Virtual Resource Allocation in MEC Using Matching Algorithms

Quantitative Evaluation of Stochastic Models
Universtiy of Florence
July 8, 2025

Gregorio Piqué
gregorio.pique@edu.unifi.it

1. Introduction

This report, submitted as the final project for the *Quantitative Evaluation of Stochastic Models* course, investigates the application of matching algorithms to the problem of virtual resource allocation in *Mobile Edge Computing* (MEC) environments. The objective is to study how different algorithmic strategies for assigning Virtual Machines (VMs) to Physical Machines (PMs) can impact overall system efficiency, particularly with respect to energy consumption and resource utilization.

The definition of the problem and its formalization are based on the work presented in the paper *"Virtual Resource Allocation for Mobile Edge Computing: A Hypergraph Matching Approach"* [3]. In this work, the authors propose a weighted hypergraph-based optimization framework aimed at minimizing the total energy consumption of a MEC system.

In this project, however, an alternative direction is explored. Rather than adopting the hypergraph-based optimization approach proposed by the authors, the focus is placed on evaluating and comparing matching algorithms for the VM placement problem. The considered algorithms include a set of baseline methods (*Random*, *Greedy (First Fit)*, and *Round Robin*), as well as *Gale-Shapley (with dynamic preferences)* and an *Auction-based* algorithm. Each of these methods exhibits different trade-offs in terms of computational simplicity, load balancing behavior, and fragmentation characteristics, which may in turn affect overall resource efficiency.

The problem formulation used in this project follows that of the original paper—specifically, the system model, task definitions, VM/PM specifications, and energy consumption model. However, the core component of the solution, namely the algorithms used to perform the allocation, is different.

This report is structured as follows: Section 2 outlines the problem definition and system model. Section 3 presents the matching algorithms implemented for solving the allocation problem. Section 4 provides an overview of the class structures representing the MEC system and the algorithms. Section 5 details the experimental setup and discusses the results and key insights. Finally, Section 6 concludes the report with reflections and possible directions for future work.

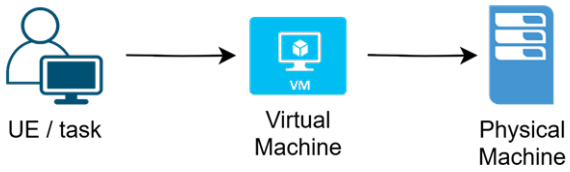

Figure 1. Simplified illustration of the resource allocation process.

2. Problem Formulation

This section introduces the problem definition and system model, presenting the key components involved in the virtual resource allocation for MEC. The problem formulation follows the framework described in the reference paper.

2.1. System Model

The system consists of a set of user equipments (UEs) that generate computational tasks, a pool of virtual machines (VMs) instantiated to execute these tasks, and a set of physical machines (PMs) with finite resource capacities, such as CPU cores and memory Gbs. Each task submitted by a UE must be offloaded and executed within a time duration T by a VM, which in turn must be allocated to a suitable PM. This process is illustrated in a simplified manner in Figure 1

Each VM consumes a certain amount of resources (e.g., CPU cycles), and each PM has a limited number of cores it can provide. A feasible VM-to-PM assignment must respect these capacity constraints while attempting to minimize energy consumption.

There is a total of L tasks, M physical machines, and N virtual machines. Each task (or UE) is defined by its resource requirements, specifically the number of CPU cores and the amount of memory (in GB). Each virtual machine provides a certain amount of these resources and is also characterized by its energy consumption per processor cycle, denoted in [3] as ξ_i where v_i is the VM instance. Physical machines are described by the total resources they offer (cores and memory) and the computing capability of a single core, measured in processor cycles per second and denoted as θ_j. These constraints and characteristics define the feasible mappings between VMs and PMs, which are central to the allocation problem. Since this project focuses on energy-efficient VM-to-PM allocation using matching algorithms, the initial assignment of UEs to VMs is assumed to be predefined and fixed.

In addition, each PM p_j can host at most δ_j VM instances, and each VM instance can be distributed across at most d different PMs. The placement of VMs onto PMs is represented by a binary matrix $B \in \{0,1\}^{N \times M}$, where each element $b_{i,j}$ indicates whether VM v_i is placed on PM p_j ($b_{i,j} = 1$) or not ($b_{i,j} = 0$).

2.2. Energy Consumption Model

Based on the system model previously described, the authors in [3] define an energy consumption model for the virtual resource allocation problem. They distinguish three stages in the task execution pipeline, each associated with specific energy consumption formulations:

- **Local computing**: part of the task may be executed locally by the UE. The energy consumed during this stage is denoted as E^{local}.
- **Computation offloading**: involves transferring the task to the MEC system and includes two phases: the offloading phase (with duration τ, where $0 < \tau < T$) and the downloading phase. The total energy consumed for offloading is denoted as E^{off}.
- **Remote computing**: the remaining portion of the task is processed by VM instances hosted on the PMs of the MEC system. The energy consumed during this stage is represented as E^{vm}.

The complete formulation of the energy consumption models is given in Equation 1.

$$E^{total} = \sum_{k=1}^{L} (E_k^{local} + E_k^{off}) + \sum_{i=1}^{N} E_i^{vm} \tag{1}$$

The authors then state that the total energy consumption of the MEC system actually depends only on the energy consumption for computing at the MEC data center. As a result, they simplify the objective by focusing solely on minimizing the energy consumption of the computing stage, represented by E^{vm}. This component is expanded in Equation 2.

$$E^{vm} = \sum_{i=1}^{N} \sum_{j=1}^{M} (T - \tau) \theta b_{i,j} \alpha_{i,j} \xi_i \tag{2}$$

In this context, θ denotes the computing capability of a single PM core, $b_{i,j}$ indicates whether VM instance vm_i is assigned to PM pm_j, $\alpha_{i,j}$ specifies the number of processor cores from vm_i allocated on pm_j, and ξ_i represents the energy consumption per processor cycle of vm_i.

2.3. Objective

Given the system and energy consumption models, the objective is to minimize the total energy consumption by determining an optimal VM-to-PM placement. This must be achieved while satisfying resource constraints and ensuring that tasks are properly allocated. In this project, instead of solving the problem via hypergraph matching as in the original paper, different matching algorithms are applied and analyzed with respect to their effectiveness in achieving low energy consumption through efficient resource utilization.

3. Matching algorithms

This section describes the algorithms implemented to address the resource allocation problem, , outlining their core mechanisms and operational logic. The methods include: *Random*, *Greedy (First-Fit)*, *Round-Robin*, *Gale-Shapley*, and an *Auction-based* algorithm.

3.1. Random

The *Random* algorithm serves as a simple baseline for comparison. The list of VMs is first shuffled to avoid bias. Then, for each VM, a random PM is selected from the subset of PMs that have sufficient resources to host it. If no such PM exists, the assignment process is terminated. This approach avoids repeated sampling by filtering out invalid candidates beforehand. The method's pseudocode is shown in Algorithm 1.

