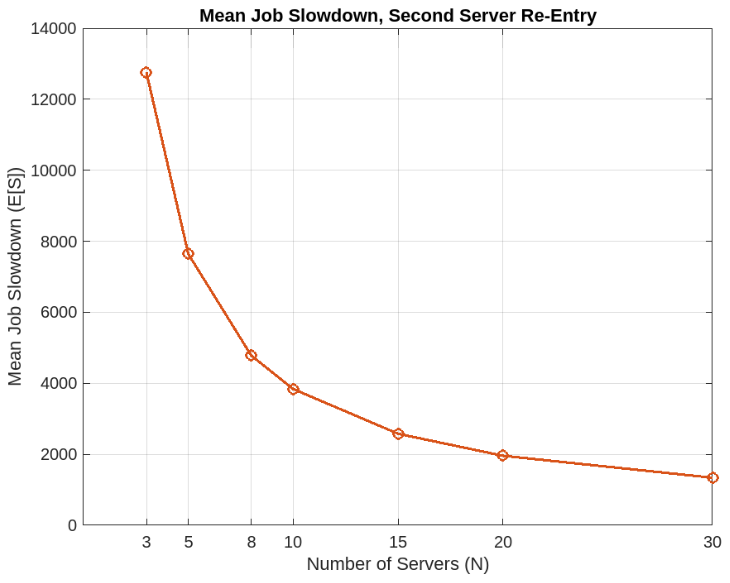
This conclusion, obviously unsatisfactory, led us to search for solutions to improve the system's performance. However, this analysis allowed us to identify an important problem and was therefore fundamental for the development of the project.

SSRE-LAS

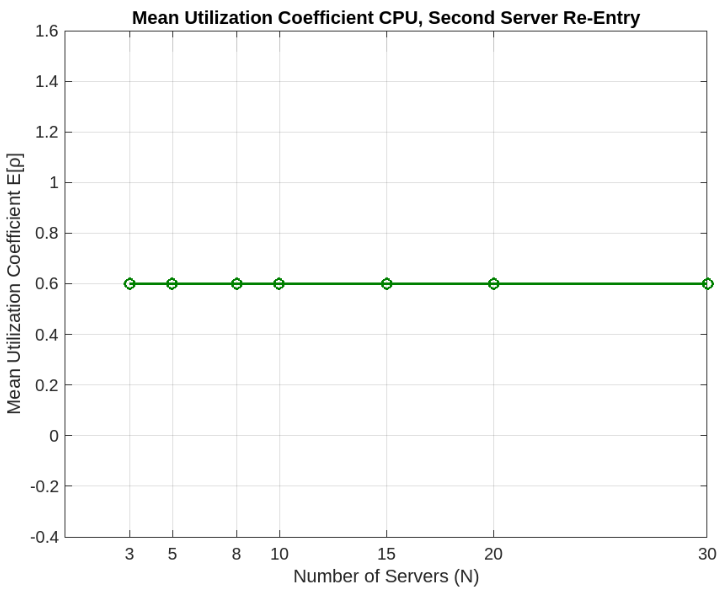
The search for appropriate variants of our B-LAS to solve the problem of the overload on the last server led us to design the two variants previously presented. The following graphs analyze the behavior of the SSRE-LAS implementation as the number of servers varies:



A significant improvement in the average response time is observed when parallelizing the system, due to the saturation of system utilization, with the first server being less utilized than the others but enabling the quick completion of small tasks. After N=8, a slight, gradual, and controlled increase is observed, which is not concerning. This is because, with all servers having equal capacity, as N increases, their individual capacity decreases, thus also that of the first server, which can only complete increasingly smaller tasks. Therefore, the mean response time remains good over a wide range.



The slowdown benefits from the parallelization of the servers. In fact, by assigning tasks to the second server, the increase in the number of servers ensures greater fairness.



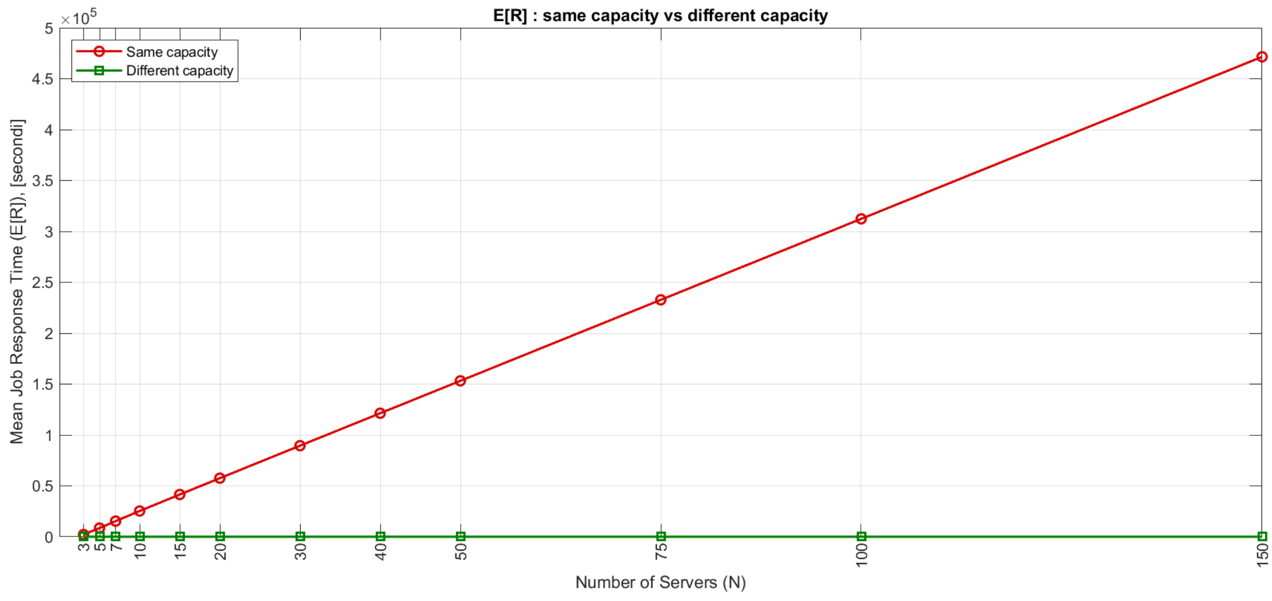
The average system utilization is always saturated at its maximum threshold and is distributed in a strong underutilization of the first server (compared to 0.6), which is nevertheless essential for completing short tasks without causing them to queue, and a slight overload (compared to 0.6) of all the others.

In conclusion of the SSRE-LAS implementation analysis, it can be stated that this implementation achieves a significant improvement in the metrics of interest, also taking advantage of the benefits of system parallelization, both in terms of performance and in terms of physical, economic, and technical evaluations of its possible construction. Lastly, and no less important, it is worth highlighting the robustness of these results, obtained from the simulation on the entire available dataset.

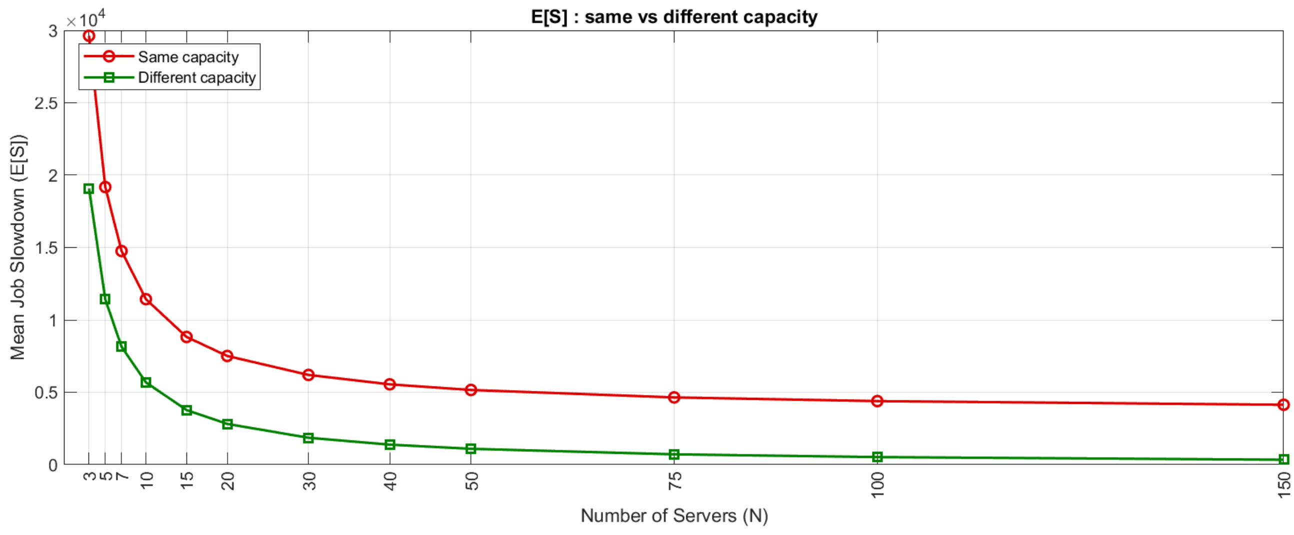
DC-LAS

Now, let's move on to the analysis of the results of the DC-LAS implementation. The following graphs highlight its performance as the number of servers varies, based on a sample of tasks equivalent to a typical workday of the system. Note that, to streamline the overall execution time of this new simulation, it was decided to work on this reduced sample of tasks only after having verified with broader simulations that the improvement impact of this variant was valid.

