Journal of Theoretical and Applied Information Technology

30th September 2023. Vol.101. No 18

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

REINFORCEMENT LEARNING BASED LOAD BALANCING FOR FOG-CLOUD COMPUTING SYSTEMS: AN OPTIMIZATION APPROACH

MUSTAFA AL-HASHIMI[1], AMIR RIZAAN RAHIMAN[2], ABDULLAH MUHAMMED[3], NOR ASILAH WATI HAMID[4]

[1,2,3,4] Department of Communication Technology and Networks, Universiti Putra Malaysia (UPM), Serdang 43400, Malaysia

E-mail: [1]gs59883@student.upm.edu.my, [2]amir_r@upm.edu.my, [3]abdullah@upm.edu.my, [4]asila@upm.edu.my

ABSTRACT

Fog-cloud computing is a promising approach to enhance distributed systems' efficiency and performance. Though, managing resources and balancing workloads in such environments remains challenging due to their inherent complexity and dynamic nature. The need for effective load-balancing techniques in fog-cloud computing systems is crucial to optimize resource allocation, minimize delays, and maximize throughput. This article presents a reinforcement learning (RL)-based load balancing system for fog-cloud computing, employing two RL agents: one for allocating new tasks to fog or cloud nodes and another for migrating tasks between fog nodes to ensure fair distribution and increased throughput. This study derived up with novel state, action, and reward models for both agents, facilitating collaboration during the load-balancing process. Three types of rewards for the RL agents are explored: single objective, multi-objective under non-dominated sorting, and multi-objective under lexicographical sorting. The performance of these methods is assessed using metrics such as average utilization, number of tasks completed, serve rate, and delay. The experimental results showed that RL-based scheduling methods, particularly the Reinforce Learning Multiple Objective (RLRLM) with RL-based migration method outperforms greedy on CPU (GR_c) and greedy on reliability (GR_r) methods across all performance metrics. The choice of migration method and reward type also influences performance. These finding highlight RL's potential in optimizing fog-cloud computing and offer valuable insights for future research and practical applications in this field.

Keywords: *Fog-Cloud Computing, Load Balancing, Reinforcement Learning, Resource Allocation, Multi-Objective Optimization.*

1. INTRODUCTION

Fog computing is a modern method that expands cloud computing by placing computing resources closer to the network's edge [1]. It has become crucial with the growth of the Internet of Things (IoT) and the need for rapid data processing. Moreover, the fog computing efficiently manages the massive data generated by IoT devices [2]. By processing, storing, and analyzing data closer to the source, it reduces delays and bandwidth requirements on cloud computing. Furthermore, it improves reliability and security by utilizing resources from edge devices like routers and gateways. Fog computing is used in various application, such as healthcare, transportation, and smart cities. For example, in healthcare, it allows real-time patient monitoring, leading to early diagnosis and treatment [3 - 5]. In transportation, fog computing plays a vital role in advancing autonomous vehicles by enabling rapid data processing and analysis. In smart cities, it helps manage traffic, conserve energy, and improve public safety.

Managing fog networks and distributing tasks between fog and cloud is crucial for achieving optimal performance in fog computing [6]. This involves deploying resources and assigning tasks based on factors like workload, network latency, and resource availability [7]. A major challenge in managing the fog is load balancing, ensuring an even distribution of computational work across the network to prevent nodes from becoming overloaded [8]. To address this challenge, fog computing management systems use techniques like task

migration, resource allocation, and workload balancing algorithms [9]. These techniques enable fog nodes to collaborate, distributing tasks based on their availability, processing power, and network connectivity. The dynamic nature of fog networks, their ability to adapt to changes in workload and network conditions, makes them appealing for various applications [10]. However, managing fog computing becomes challenging due to this dynamic nature, which demands a dynamic-aware approach for effective network operation. Factors such as device mobility and changing network conditions influence the fog network's dynamic nature. For instance, in a smart city, the number and location of connected devices may change over time, leading to fluctuations in workload and available resources.