Algorithm 1 Random (Implemented)

Input: VMs, PMs

$VMs_{shuffled} \leftarrow$ shuffle(VMs)

foreach $vm \in VMs_{shuffled}$ **do**
  $ValidPMs \leftarrow \{pm \in PMs \mid pm$ has enough resources for $vm\}$
  if $ValidPMs = \emptyset$ **then**
    mark vm as unassigned
    continue
  end
  $pm_{chosen} \leftarrow$ sample randomly from $ValidPMs$
  assign vm to pm_{chosen}
  update resources of pm_{chosen}
end

Output: VM-to-PM assignments

3.2. Greedy (First-Fit)

The *First-Fit* (FF) algorithm follows a greedy strategy for resource allocation. For each task (and its associated VM), it sequentially checks the list of PMs and assigns the VM to the first PM with enough available resources. If a PM lacks sufficient resources, it is skipped and the next one in order is evaluated. The PMs are always considered in a fixed order. If none of the PMs can accommodate the VM, the task is left unallocated. For every new task, the PM list is re-evaluated from the beginning. The method's pseudocode is shown in Algorithm 2.

Algorithm 2 Greedy (First-Fit)

Input: VMs, PMs

foreach $vm \in VMs$ **do**
  $assigned \leftarrow$ False
  foreach $pm \in PMs$ **do**
    if pm *has enough resources for* vm **then**
      assign vm to pm
      update resources of pm
      $assigned \leftarrow$ True
      break
    end
  end
  if $\neg assigned$ **then**
    mark vm as unassigned
  end
end

Output: VM-to-PM assignments

3.3. Round-Robin

The third algorithm is a variation of the *Greedy (FF)* approach, with a key difference in how PMs are selected. Instead of always starting from the first PM, the *Round-Robin* method cycles through the PM list. After assigning a task to the i-th PM, the algorithm starts the next allocation from the $(i+1)$-th PM, continuing in a circular manner. The method's pseudocode is shown in Algorithm 3.

Algorithm 3 Round-Robin

Input: VMs, PMs
Initialize: $current\_pm \leftarrow 0$

foreach $vm \in VMs$ **do**
  $assigned \leftarrow$ False
  for $j = 0$ **to** $|PMs| - 1$ **do**
    $pm \leftarrow PMs[(current\_pm + j) \mod |PMs|]$
    if pm *has enough resources for* vm **then**
      assign vm to pm
      update pm resources
      $current\_pm \leftarrow (pm\ index + 1) \mod |PMs|$
      $assigned \leftarrow$ True
      break
    end
  end
  if $\neg assigned$ **then**
    mark vm as unassigned
  end
end

Output: VM-to-PM assignments

3.4. Gale-Shapley

The fourth method is the *Gale-Shapley* matching algorithm, originally designed to solve the stable marriage problem. In this context, an adaptation of its *Hospitals / Residents* variant [1] is used, to match VMs (each associated with a task) to PMs based on dynamic preferences.

Each VM maintains a preference list of PMs, ranked according to criteria such as available resources and energy efficiency. Each PM ranks VMs in a similar manner. The matching process is iterative. In each round, unassigned VMs propose to their most preferred PM not yet rejected by. PMs then tentatively accept the most preferred proposals among those received and reject the rest. The rejected VMs update their preference list and propose again in the next round. This process continues until all VMs are either matched or no further proposals can be made. The algorithm ensures a stable matching when preferences are strict (i.e., no ties) and static, meaning no VM-PM pair would prefer each other over their current match.

In this implementation, preferences are recomputed dy-

namically based on current resource availability, ensuring that decisions reflect the system's state at each step.

The preferences of both VMs and PMs are computed based on two main factors: the energy consumption of a potential match (as defined in Equation 2) and the efficiency of resource utilization, specifically load balancing and resource fragmentation. These metrics are normalized to ensure comparability. Each component in the preference calculation is weighted by a corresponding coefficient, allowing fine-tuning of the algorithm's behavior. For instance, increasing the weight of the energy consumption term biases the algorithm toward more energy-efficient matchings. Preferences are only calculated between VMs and PMs that can feasibly be matched—i.e., a VM only considers PMs that can meet its resource requirements.

For each eligible PM, a VM computes its preference as the sum of the estimated energy consumption and a load-balancing term, which encourages assigning tasks to under-utilized (or empty) PMs to evenly distribute the load. The preference value of VM instance vm_i for PM pm_j is given in Equation 3.

$$\begin{aligned} preference(vm_i,pm_j) = -\lambda \cdot E^{vm}(vm_i, pm_j) \\ + \gamma \cdot availableResources(pm_j) \end{aligned} \tag{3}$$

The coefficient λ weighs the energy consumption component; its negative sign ensures that lower energy consumption increases the overall preference score. Conversely, the coefficient γ scales the term related to the available resources of PM pm_j. Since γ is positive, VMs are encouraged to prefer less utilized (or empty) PMs, promoting a balanced distribution of tasks across the physical machines.

On the other hand, each PM evaluates its preference for a VM based on two main factors: the energy consumption of the potential match and a fragmentation term, which encourages PMs to prefer VMs whose tasks make efficient use of their available resources. The preference of PM pm_j for VM vm_i is computed as shown in Equation 4.

$$\begin{aligned} preference(pm_j, vm_i) = -\lambda \cdot E^{vm}(vm_i, pm_j) \\ + \mu \cdot (usedResources(pm_j) + requiredResources(vm_i)) \end{aligned} \tag{4}$$

Similar to VM preferences, the energy consumption term contributes negatively to the overall preference score, meaning that PMs favor VMs that lead to lower energy consumption. The coefficient μ controls the weight of the consolidation term, which promotes efficient resource usage by reducing fragmentation. This term contributes positively to the preference value, encouraging PMs to prioritize VM tasks that make better use of their available resources and avoid underutilization across many PMs.

The method's pseudocode is shown in Algorithm 4.

Algorithm 4 Gale-Shapley (with Dynamic Preferences)

Input: VMs, PMs

while *there are unmatched VMs* **do**
  foreach *unmatched* $vm \in VMs$ **do**
    compute vm's preference list over feasible PMs
    $pm^* \leftarrow$ most preferred PM not yet proposed to vm proposes to pm^*
  end
  foreach $pm \in PMs$ **do**
    compute pm's preference list over proposing VMs
    accept the most preferred VMs within resource limits
    reject the others
  end
  update VM matches based on PM responses
end

Output: VM-to-PM assignments

3.5. Auction-based

The last method is the *Auction-based* matching algorithm, which approaches the VM-to-PM assignment problem using principles from auction principles. In this setting, VMs act as bidders, and PMs act as items being auctioned.