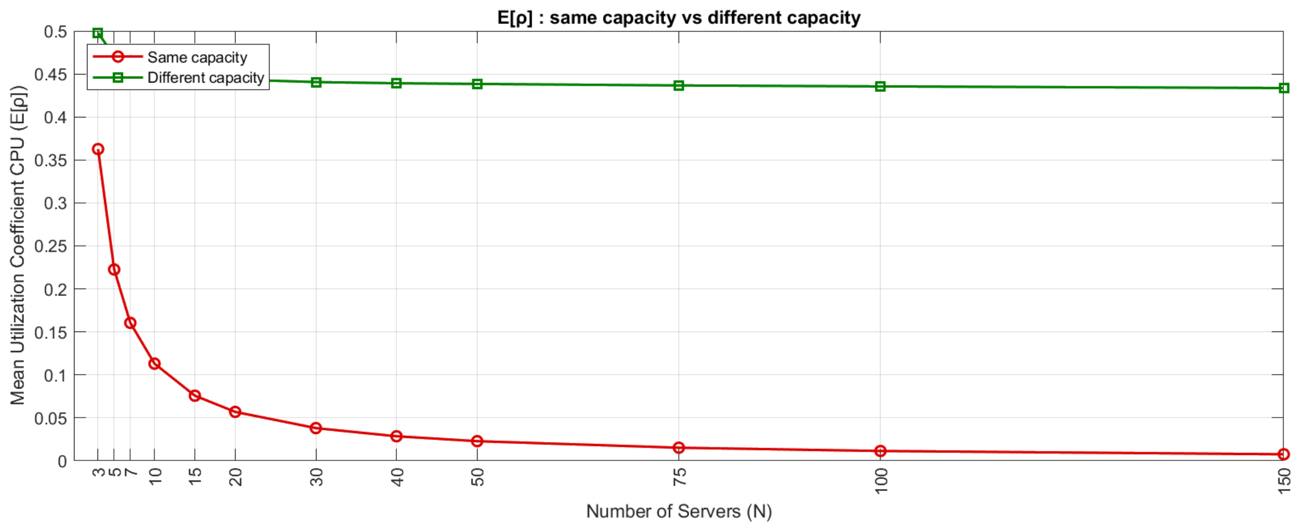
It is interesting to observe, first and foremost, the enormous improvements obtained in all the metrics of our interest with this variant compared to the base case of LAS previously discussed:



The average response time diverges in the 'same capacity' case (B-LAS) but remains stable and close to zero on this scale in the 'different capacity' case (DC-LAS).

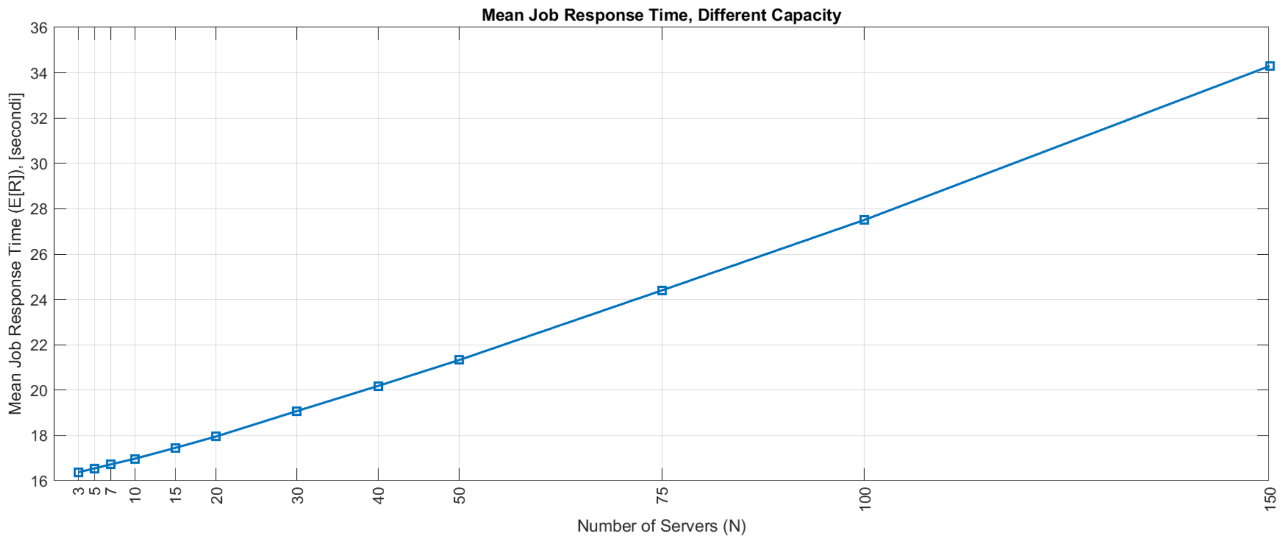


The two slowdowns have qualitatively similar trends; however, the values in the 'different capacity' case are always lower, and thus better, than those in the 'same capacity' case.

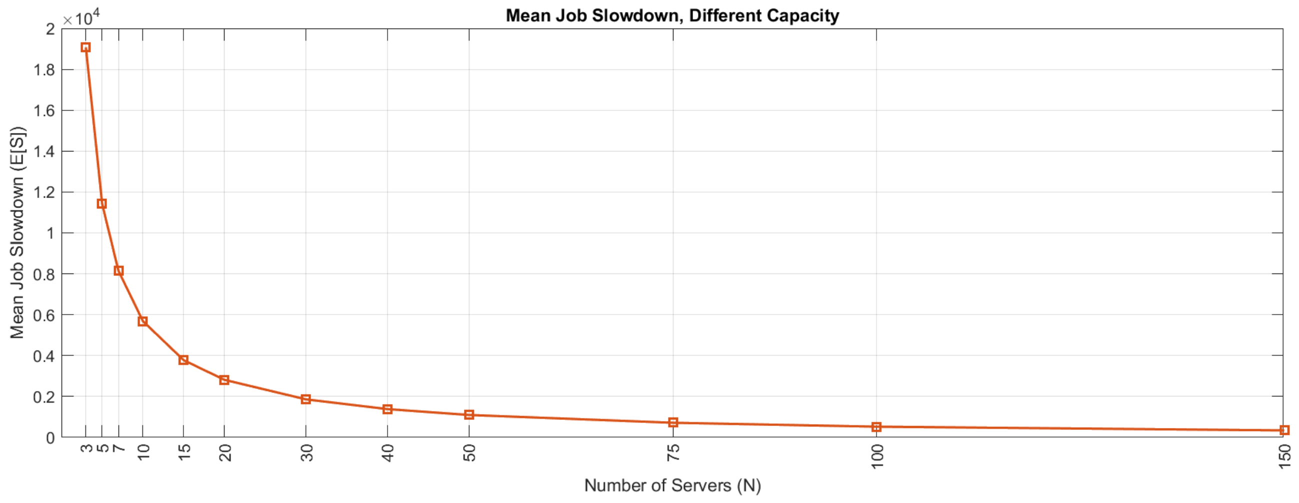


The utilization coefficient, a metric that was the subject of in-depth analysis in our previous study, has been stabilized in the 'different capacity' case.