Each VM evaluates all PMs using a scoring function that combines multiple normalized metrics, including energy consumption and PM resource utilization. These metrics are weighted by configurable coefficients and summed to compute an overall score for each PM, as shown in Equation 5.

$$\begin{aligned} eval_{vm_i}(pm_j) = -\lambda \cdot E^{vm}(vm_i, pm_j) - \phi \cdot price(pm_j) \\ - \gamma \cdot loadFactor(pm_j) + \rho \cdot computeSpeed(pm_j) \end{aligned} \tag{5}$$

The evaluation function accounts for several factors. The PM's current load factor contributes negatively, encouraging VMs to prefer under-utilized or empty PMs and promoting load balancing—similarly to the Gale-Shapley approach. In contrast, the PM's computational performance (e.g., core operations per second) has a positive impact, making more powerful PMs more desirable. Finally, both the energy consumption of the VM-PM match and the PM's base price negatively affect the score, favoring allocations that are energy-efficient and cost-effective.

Each unassigned VM uses the evaluation function to assess all PMs to which it can be feasibly allocated. It then places a bid for the PM with the highest evaluation score. The bid value for a VM vm_i is calculated as shown in Equation 6.

$$bid_{vm_i} = best\_PM\_value - second\_best\_PM\_value + \epsilon \tag{6}$$

Each PM that receives bids accepts the highest one and rejects the rest. The PM's base price (initially set to zero) is then updated using an Exponentially Weighted Moving Average (EWMA) of the winning bid, balancing current market demand with historical pricing trends, as shown in Equation 7.

$$price(pm_j)' = \alpha \cdot bid^* + (1 - \alpha) \cdot price(pm_j) \quad (7)$$

The accepted VM-to-PM match is added to the current assignments, while rejected VMs re-enter the bidding process in the next round. This auction and rebidding process continues until no further updates occur — that is, when all VMs are either successfully matched or can no longer be allocated due to resource constraints.

This method encourages both load balancing and energy-efficient assignments through the use of dynamic pricing, as PMs that are in high demand become more "expensive" and thus less attractive to other VMs in subsequent rounds. The method's pseudocode is shown in Algorithm 5

Algorithm 5 Auction-based

Input: VMs, PMs

while *there are unmatched VMs* **do**
  foreach *unmatched* $vm \in VMs$ **do**
    compute vm's evaluation of all feasible PMs
    select $pm^* = \arg\max_{pm_j} eval_{vm_i}(pm_j)$
    compute and submit bid to pm^*
  end
  foreach pm_j *that received bids* **do**
    accept the highest bid bid^* from vm^*
    assign vm^* to pm_j
    update pm_j price
    reject all other bids
  end
end

Output: VM-to-PM assignments

4. Implementation

This section briefly describes the hierarchy structure of the implemented classes for the MEC system, the algorithms, their respective models and services. The complete set of UML-like diagrams are shown in Appendix 7.1. The publicly available source code [2] is written in Java.

4.1. MEC system

This section presents the classes that define the MEC system network. Other class diagrams are in Appendix 7.1.1.

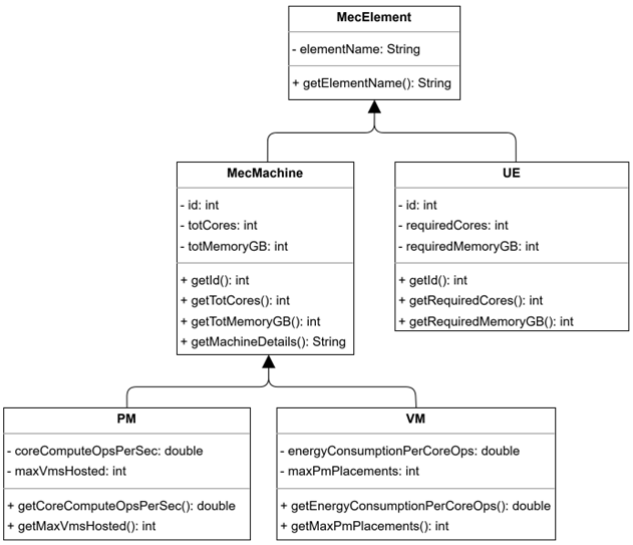

Figure 2. Class structure of the MEC elements

4.1.1 MecElements classes

The *MecElement* class describes the generic MEC network element. It is an abstract class which is then extended by all network elements, namely UEs, and by the *MechMachine*s (i.e., VMs and PMs).

The *MecElement* class is extended by the *UE* class and by the *MecMachine* abstract class. The *UE* class represents the user equipments, and is defined by the resource requirements of the corresponding task. The *MecMachine* class represents instead the generic elements of the MEC network, namely VMs and PMs, whose classes extend the *MecMachine* one. These elements are very similar and share many class fields, such as the available resources (in terms of cores and memory GBs) and the maximum number of matchings the element can be placed in. These are represented by the *VM* and *PM* classes.

The complete class diagram of the *MecElement* class and of its subclasses is shown in Figure 2.

4.1.2 MecSystem class

The *MecSystem* class describes the whole network system. It defines lists containing the UEs, the VMs, and the PMs, as well as system configuration values like the duration time and the offloading time (as defined in Section 2.2). Aside from *getter* methods, it also provides methods to add elements to the network.

The class's diagram is shown in Figure 11.

4.1.3 MecMapping class

The *MecMapping* class defines the mappings between VMs and PMs, in the form of the binary assignment matrix B

(as defined in Section 2.1), and more specific ones defining the resources (core and memory GBs) assignments in more detail. Aside from *getter* methods, it also provides methods to add or remove VM-to-PM assignments.

The class's diagram is shown in Figure 12.

4.2. Algorithms

This section presents the classes that define and implement the resource allocation algorithms, and their helper model classes. Other class diagrams are in Appendix 7.1.2.

4.2.1 MatchingAlgs classes

All algorithms extend the abstract *MatchingAlg* class, representing a generic resource allocation method. This contains fields representing the defined matches, as well as the services for handling the MEC system elements and for computing the energy consumptions, namely the *MecSystemService* and the *EnergyConsumptionService* (presented below in Section 4.3). It also provides methods for allocating the matches' required resources on PMs, for preparing the algorithm's results with the final allocation statistics and results, and the abstract method *run()* which gets implemented in all the classes extending it.

The *MatchingAlg* class is extended by all the implemented algorithms. Its first subclass is the *RandomAlg* class, representing the Random approach (see Section 3.1): this only contains the seed for performing the random list shuffles and sampling. The *GreedyAlg* class implements the Greedy (FF) method (see Section 3.2), and only provides the default methods for running the algorithm and returning its extended name. The *RoundRobinAlg* class implements the Round-Robin method (see Section 3.3); similar to the *GreedyAlg*, this class does not provide methods other than the default ones.

The *GaleShapleyAlg* class implements the Gale-Shapley method (see Section 3.4). The class fields represent the VM and PM preferences and the multiplying coefficients used in the preference computation step (see Equations 3 and 4). Other than the method for running the algorithm, it contains private ones for computing the preferences and performing the method's main loop.

The last method is the Auction-based algorithm (see Section 3.5), implemented with the *AuctionAlg* class. The class fields represent the VM evaluations of PMs, the energy consumption costs of the potential matches, the PMs' base prices, and the bids the VMs offer to PMs at each round. The other fields are the multiplying coefficients used in the PM evaluation and price updates steps (see Equations 5 and 7). Other than the method for running the algorithm, it contains private ones for computing the energy costs, evaluating the PMs and performing the method's main loop.

The complete class diagram of the *MatchingAlg* class and of its subclasses is shown in Figure 3.

4.2.2 Algorithms' helper classes

The algorithms make use of other helper classes that allow to keep track of (temporary) resource allocation on PMs, and to define the Gale-Shapley's preference and Auction-based bids, the (predefined and fixed) UE-to-VM mappings, and the final algorithm's results.

Among these classes, there is the *Ue2VmMapping* class, which defines the fixed matchings between UEs and VMs. It contains fields identifying the UE and the VM, and representing the task's resource requirements in terms of cores and memory GBs. The class's diagram is shown in Figure 13.

The *ResourceAvailability* class is used to keep track of the allocation of PM resources during the algorithms' execution. This provides fields and methods to query and update (allocate or release) the available resources, or to check if a potential match is allowed (e.g. for resource availability). The class's diagram is shown in Figure 14.

The *Preference* class is primarily used by the Gale-Shapley and Auction-based algorithms, and represents the proposals and offers of VMs to PMs, and vice-versa. It contains identifiers of the proposers and the receivers, and the preference the former have for (or bid they offer to) the latter. The class's diagram is shown in Figure 15.

The final helper class for the algorithms is the *AlgorithmResults* record class, containing fields representing the results of the algorithms' execution: these include the total number of allocated tasks, VMs, and PMs, and the energy consumption resulting from the found match. The class's diagram is shown in Figure 16.

4.3. Services

This section presents the service classes used to handle, define and update the VM-to-PM matches and the system's energy consumption. The class diagrams are in Appendix 7.1.3.

The *MecSystemService* class provides functionalities to handle and update the configuration of the MEC system and of its mappings. These include methods for getting the UE-to-VM and VM-to-PM mappings (in terms of presence or absence of matches, or in terms of allocated resources), add new matches or remove already defined ones, and to check if potential mappings are allowed. The class's diagram is shown in Figure 17.

The *EnergyConsumptionService* class provides functionalities to either compute the partial energy consumption resulting from a single VM-to-PM match or calculate the total system's energy consumption. The class's diagram is shown in Figure 18.

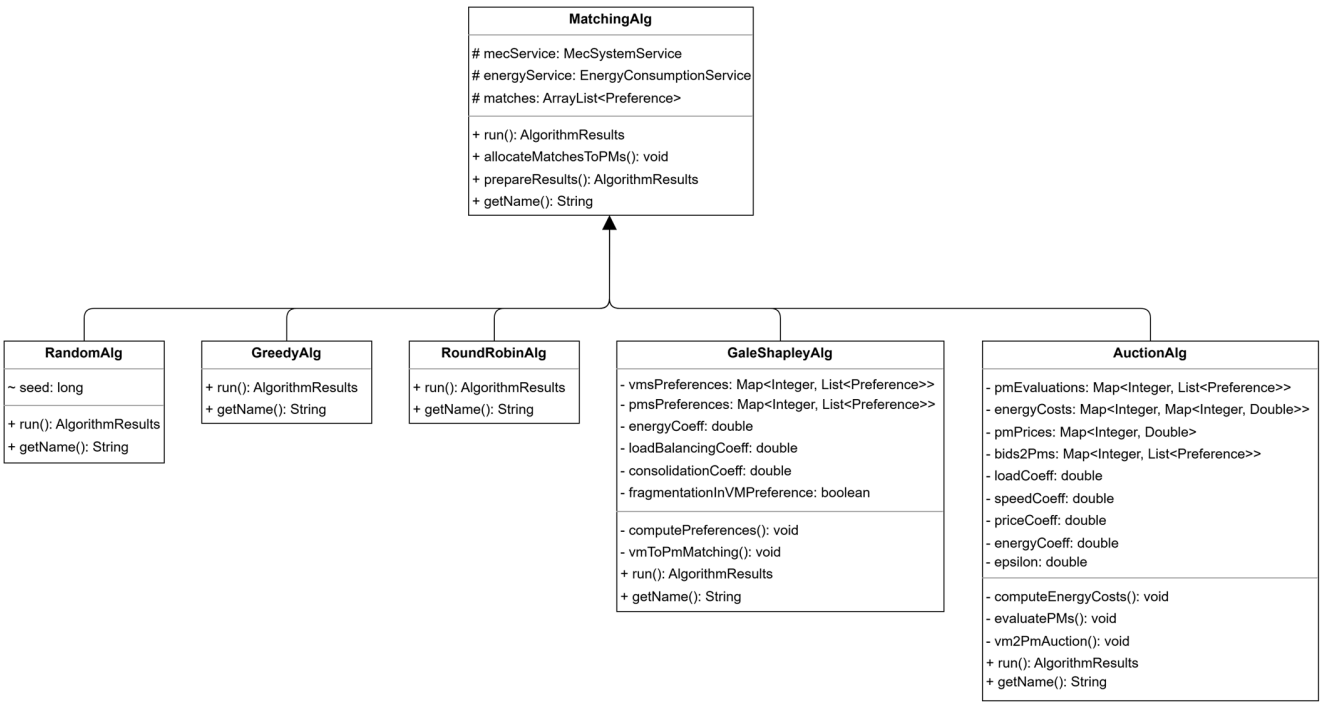

Figure 3. Class structure of the algorithms

5. Experimental Tests

This section describes the experimental setup and analyzes the results. The tests are divided into two distinct analyses. The first compares all implemented methods in terms of energy efficiency, execution time, and overall resource allocation. The second focuses solely on the Gale-Shapley algorithm, using different preference weight coefficients to study the effect of shifting the consolidation term from the receiver (PMs) to the proposer (VMs). Since the two analyses have different objectives and parameter settings, their results are not directly comparable. In particular, differences in Gale-Shapley's resource allocation outcomes may appear ambiguous due to the change in optimization goals and weight configurations.

5.1. Settings

This subsection presents the experiments settings in terms of problem instance configurations.

The number of physical machines (PMs), virtual machines (VMs), and user equipments (UEs) was fixed to 30 for all experiments. Their configurations were randomly generated to simulate a variety of realistic scenarios in terms of computational and memory demands. Each PM was initialized with a number of CPU cores randomly selected between 1 and 32. The memory capacity was set proportionally to the core count, computed by multiplying the number of cores with a random factor between 2 and 4 GB. Each VM was assigned between 1 and 16 CPU cores. Its memory capacity was set as the number of cores multiplied by a random factor between 1 and 4 GB. Each UE was configured with a number of required CPU cores between 1 and 8, and a memory requirement between 1 and 8 GB.