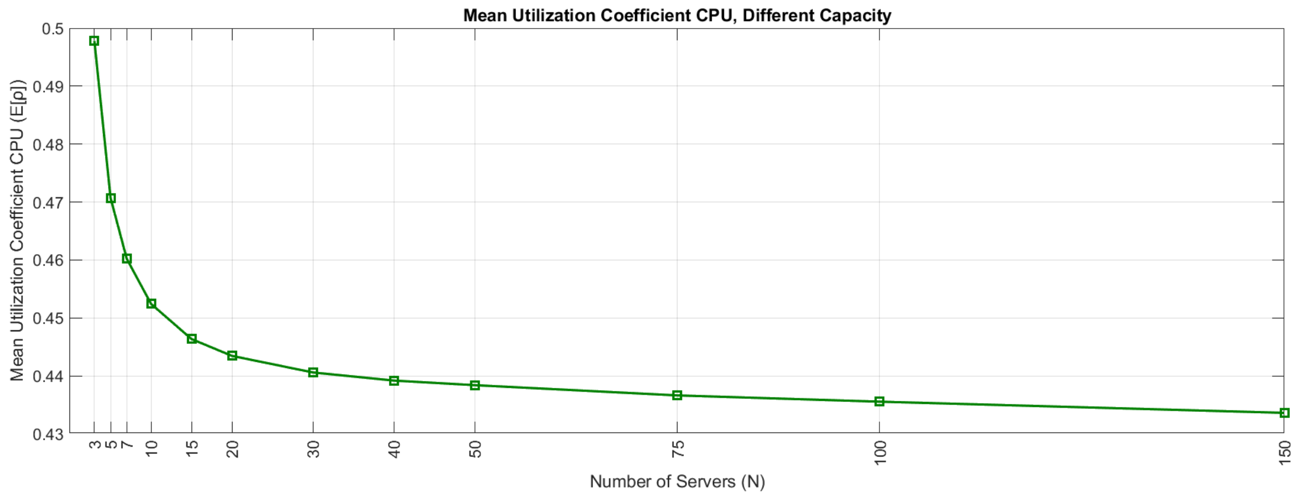
Based on the above, the DC-LAS implementation strongly dominates the previous B-LAS. Investigating the performance indices of this variant further:



A linear, yet significantly reduced, increase in the average response time is observed, within a range of values that can be considered good.



The slowdown exhibits a negative exponential-like trend concerning the values on the ordinate scale, with a significant reduction from 3 to 15/20 servers and a tendency to stabilize after these values. This highlights how the parallelization of servers allows smaller and larger tasks to be processed without the delay caused by executing large tasks, which, in the long run, occupy only the last server. Indeed, as the number of servers increases, due to the way capacity is assigned to them, the fraction of capacity dedicated to the last server—where large tasks accumulate—decreases. This phenomenon negatively impacts the average response time (caused by the significant increase in Job Response Time of large tasks, which consistently affect the overall average) and positively impacts the average slowdown (mainly influenced by the increasing number of tasks completing with low slowdown).



Finally, for the average utilization coefficient values, a more pronounced initial reduction is observed, which tends to stabilize for large values of N. This description of the qualitative trend should also be interpreted relative to the values on the ordinate scale, which constitute a narrow range. Therefore, it can be stated that this slight reduction, with values still fairly close to the maximum load, is not particularly significant.

As mentioned earlier, the reference metric for evaluating the quality of the designed system is the average response time, which achieves an optimal value in our simulation for N=3. However, considering that this metric remains fairly stable, given the noteworthy reduction in the average slowdown in the initial phase and the economic/physical/design advantages of a parallelized system, it is reasonable to reconsider the optimal value from our simulation, choosing N values in the range [40, 60] servers. A possible set of values is as follows:

Time interval considered	1 day
Task processed	226859
Number of servers (N)	60
Mean Job Response Time (E[R])	22,54 sec
Mean Job Slowdown (E[S])	906,19
Mean Utilizzation Coefficient of CPU (E[ϱ])	0,437

However, for a set of design, physical, economic, and technical reasons mentioned earlier regarding the choice of parallelization, it is also appropriate to consider a system with N different servers, each with a different capacity, as somewhat unrealistic. It is decidedly more realistic to consider the case of a division into classes of servers, with each class having servers of the same capacity. This practical revision of the designed variant, although it may not exclude a slight deterioration, still allows for acceptable and robust performance. In particular:

Time interval considered	1 day
Task processed	226859
Number of servers (N)	60
Mean Job Response Time (E[R])	26,17 sec
Mean Job Slowdown (E[S])	1267,58
Mean Utilizzation Coefficient of CPU (E[ϱ])	0,4787

In conclusion, it is essential to compare the metrics in the following identified realistic best cases:

Policy	JIQ – FCFS	DF-LAS
Time interval considered	1 day	1 day
Number of servers (N)	700	60
Mean Job Response Time (E[R])	161 sec	26 sec
Mean Job Slowdown (E[S])	46,3	1267
Mean Utilization Coefficient of CPU (E[ϱ])	0,35	0,47

Therefore, the key performance indicator (E[R]) shows a significant improvement of 83.8%. Moreover, the improvement impact would be even more evident as the time interval considered for our simulation increases: with a longer interval, the variance of service times increases, and as this variance increases, 'JIQ – FCFS' performs significantly worse than DC-LAS. Hence, our conclusions have not lost their general applicability. It is worth noting an evident deterioration in the average slowdown. However, overall, we can be satisfied with the result achieved: on average, it is better to lead everyone to better outcomes rather than equitably leading everyone to worse ones!

4.2 – Hybrid JSQ with dynamic server allocation

4.2.1-Algorithm description

A more complex algorithm is now proposed, utilizing multiple dispatchers to filter tasks based on their service times.

The algorithm can be classified as preemptive and non-size-based, allowing for the interruption and subsequent resumption of service for each individual task while it does not assume a priori knowledge over the service demand of the incoming tasks.

In analyzing the distribution of service times, it was observed that there is a predominance of "short" tasks, accompanied by a relatively smaller number of "long" tasks. These "long" tasks, however, demand substantially more service time than their "short" counterparts.

To address this, a differentiated treatment of tasks based on their service demand is proposed; however, to preserve the non-anticipatory nature of the algorithm, where the task length is not known a priori, a filtering system has been introduced.

This allows the algorithm to classify tasks according to their length, based on the service already given to each task.

Thresholds value where identified to classify tasks as short, medium or long based on their service times; the classification criteria are as follows:

- Short tasks:
  $x_i \leq SS_s$ (a generic task i is defined as short if its requested service time is lower than the threshold SS_s)
- Medium tasks:
  $SS_s \leq x_i \leq SS_m$ (a generic task i is defined as medium if its requested service time is between SS_s and SS_m)
- Long tasks:

$x_i \geq SS_m$ (a generic task i is defined as medium if its requested service time is greater than SS_m)

The algorithm utilizes three groups of servers (one for each group of tasks) denoted as small (S), medium (M) and large (L), each with its own dedicated dispatcher.

Servers and dispatchers are organized as illustrated in the diagram below:

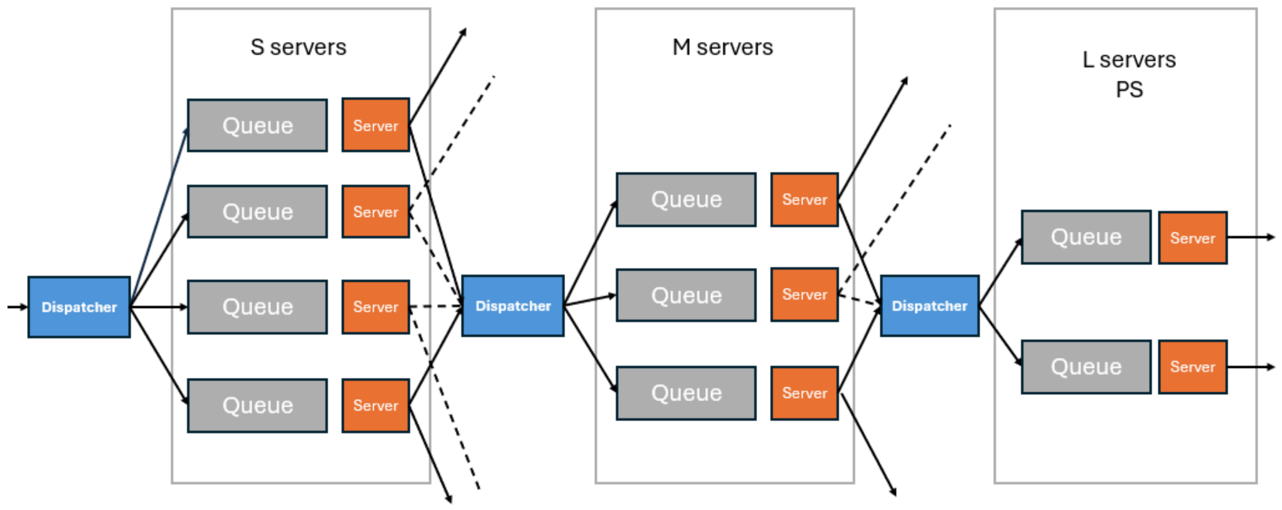


Figure: example of Hybrid JSQ system with 4 S servers, 3 M and 2 L.