These randomized configurations were used to create diverse and balanced test instances, ensuring that the evaluation covered a broad set of allocation scenarios.

5.2. Results

This subsection discusses the test results presenting a set of different analyses. More results plots are shown in Appendix 7.2.

Figure 4 shows the average energy consumption as a function of the computing capabilities of the PMs (measured in cycles per second) for each implemented method, normalized by the corresponding number of allocated tasks. As expected, energy consumption increases linearly with higher PM performance, regardless of the algorithm used. This trend is consistent with Equation 2, where the power of the physical machine's cores directly influences total energy usage. Despite this shared trend, the algorithms differ in their overall energy efficiency. The Gale-Shapley method yields the lowest energy consumption, followed by the Auction-based and First-Fit (Greedy) approaches. The Round-Robin method performs slightly worse, while the Random algorithm results in the highest energy usage.

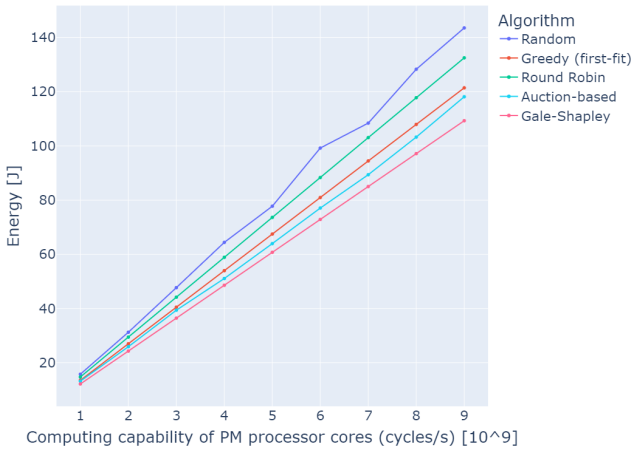

Figure 4. Energy consumption over PM computing capabilities.

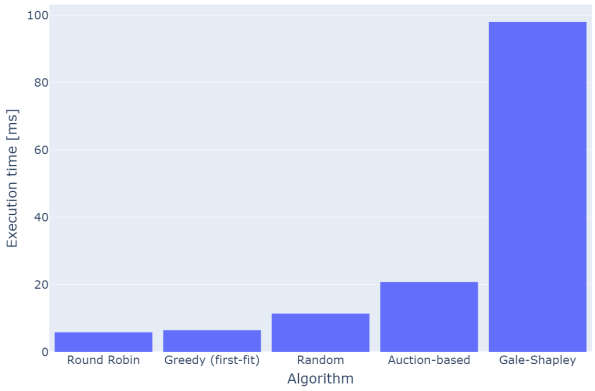

Figure 5. Execution time over algorithm.

Figure 5 reports the average execution time (in milliseconds) for each implemented algorithm. The fastest methods are the greedy ones—specifically, Round-Robin and First-Fit—which quickly assign each task to the first available PM that satisfies its requirements. The Random algorithm performs slightly worse due to the overhead of filtering valid PMs for each VM and the randomness of selection. The Auction-based method shows a noticeable increase in execution time due to the additional overhead of evaluating PMs and managing the bidding process. The Gale-Shapley algorithm is the slowest overall, with an average execution time approximately five times higher than the second-slowest method. This is primarily due to the added complexity of computing preferences for both VMs and PMs, along with the iterative proposal and rejection mechanism.

Figure 6 shows the average percentage of PMs allocated by each algorithm. As expected, the Greedy (FF) method achieves the lowest allocation rate, using the fewest physical machines. This is due to its sequential assignment logic, which always selects the first suitable PM in a fixed list. The Auction-based algorithm results in slightly higher PM

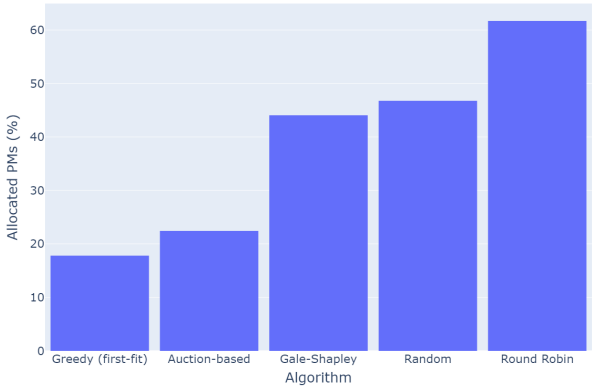

Figure 6. PM allocation percentage over algorithm.

usage, allocating just over 20% of the machines on average. This is a consequence of its evaluation function, which accounts for current resource usage and load balancing. The Gale-Shapley method leads to a more distributed allocation, with over 40% of PMs being used. The algorithm promotes consolidation via a dedicated term in the PM preference computation. Notably, this term is included by default in the PMs' preferences (a deeper analysis on its placement between VMs and PMs is presented later). The Random algorithm performs as expected, allocating nearly 50% of available PMs due to its completely random task placement. The Round-Robin method is the least efficient in terms of consolidation, resulting in the highest PM allocation rate. This behavior is consistent with its cyclic assignment logic, which can spread tasks evenly across all PMs.

Figure 7 provides a more detailed view of PM resource usage for each algorithm. Specifically, it shows the distribution of allocated resources across all PMs, where each data point represents a single PM and its percentage of allocated CPU cores (left subplot) or memory GBs (right subplot). Consistent with the trends observed in Figure 6, the Greedy (FF) and Auction-based algorithms show the highest level of resource consolidation. Most PMs in these methods remain unused, with the majority of data points clustered at 0%, indicating that over 80% and slightly less than 80% of PMs, respectively, are completely unallocated. The Gale-Shapley and Random algorithms yield more distributed allocations, with more than 50% of PMs remaining unused but showing greater variability in resource usage. The Round-Robin algorithm results in the most widespread use of PMs. This is reflected in its boxplot distribution, where the median allocation exceeds 10%, confirming its tendency to spread tasks evenly across all available machines.