To filter tasks, the servers in the S and M groups maintain a fixed service threshold, SS_s and SS_m respectively. This means that each server can provide a maximum amount of service defined by the corresponding thresholds; the setup result into two possible scenarios for an incoming task:

1) The service request is below the server's threshold: this implies that the task belongs to the server's corresponding length class, the server completes the task and it exits the system.
2) The service request exceeds the server's threshold: it means that the task belongs to a length class higher than the one of the considered server, consequently the task is redirected to the next dispatcher and remains in the system, with its service request reduced by the amount already provided.

At this stage, we can begin defining the dispatching and scheduling polices applied to each server group:

For the S and M servers, a classic Join Shortest Queue (JSQ) dispatching strategy combined with a First-Come First-Served (FCFS) scheduling policy on each server's queue is employed; by leveraging the Heavy traffic delay optimality (HTDO) we can assert that the load balancing will tend toward the optimality (at least under heavy loads).

For the L servers we introduce a JSQ dispatching combined with a Processor sharing (PS) model in which each server serves ''simultaneously'' (a simultaneous service is not feasible for a practical implementation so a different approach must be considered like the RR, but this issue will be discussed in detail later).

One of the strengths of this algorithm lies in enhancing the performance of a simple scheduling method like the FCFS through a filtering process: it is well-known that a major issue with the FCFS is its high sensitivity to increased variability in service times, which can be observed in the formulation of the response time for a single server, operating under an FCFS policy:

$$E[S] = E[X] + \frac{\rho}{1-\rho} E[X] \cdot \frac{1 + SCOV}{2}$$

Clearly an increment in the square coefficient of variation (SCOV) can proportionally increase the mean waiting time while a decrement is going to reduce that mean time.

This reduction can be achieved by correctly setting a threshold on the service delivered by each server; therefore, the choice of the threshold should aim to minimize the SCOV while keeping in mind the existence of a trade-off between this and the offered amount of service:

A very low value of variability for a group of servers obtained by setting a very low threshold on one hand decreases the SCOV (and indeed improves the performance of the

FCFS) but on the other implies that tasks receive a very limited amount of service, which transfer much of the variability to the next servers' group.

The criteria in the choice of the threshold were to keep the SCOV of the S and M group under 1 (an Upper Bound of threshold can be found analytically), since this leads to better performances of the FCFS over other policies like the PS, while minimizing the number of tasks that enters the L group.

Given that, the expected distribution of the tasks entering the L servers' group after the filtering process consists of a small number of tasks with a very large request of service and also a large variability (high value of SCOV).

A PS policy was adopted, instead of an FCFS, to address such distribution due to its robustness (insensitivity to SCOV) and its fairness that contributes to the reduction of the overall slowdown improving the equity of the system

4.2.2 - Servers' dynamic allocation

Through the data analysis, it was observed that on certain specific days the number of long tasks increases significantly compared to the total number of tasks on that specific day.

To further enhance the performance of the algorithm, a dynamic server allocation system was introduced:

Servers can be virtually reallocated among the S, M and L groups based on the prevalence of long tasks in the system, for instance, if the number of long tasks increases, some S and M servers (the exact number depending on the magnitude of the increase) will be transferred to the L group; conversely, if the number of long tasks decreases, servers might be reassigned to the S and M groups.

To implement this Dynamic transfer, several rules were necessary:

1) Transfer from FCFS groups (S or M) to PS (L): transfer cannot occur immediately since the server queue might not be empty. The proposed solution is to block the server's queue (preventing the dispatcher from assigning new tasks to the server) once the controller decides that a specific server needs to be moved and the transfer occurs only after the server has finished processing its current queue. Although this causes a delay in the responsiveness of the controller, the significant difference between the thresholds and the length of tasks considered long should mitigate the adverse impact on performance
2) Transfer from PS (L) to FCFS groups (S or M): when a transfer occurs, the queue of the transferred server is freed by sending again all the task inside to the dispatcher of the corresponding group (L) so they are going to be assigned to a new L's server. This allows the algorithm to move the server without a delay as in the previous case.

By implementing this virtual reallocation process, the algorithm can dynamically adapt to fluctuations of the tasks' service demand, optimizing resource allocation and maintaining efficient system performance while improving the overall flexibility of the algorithm.

4.2.3 - Benefits and drawbacks

The main Pros of this algorithm can be summarized as:

- High overall performance: the system provides, as further discussed in the result analysis, a lower response time and slowdown at least compared to simple policies like the JIQ or JSQ or the implemented LAS.
- High flexibility: thanks to the dynamic server allocation the algorithm can adapt even to huge variations in the distribution of the service time.
- Fair policy: the algorithm is also designed to minimize the expected slowdown by avoiding the prioritization of a specific class of tasks.
- Non anticipative: the algorithm does not assume to know the service request a priori.

- Easily scalable system: due to the dynamic server allocation process is also possible to easily increase (and decrease) the number of servers inside the system.

The main drawbacks are:

- Nonzero dispatching time: JSQ requires for the dispatcher the knowledge of the state of the queue of all servers before taking decisions and this generates a delay in the system that depends on the number of servers, the network latency and the dispatcher's speed in solving the problem
- Complexity of the algorithm: the algorithm is clearly more complex than a simple JIQ-FCFS or the proposed LAS and this exponentially increases the running time of the simulations (especially due to the use of a PS policy) but could also make a physical implementation more difficult
- Preemption: the algorithm works only if preemption is possible, but there may be a scenario in which that is not possible and indeed the algorithm cannot be implemented

4.2.4 - Future improvements

A lot can be done to further improve the algorithm performance:

- Multidimensional optimization of all the parameters used (that are going to be discussed in detail later)
- Dynamic thresholds and server switch parameters: by implementing dynamic thresholds and switching parameters (for example using ML algorithms) the system could be able to chase variations in the service time distribution by adapting over the variations of the SCOV and indeed dynamically optimizing such values based on what stated in the algorithm description.

As example (limit case) consider an initial situation like the one given by the data (which can be roughly considered as a pareto distribution), then over time the service demand distribution moves toward a distribution in which all the tasks have the same service's request ($var(x) \to 0$), in this case since the SCOV tends to zero the optimal policy is a simple FCFS. The system, to reach such configuration, should move all the servers to the S group but also has to be able to change the value of the threshold such that all the tasks are processed inside the S group.

4.2.5 - Implementation

Functioning

The main **parameters** of the algorithm are:

- **N** (the total number of servers): given that we imposed a **fixed load of 0.6**, N will be inversely proportional to the **capacity** of each one of the servers.

- **Nbig**, **Nmed** and **Nsmall** (respectively the number of L, M and S servers): they are initially defined as a portion of N, but their value is **dynamic** and they will change according to where additional capacity is needed.

- **ss** and **ss1** (respectively the **threshold** of server groups S and M): by fixing N, we can say that their value is directly proportional to the **waiting time** that creates in the first 2 groups of servers, while it is inversely proportional to the **running time** of the algorithm given that, once we serve all the "small" and "medium" tasks, the only group working will be the L one, so the algorithm automatically skips to the end of the service because we have no further arrivals and there is no possibility that the tasks will leave the servers without completing their service (see Process closure).

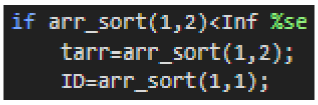
- **Levels**: we've defined a level for each type of server transfer (S to B, S to M, M to S, M to B, B to S and B to M) in order to determine the **minimum ratio** for which the transfer is required (this will be treated more specifically later). The higher these levels are, the lower will be the rate at which transfers happen.

These parameters can be evaluated also by looking at the **work** of each server, if the variability of the values related to servers' work is high then we have a poor optimization of server transfers, while in case of optimal parameters, this variability should be minimal.