Gale-Shapley's Consolidation Term Strategy In the Gale-Shapley method, both VMs and PMs compute preference rankings to determine their most desirable match. As

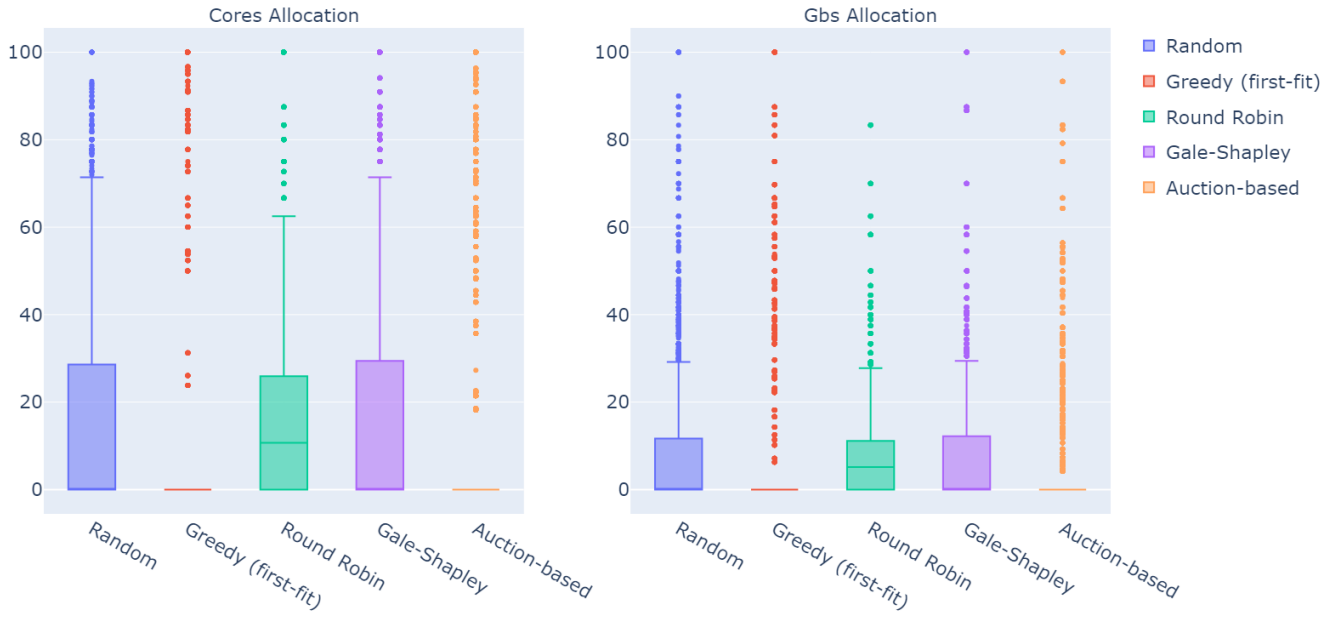

Figure 7. PMs' resource utilization (cores left, memory GBs right) over algorithms.

previously discussed, these preferences are based on different evaluation criteria depending on whether the agent is a VM or a PM. For instance, VMs typically consider load balancing in their evaluations, while PMs may prioritize other factors. One important factor is the *consolidation* term, which aims to reduce task fragmentation by promoting the grouping of tasks on fewer machines. By default, this consolidation term is included in the PMs' preference computation. However, in the Gale-Shapley algorithm, the proposer—typically the VM—has a strategic advantage during the matching process. Therefore, this section explores the impact of relocating the consolidation term from the PMs' to the VMs' preference computation. The goal is to assess how this shift influences the final matching outcome and the overall system efficiency.

Figure 8 starts by showing the difference in average of the allocated PMs percentage: when the consolidation is enforced by the proposer (i.e., the VMs) the used PMs decrease by almost 10%.

Figure 9 presents the distribution of PM resource utilization, with core usage shown in the top subplot and memory usage (in GBs) in the bottom one. The plots compare the effects of including the consolidation term in the proposer's or receiver's preference computation. When the consolidation term is assigned to the proposer (i.e., the VMs), the distribution of resource usage shifts noticeably. In particular, the distribution of allocated cores becomes more spread out and skews toward higher utilization values. This indicates that tasks are more densely packed onto fewer PMs, resulting in more efficient consolidation. A similar trend

Figure 8. PM allocation percentage with the proposers/VMs (left-/blue) or receiver/PMs (right/red) as the consolidation enforcer.

can be observed for memory usage, though the effect is less pronounced.

Figure 10 provides a more detailed visualization of core usage distributions across PMs, highlighting the impact of who enforces the consolidation objective. The top subplot (in blue) displays results when consolidation is enforced by the proposer, while the bottom subplot (in red) shows the case where the receiver enforces it. The results clearly indicate that when the proposer (i.e., the VM) incorporates consolidation into its preference computation, the outcome leads to a higher number of unused (empty) PMs, fewer PMs in the $5\% - 70\%$ utilization range, and a noticeable increase in highly (or fully) utilized PMs—nearly 30% of

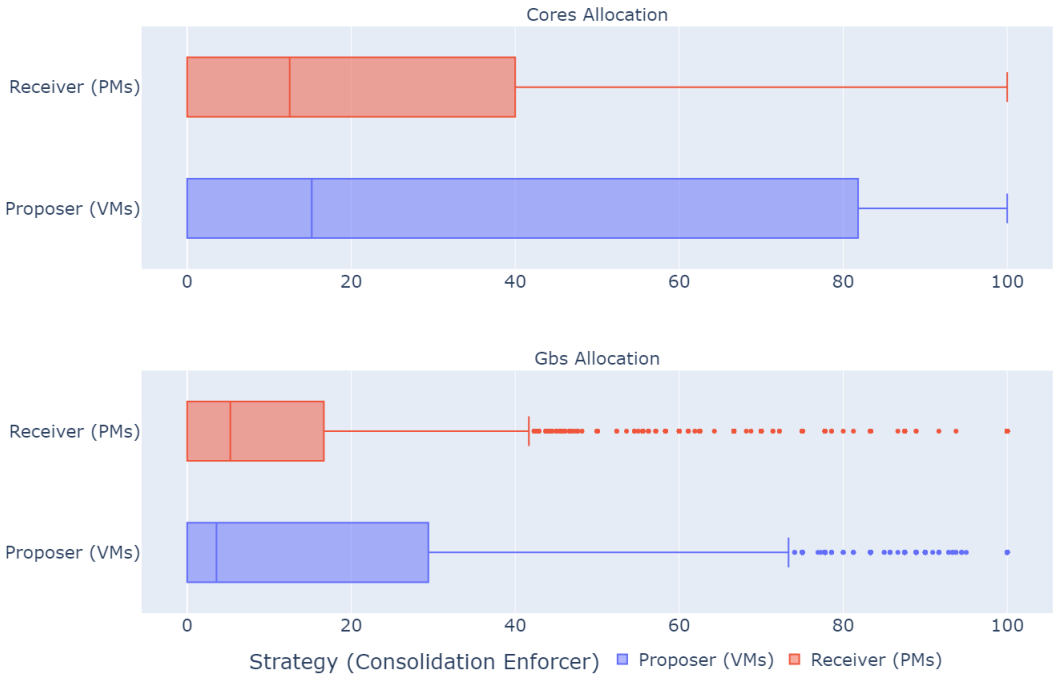

Figure 9. PMs' resource utilization percentage (cores in subplot on top, memory GBs on bottom one) with different consolidation enforcers: proposers/VMs in blue and receiver/PMs in red.

the total PMs have 80% of core allocation, or higher. In contrast, when consolidation is handled by the receivers (i.e., the PMs), the distribution shifts. There are fewer completely idle or highly utilized PMs, and a significantly higher concentration of machines operating in the $5\%-50\%$ allocation range.

6. Conclusions

This work presented and compared several VM-to-PM matching algorithms for efficient resource allocation in MEC systems, evaluating their performance in terms of execution time, energy efficiency, and resource utilization. The results show that simple greedy methods offer fast execution and good consolidation, while more sophisticated approaches like the Auction-based and Gale-Shapley algorithms enable better energy efficient matchings and adaptability through dynamic preferences and pricing. Notably, preference design—particularly the placement of consolidation objectives—significantly affects the final allocations. Future work may explore hybrid strategies and more realistic dynamic workloads to further improve system performance.

References

[1] D. F. Manlove. *Hospitals/Residents Problem*, pages 390–394. Springer US, Boston, MA, 2008.

[2] G. Piqué. Pikerozzo / matching-service-placement. https://github.com/pikerozzo/matching-service-placement. (19.05.2025).

[3] L. Zhang, H. Zhang, L. Yu, H. Xu, L. Song, and Z. Han. Virtual resource allocation for mobile edge computing: A hypergraph matching approach. In *2019 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6, 2019.

7. Appendix

7.1. Class diagrams

The following subsection presents the extended set of UML-like diagrams of the implemented classes.

7.1.1 MEC system

The MEC system class structures are shown in Figures 11,12.

7.1.2 Algorithms' helper classes

The algorithms class structures are shown in Figures 13,14,15,16.

7.1.3 Services

The services class structures are shown in Figures 17,18.

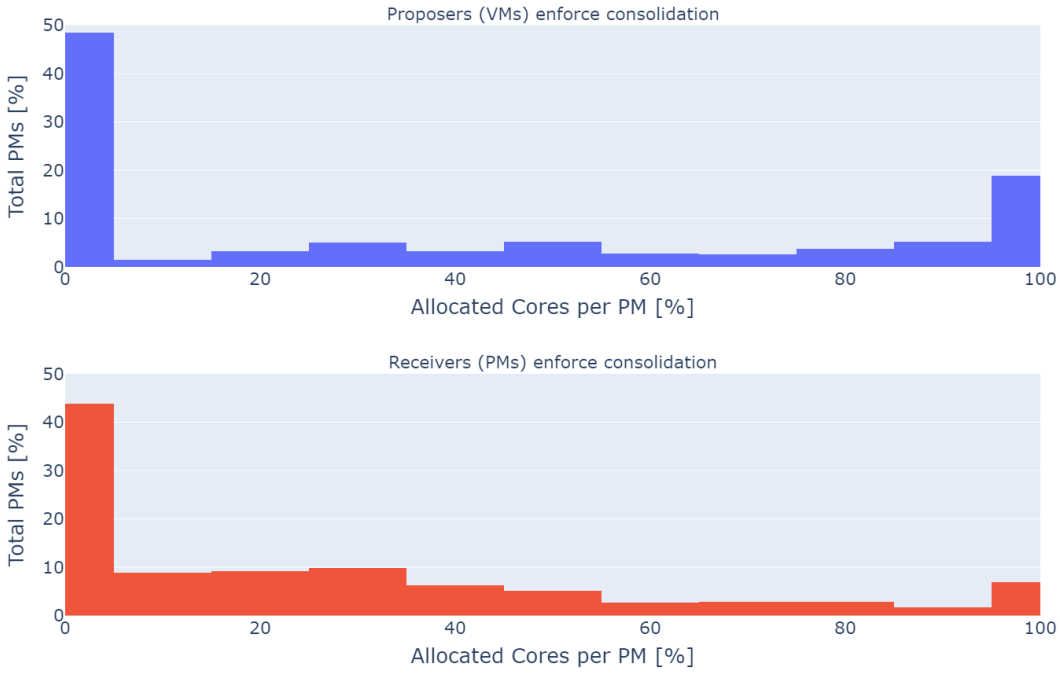

Figure 10. PMs' cores utilization percentage with different consolidation enforcers: proposers/VMs in blue on top, and receiver/PMs in red on bottom.

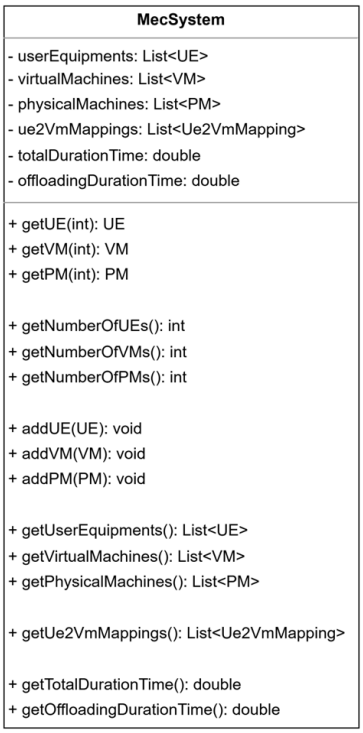

Figure 11. Class structure of the MEC system

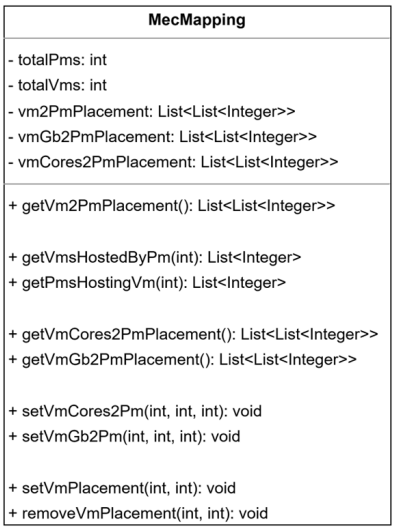

Figure 12. Class structure of the MEC mappings

7.2. Additional Results

This appendix section presents additional tests results. In particular, it displays further details on the PMs' core allocation distributions for each method (Figures 19,20,21,22,23).