- **Minimum active servers**: we define the minimum number of servers that we need to keep for each group (at least equal to 1), so that in case a transfer is needed, if the number of servers of the sender's group is already its minimum, then the transfer is cancelled. Obviously also these values have influence on the servers' **transfer rate**.

Given that we have all these parameters, together with the dimension of the dataset, the optimization of the algorithm would be very difficult as we're talking about **multidimensional optimization**. So, in order to reduce its complexity, we can locally optimize just a few parameters (like the number of servers and the thresholds) while manually adjusting the others according to the **desired behavior** of the system.

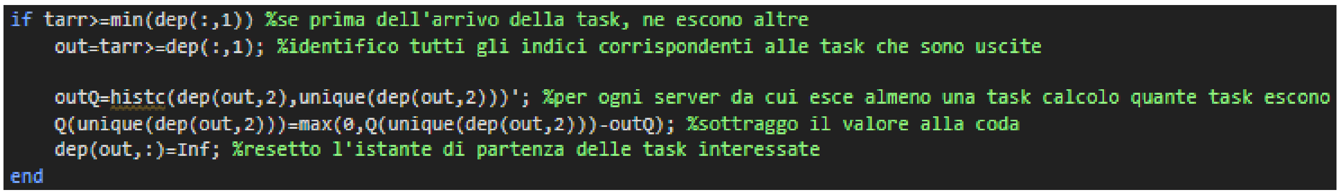
The algorithm runs until all the service times of the tasks (stored in vector **X**) are equal to 0, while the time is defined by considering the **inter-arrival times**. At each iteration the arrival time **tarr** and the **ID** of the next task are determined considering the first element of the arrival matrix (**arr_sort**), that gets sorted each time in order to have the closest arrival always in the first position, then we compute the IAT (**deltat**) by the difference between tarr and the previous instant **tnow**, that will then get updated with the current time.


```
if arr_sort(1,2)<Inf %se
    tarr=arr_sort(1,2);
    ID=arr_sort(1,1);
```

Deltat is used to update the **unfinished work** of the servers and the **response time** of the tasks located in the L group, where each server **equally splits** deltat between all the elements in its queue (for the first 2 groups, the response time can be immediately defined when the task gets assigned to a server as they follow a **FCFS** policy).

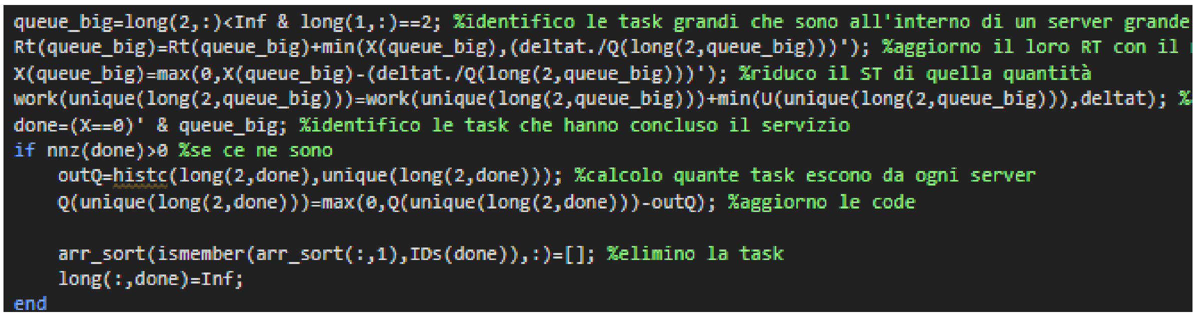
During this time, before considering the arrival of the task, 2 different events may happen:

- **Queue update**: if tarr is greater than at least one time instant in the departure matrix (**dep**), then it means that a task is ready to leave from a server. So, thanks to that matrix we identify the number of tasks that leave each server and we update their queues following these informations.



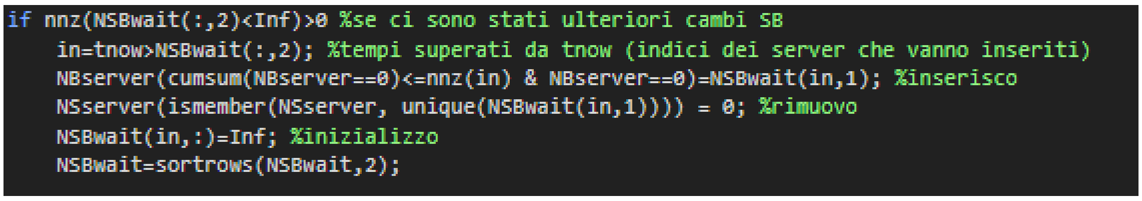
```
if tarr>=min(dep(:,1)) %se prima dell'arrivo della task, ne escono altre
    out=tarr>=dep(:,1); %identifico tutti gli indici corrispondenti alle task che sono uscite

    outQ=histc(dep(out,2),unique(dep(out,2)))'; %per ogni server da cui esce almeno una task calcolo quante task escono
    Q(unique(dep(out,2)))=max(0,Q(unique(dep(out,2)))-outQ); %sottraggo il valore alla coda
    dep(out,:)=Inf; %resetto l'istante di partenza delle task interessate
end
```

Note: this update mechanism works just with **S and M servers**, because L servers need to be treated differently due to their behavior. In fact their queues are updated by directly looking at how many tasks are completed each iteration, rather than looking at the departure time (that would be hard to define in the beginning expecially because of the **dynamic server allocation**).



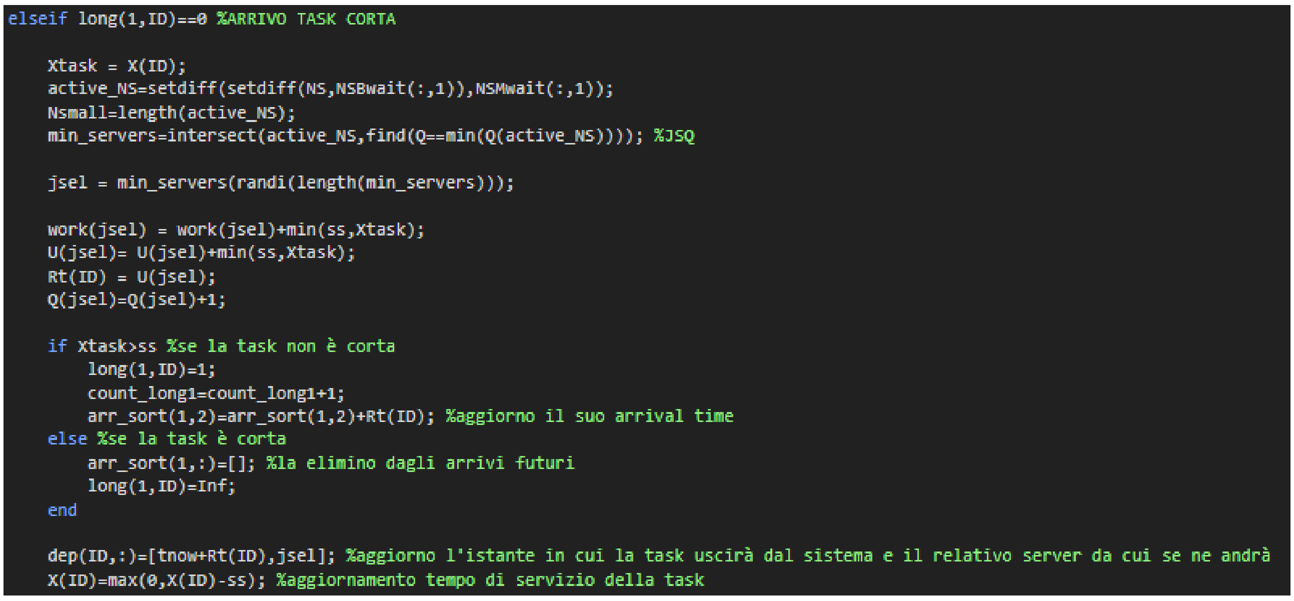
```
queue_big=long(2,:)<Inf & long(1,:)==2; %identifico le task grandi che sono all'interno di un server grande
Rt(queue_big)=Rt(queue_big)+min(X(queue_big),(deltat./Q(long(2,queue_big)))'); %aggiorno il loro RT con il
X(queue_big)=max(0,X(queue_big)-(deltat./Q(long(2,queue_big)))'); %riduco il ST di quella quantità
work(unique(long(2,queue_big)))=work(unique(long(2,queue_big)))+min(U(unique(long(2,queue_big))),deltat); %
done=(X==0)' & queue_big; %identifico le task che hanno concluso il servizio
if nnz(done)>0 %se ce ne sono
    outQ=histc(long(2,done),unique(long(2,done))); %calcolo quante task escono da ogni server
    Q(unique(long(2,done)))=max(0,Q(unique(long(2,done)))-outQ); %aggiorno le code

    arr_sort(ismember(arr_sort(:,1),IDs(done)),:)=[]; %elimino la task
    long(:,done)=Inf;
end
```

- **Server update**: if tarr is greater than at least one time instant in the server transfers waiting line, then the transfer gets validated and the number of active servers for each group is updated (the argument will be sufficiently exploited later).



```
if nnz(NSBwait(:,2)<Inf)>0 %se ci sono stati ulteriori cambi SB
    in=tnow>NSBwait(:,2); %tempi superati da tnow (indici dei server che vanno inseriti)
    NBserver(cumsum(NBserver==0)<=nnz(in) & NBserver==0)=NSBwait(in,1); %inserisco
    NSserver(ismember(NSserver, unique(NSBwait(in,1)))) = 0; %rimuovo
    NSBwait(in,:)=Inf; %inizializzo
    NSBwait=sortrows(NSBwait,2);
```

The arrival of the tasks also depends on the **long** matrix, that indicates the status of the task in terms of its **age of service** (the value assigned to the task is 0 if it's considered as small, 1 if it's medium and 2 if it's large), in this way we can identify the server group where it will be sent once arrived in the system. For example, if a small task arrives, it gets sent to the S group, where it will be allocated to a specific server following a JSQ policy and the **response time**, as well as the **service time** and the **departure time** (tracked by the dep matrix), will be updated following the **FCFS** policy. If the task is not concluded yet, it will be tagged as medium (long =1) and its **arrival time** will be updated.



```
elseif long(1,ID)==0 %ARRIVO TASK CORTA

    Xtask = X(ID);
    active_NS=setdiff(setdiff(NS,NSBwait(:,1)),NSMwait(:,1));
    Nsmall=length(active_NS);
    min_servers=intersect(active_NS,find(Q==min(Q(active_NS)))); %JSQ

    jsel = min_servers(randi(length(min_servers)));

    work(jsel) = work(jsel)+min(ss,Xtask);
    U(jsel)= U(jsel)+min(ss,Xtask);
    Rt(ID) = U(jsel);
    Q(jsel)=Q(jsel)+1;

    if Xtask>ss %se la task non è corta
        long(1,ID)=1;
        count_long1=count_long1+1;
        arr_sort(1,2)=arr_sort(1,2)+Rt(ID); %aggiorno il suo arrival time
    else %se la task è corta
        arr_sort(1,:)=[]; %la elimino dagli arrivi futuri
        long(1,ID)=Inf;
    end

    dep(ID,:)=[tnow+Rt(ID),jsel]; %aggiorno l'istante in cui la task uscirà dal sistema e il relativo server da cui se ne andrà
    X(ID)=max(0,X(ID)-ss); %aggiornamento tempo di servizio della task
```

Once the task rejoins the system, it will be marked as medium and sent to the M group, where it will follow the exact same procedure and eventually end in the L group where it will remain until it gets **fully served**, unless a server transfer takes place (see later). While if the task gets fully served in the S or in the M group, its arrival time will be removed from the matrix and its values in the dep and long matrices will be set to Inf (because we need them to be the same length as the total number of tasks in order to facilitate the indexing).

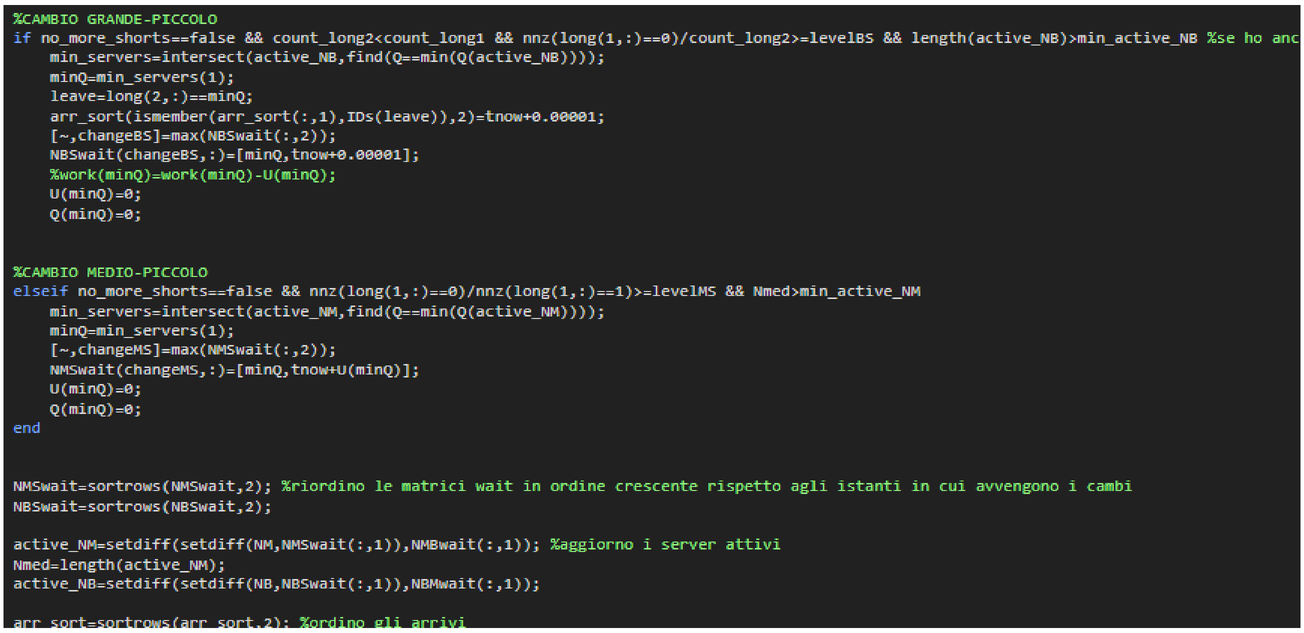
Dynamic server allocation

This function sort of recreates the phenomenon described by the **Halfin-Whitt** theoretical model, which states that if we increase both capacity and traffic the mean waiting time reduces as the **large scale** dominates variability. Obviously, this is just a light comparison because we do not work at an asymptotic regime and we also have constraints on load.

After a task gets sent to a server, a **transfer** directed to the group where the task get allocated is requested if the following **conditions** are satisfied (we take as an example the transfer towards the S group in order to make specific references):

- If the number of medium tasks (**count_long1**) is greater than the number of large tasks (**count_long2**), then the transfer that may happen will be from the M group to the S group (MS), otherwise it will be BS.
- The **ratio** between the number of small tasks (**nnz(long(1, :)==0)**) and the number of the sender group's tasks has to be greater or equal than a certain level (if we consider MS, then the ratio between small and medium tasks has to be greater than a value named **levelMS**).
- The **number of active servers** in the sender group must be greater than the minimum number of servers allowed (ex: **min_active_NM**), otherwise the transfer will be rejected or passed to the other group supposing its conditions are satisfied.