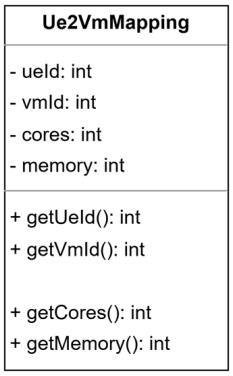

Figure 13. Class structure of the fixed UE-to-VM mappings

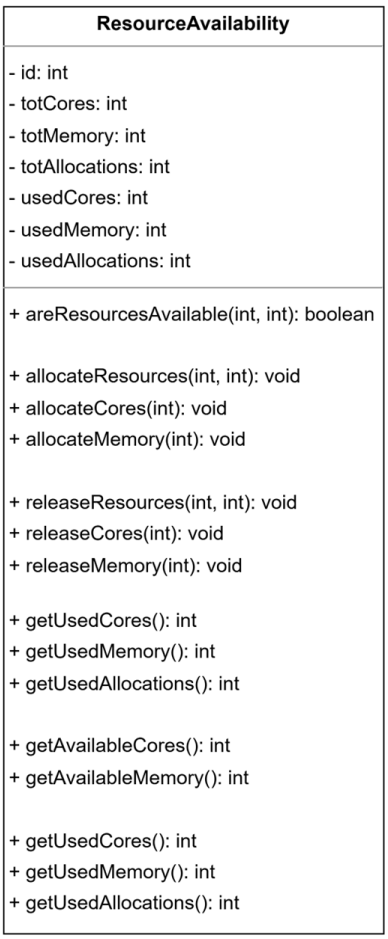

Figure 14. Structure of the resource availability class

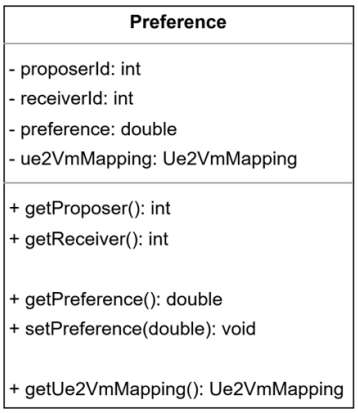

Figure 15. Structure of the class representing the preferences and bids

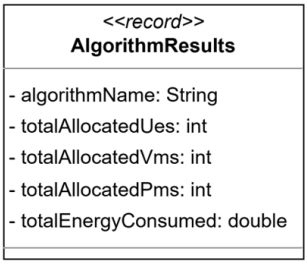

Figure 16. Structure of the class representing the algorithms' results

MecSystemService

+ addUe2VmMapping(Ue2VmMapping): void
+ getUe2VmMappings(): List<Ue2VmMapping>

+ checkEnoughPmResources(int, int, int, int): boolean
+ checkAssignmentAllowed(int, int): boolean

+ getHostedVmsCoresByPm(int): List<Integer>
+ getHostedVmsGbsByPm(int): List<Integer>

+ getRemainingCoresInVm(int): int
+ getRemainingGbsInVm(int): int

+ getRemainingCoresInPm(int): int
+ getRemainingGbsInPm(int): int

+ getVm2PmPlacement(): List<List<Integer>>
+ getVmCores2PmPlacement(): List<List<Integer>>
+ getVmGb2PmPlacement(): List<List<Integer>>

+ getVmCores2Pm(int, int): int
+ getVmGb2Pm(int, int): int

+ setVmCores2Pm(int, int, int): void
+ setVmGb2Pm(int, int, int): void

+ addVMResourcesOnPm(int, int, int, int): void
+ removeVMResourcesOnPm(int, int, int, int): void

+ setVmPlacementOnPm(int, int): void
+ removeVmPlacementOnPm(int, int): void

+ getTotalAllocatedVms(): int
+ getTotalAllocatedPms(): int

+ getVmsHostedByPm(int): List<Integer>
+ getPmsHostingVm(int): List<Integer>

Figure 17. Class structure of the MEC system service class

EnergyConsumptionService

+ getEnergyConsumptionWithVmAndPm(VM, PM): double

+ getEnergyConsumptionPerVm(int): double
+ getEnergyConsumptionPerPm(int): double

+ getTotalEnergyConsumption(): double

Figure 18. Structure of the energy consumption service class

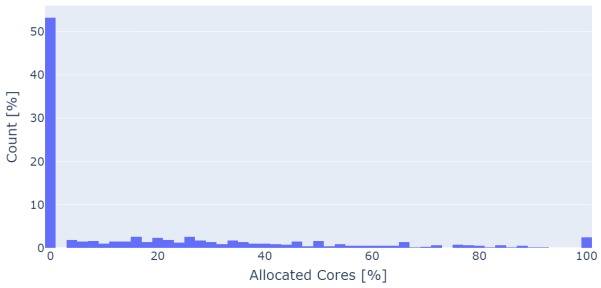

Figure 19. PM core allocation distribution with the Random algorithm

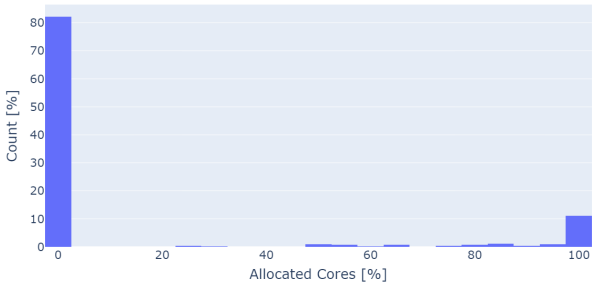

Figure 20. PM core allocation distribution with the Greedy (FF) algorithm

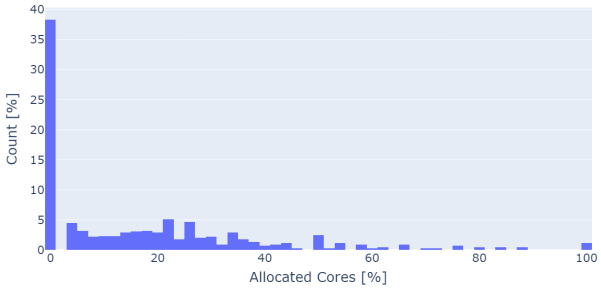

Figure 21. PM core allocation distribution with the Round Robin algorithm

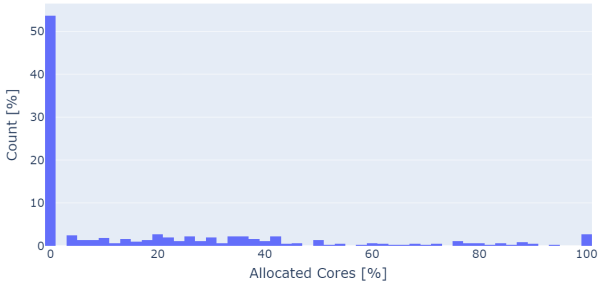

Figure 22. PM core allocation distribution with the Gale-Shapley algorithm

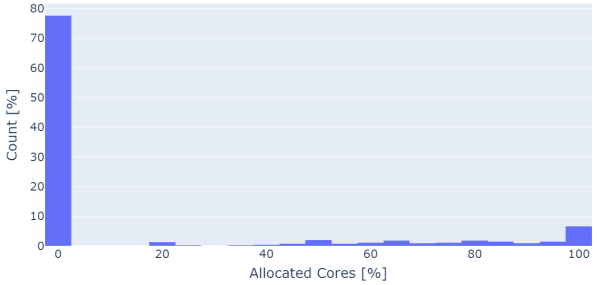

Figure 23. PM core allocation distribution with the Auction-based algorithm