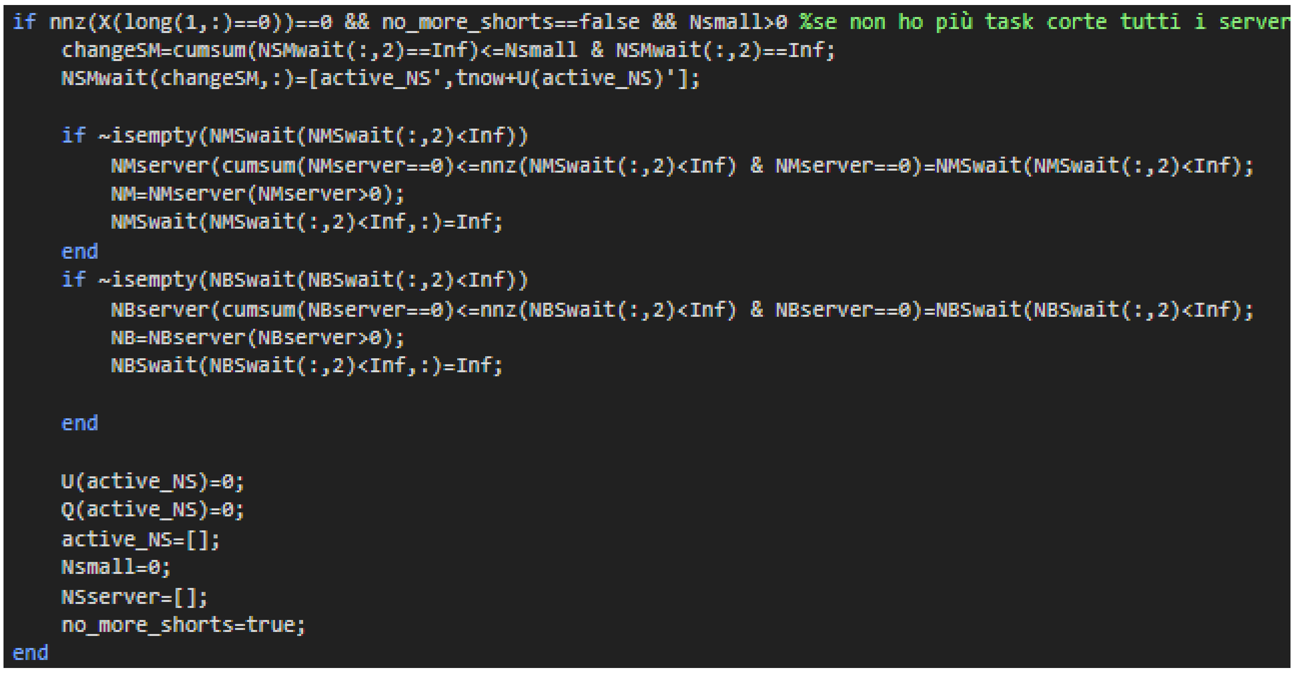
Once the transfer is authorized, the chosen server (the one with the **smallest queue**) will leave the active servers of its group (that will be updated as the difference between the total set of servers of that group and the servers located in the leaving transfer waiting lines, for example **active_NS** will be equal to the set of element that are present in NSserver and not in NSBwait or NSMwait) and join a **waiting matrix** (ex: **NMSwait**) that will report the server that will be transfered and the instant in which the transfer will happen. If we have a transfer coming from the S or the M group, it will take place in an instant defined by tnow plus the **unfinished work** of the server (obviously, considering the fact that the server is **not active** no more, it won't receive any upcoming task), while if the sender group is the L one, the tasks will temporarily leave the system, the transfer will be immediately validated and the tasks will then rejoin the L group getting allocated in different servers following the same JSQ policy as before.



```
%CAMBIO GRANDE-PICCOLO
if no_more_shorts==false && count_long2<count_long1 && nnz(long(1,:)==0)/count_long2>=levelBS && length(active_NB)>min_active_NB %se ho anc
    min_servers=intersect(active_NB,find(Q==min(Q(active_NB))));
    minQ=min_servers(1);
    leave=long(2,:)==minQ;
    arr_sort(ismember(arr_sort(:,1),IDs(leave)),2)=tnow+0.00001;
    [~,changeBS]=max(NBSwait(:,2));
    NBSwait(changeBS,:)=[minQ,tnow+0.00001];
    %work(minQ)=work(minQ)-U(minQ);
    U(minQ)=0;
    Q(minQ)=0;


%CAMBIO MEDIO-PICCOLO
elseif no_more_shorts==false && nnz(long(1,:)==0)/nnz(long(1,:)==1)>=levelMS && Nmed>min_active_NM
    min_servers=intersect(active_NM,find(Q==min(Q(active_NM))));
    minQ=min_servers(1);
    [~,changeMS]=max(NMSwait(:,2));
    NMSwait(changeMS,:)=[minQ,tnow+U(minQ)];
    U(minQ)=0;
    Q(minQ)=0;
end


NMSwait=sortrows(NMSwait,2); %riordino le matrici wait in ordine crescente rispetto agli istanti in cui avvengono i cambi
NBSwait=sortrows(NBSwait,2);

active_NM=setdiff(setdiff(NM,NMSwait(:,1)),NMBwait(:,1)); %aggiorno i server attivi
Nmed=length(active_NM);
active_NB=setdiff(setdiff(NB,NBSwait(:,1)),NBMwait(:,1));

arr_sort=sortrows(arr_sort,2); %ordino gli arrivi
```

There's also another trigger for server transfers, in fact once the small tasks are completed we won't need to keep the S group active, so we transfer all its servers to the M group and we share this new status with the entire system by turning the logical value **no_more_shorts** to true. This will also happen to the M group only after all the small tasks are completed (otherwise we cannot be sure that all the medium tasks are completed as well) and its server will all be sent to the L group.

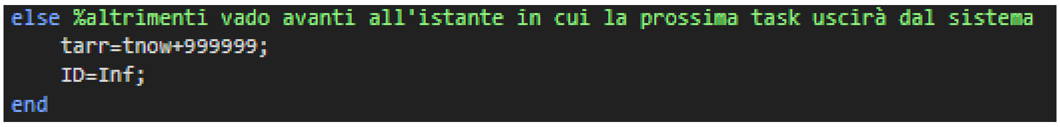
It's also important to check the presence of eventual transfers directed to an empty group that were authorized before the status update. In that case those servers will be sent back to the sender group or directly to the L one if also the sender has become empty in the meantime.



```
if nnz(X(long(1,:)==0))==0 && no_more_shorts==false && Nsmall>0 %se non ho più task corte tutti i server
    changeSM=cumsum(NSMwait(:,2)==Inf)<=Nsmall & NSMwait(:,2)==Inf;
    NSMwait(changeSM,:)=[active_NS',tnow+U(active_NS)'];

    if ~isempty(NMSwait(NMSwait(:,2)<Inf))
        NMserver(cumsum(NMserver==0)<=nnz(NMSwait(:,2)<Inf) & NMserver==0)=NMSwait(NMSwait(:,2)<Inf);
        NM=NMserver(NMserver>0);
        NMSwait(NMSwait(:,2)<Inf,:)=Inf;
    end
    if ~isempty(NBSwait(NBSwait(:,2)<Inf))
        NBserver(cumsum(NBserver==0)<=nnz(NBSwait(:,2)<Inf) & NBserver==0)=NBSwait(NBSwait(:,2)<Inf);
        NB=NBserver(NBserver>0);
        NBSwait(NBSwait(:,2)<Inf,:)=Inf;

    end

    U(active_NS)=0;
    Q(active_NS)=0;
    active_NS=[];
    Nsmall=0;
    NSserver=[];
    no_more_shorts=true;
end
```

Process closure

At some point we will have just large tasks, all being processed in the L group. This situation will be detected by looking at the arrival matrix, in fact if all of the arrival times are set to Inf (it means that all the tasks that are not done yet are inside group L), then tarr will be set equal to a value large enough to accelerate the process up to its end. This will strongly decrease the **running time** of the algorithm.

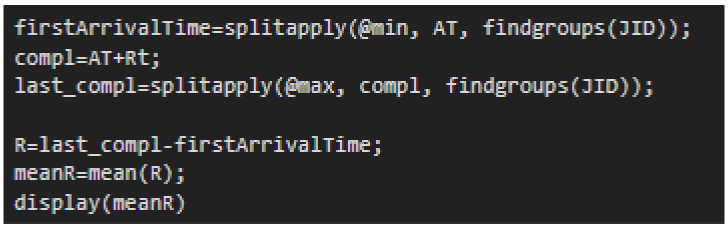


```
else %altrimenti vado avanti all'istante in cui la prossima task uscirà dal sistema
    tarr=tnow+999999;
    ID=Inf;
end
```

> **Note**: all the values that get updated during the serving process of the L group are **bounded**, so that the large deltat doesn't have any impact on the system's performances.

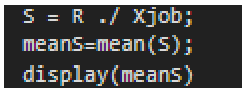
Metrics

After we complete all the tasks and we obtained the relative response time vector (**Rt**), we can compute the **response time** and the **slowdown** for each job. We start from determining the arrival instant of the **first task** for every job (**firstArrivalTime**), then we define a vector (**compl**) that reports the **completion instants** of every task (**arrival time + response time**) from which we find the **latest one** for every job (**last_compl**). Finally, we can create the job response time vector (**R**), that contains every job's response time computed as the difference between its last completion time and its first arrival time.



```
firstArrivalTime=splitapply(@min, AT, findgroups(JID));
compl=AT+Rt;
last_compl=splitapply(@max, compl, findgroups(JID));

R=last_compl-firstArrivalTime;
meanR=mean(R);
display(meanR)
```

To define the **slowdown** for every job we simply divide every member of the vector for the **job's total service time Xjob** (given by the sum of the service times of every task belonging to the job).



```
S = R ./ Xjob;
meanS=mean(S);
display(meanS)
```

The algorithm is evaluated based on the **mean values** of these 2 vectors.

Reality applications

The **Processor Sharing** policy is not actually possible to use in a real system, because is not possible to **infinitely** divide a time segment with quantums that go to zero. The actual implementation would be the **Round Robin** policy, where the quantum in finite. Actually, we could apply a **deficit RR**, that uses **diversified quantums** but in the long run each task gets the same amount of service like the PS. This kind of policy would also include a heavier **overhead** because the system needs to keep track of the age of service to follow a criterion in the allocation of quantums, also the policy wouldn't be **robust** anymore. In this case the two FCFS groups are very useful to reduce the amount of CPU coming into the last group of servers. Other important differences between our simulation and the reality are the amount of **dispatching delay** that creates in every group and the **costs of transfering a server**, in fact the latter one would include **setup times** and **disassembly activities** that would generate **additional costs** and **delay**, compromising the benefits created by the dynamic allocation of servers.

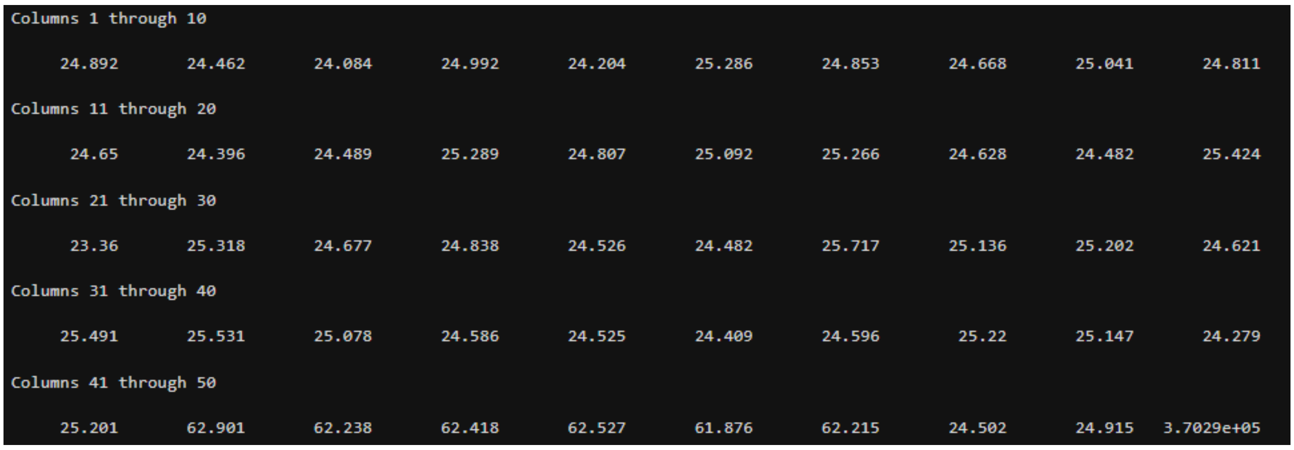
4.2.6 - Result analysis

Let’s consider actual results coming from the hybrid algorithm side by side with results from LAS:

- 100.000 tasks (Lower SCOV):

SCOV=25.14	**N**	**Mean RT**	**Mean SD**
Hybrid FCFS-PS (Not optimal):	50	12.466	0.819
LAS (Optimal):	60	12.943	556.665

Work (CPU):



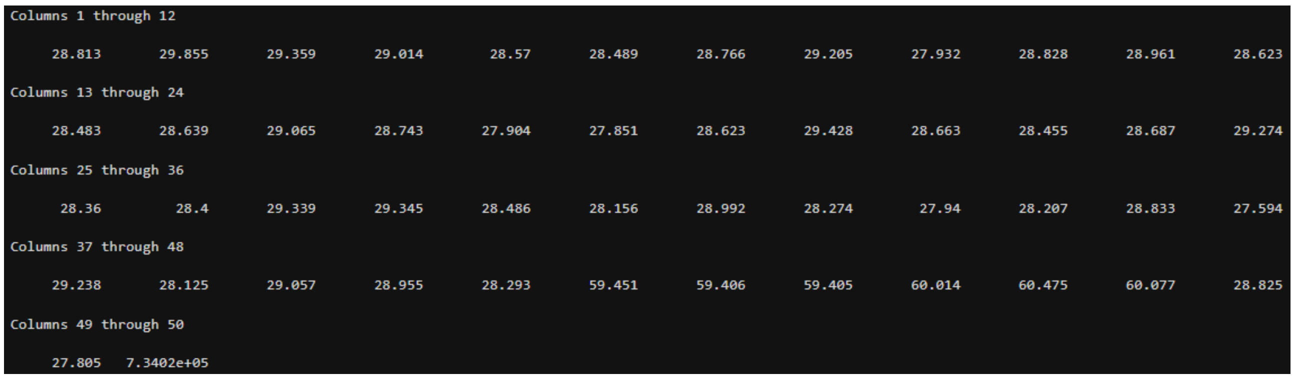
```
Columns 1 through 10

    24.892      24.462      24.084      24.992      24.204      25.286      24.853      24.668      25.041      24.811

Columns 11 through 20

     24.65      24.396      24.489      25.289      24.807      25.092      25.266      24.628      24.482      25.424

Columns 21 through 30

     23.36      25.318      24.677      24.838      24.526      24.482      25.717      25.136      25.202      24.621

Columns 31 through 40

    25.491      25.531      25.078      24.586      24.525      24.409      24.596       25.22      25.147      24.279

Columns 41 through 50

    25.201      62.901      62.238      62.418      62.527      61.876      62.215      24.502      24.915  3.7029e+05
```

In this case we used ss=prctile(X,10), ss1=prctile(X(X>ss)-ss,5) and low levels for server transfers to have more frequent server adjustments.

- 100.000 tasks (Higher SCOV):

SCOV=99.20	N	Mean RT	Mean SD
Hybrid FCFS-PS (Not optimal):	50	6.063	6.455
LAS (Optimal):	60	36.436	1743.340

Work (CPU):



```
Columns 1 through 12

    28.813      29.855      29.359      29.014       28.57      28.489      28.766      29.205      27.932      28.828      28.961      28.623

Columns 13 through 24

    28.483      28.639      29.065      28.743      27.904      27.851      28.623      29.428      28.663      28.455      28.687      29.274

Columns 25 through 36

     28.36        28.4      29.339      29.345      28.486      28.156      28.992      28.274       27.94      28.207      28.833      27.594

Columns 37 through 48

    29.238      28.125      29.057      28.955      28.293      59.451      59.406      59.405      60.014      60.475      60.077      28.825

Columns 49 through 50

    27.805  7.3402e+05
```

Here we used the same thresholds and the same levels for server transfers.

Due to the excessive **complexity** of the algorithm, it's difficult to conduct a proper analysis on the results given by the systems. But we can carry on a more theoretical analysis:

1) **Robustness**: given that the L group follows a PS policy, the hybrid algorithm is much less sensitive to variance (the small variations are mainly caused by the FCFS groups). So, if we properly optimize all the parameters in a way that minimizes the other groups' waiting time, we can obtain relatively similar results both with a small and a large variability of service times (up to a certain level established by the limits on the **FCFS groups**).

   While if we consider the LAS policy, we know that a variability increase (given a fixed load) has a **positive impact** on response time, although we do not have the necessary computational power to carry on a proper comparison based on a wider time horizon.

2) **Fairness**: the PS policy is always fair as everyone in a specific queue gets the same portion of capacity every iteration, while the LAS is discriminatory. This can be seen by the large gap between the mean slowdown values that grows as the number of large tasks increases.

3) **Dispatching delay**: if we consider the delay caused by the dispatchers, our system is more sensitive to this delay because of the multiple dispatchers (one for every group of servers).

4) **ST’s distribution**: in case of a different distribution of service times, our system is more flexible because of the dynamic allocation of servers. For example, if the ST’s **SCOV** significantly decreases, our system can easily work as a simple JSQ-FCFS system.

5. Conclusions and thoughts

In conclusion of this project experience, as a group we are satisfied with the work done and the results obtained as we managed to bring together our ideas and implement them, achieving acceptable outcomes. Despite our commitment, due to technical reasons, we had to settle for results on a reduced sample for some simulations, but along with the theoretical knowledge, we were still able to perform satisfactory analyses, justifying the missing technical component.