This article specifically concentrates on the dynamic nature of fog computing, particularly concerning fog network management and load balancing. The main objective is to explore and assess how reinforcement learning (RL) can be utilized as a possible solution to tackle these challenges. To enhance network performance in fog computing, it's essential to have dynamic-aware management systems that can adapt to changing network conditions like latency and bandwidth, influenced by the number and location of connected devices. These systems utilize techniques such as adaptive resource allocation, dynamic workload balancing, and network-aware task migration. Adaptive resource allocation adjusts the resources assigned to different fog nodes based on their workload and availability while dynamic workload balancing redistributes tasks among fog nodes, considering their current workload and network conditions.

Reinforcement Learning (RL) is an artificial intelligence approach that holds promise for fog computing management. It can optimize task and resource allocation, as well as load balancing, by learning from interactions with the environment [11]. RL's strength lies in its ability to adapt to the dynamic and unpredictable fog computing environment, where network conditions and workload change over time. This enables fog nodes to learn from experiences and enhance network performance and resource usage accordingly. This article focuses on using RL to address the dynamic challenges of fog computing, particularly in fog cloud management and load balancing.

The rest of the article is organized as follows. Section 2 presents the contributions of the study. Next, the study background is provided in Section 3. Then, Section 4 explains the used methodology. The experimental work and results are covered in Section 5. Section 6 discusses the finding and Section 7 concludes the study and outlines the future work.

2. CONTRIBUTIONS

This article makes significant contributions to the field of fog-cloud computing and load balancing.

i. Introducing a novel load-balancing system that utilizes RL agents (allocation and migration) to optimize resource allocation in fog-cloud computing environments.

ii. Proposing innovative state, action, and reward models for both RL agents, facilitating collaboration and coordination during the load-balancing process.

iii. Exploring three different reward models for the RL agents: single objective, multi-objective under non-dominated sorting, and multi-objective under lexicographical sorting.

iv. Comparing the proposed load balancing system with state-of-the-art approaches and validating the results using standard evaluation metrics and statistical analysis.

3. BACKGROUND

In recent years, RL approaches have gained popularity in scheduling and resource allocation for distributed systems. Researchers have introduced several RL-based algorithms to optimize resource usage, enhance system performance, and reduce energy consumption in fog computing and other distributed systems. These algorithms are often combined with other meta-heuristic searching methods.

For example, in [12], a combination of Mayfly Taylor Optimization and Deep-Q-Network (DQN) was employed to optimize a fitness function that considers energy consumption, service level agreement verification, and cost. However, a notable issue with such approaches is that the meta-heuristic algorithms often demand a significant amount of search time to discover the best solution. The study conducted by [13] used a RL approach with random choice to manage tasks within deadlines. However,

a limitation of their approach is the use of random selection, which may not produce optimal results. Additionally, achieving optimal performance in various environments with their approach requires fine-tuning of hyperparameters and reward functions, which can be time-consuming and challenging.

In [14], a fog layer was incorporated to achieve task execution balance. This layer consists of two distinct modules: the RL Allocation algorithm (RLA) and RL Migration algorithm (RMA). These modules employ the Q-learning approach and determine rewards based on specific metrics, including process size, RAM, CPU usage, and completion rate, collectively referred to as "process weights." However, using these discrete key modules may result in a loss of information when selecting the optimal task for execution and migration.

In [15], the authors used Q-learning with a discrete state representation, considering factors like the task's load level, sibling tasks, and parent tasks. However, this approach faced criticism for using an incomplete state representation, violating the Markov Decision Process (MDP) assumption. Additionally, [16] suggested a deep RL-based approach for resource provisioning in fog computing, while [17] proposed an energy-efficient task scheduling method using deep RL. Both approaches have limitations related to quantization of node index, leading to information loss, and longer waiting times for task scheduling, due to task prioritization. Additionally, [18] suggested an RL-based approach for scheduling live migration from underutilized hosts, but it lacked the decentralization of wait and migrate states, impacting task scalability.

In [19], a task scheduling method for load-balanced fog computing using Q-learning was introduced. However, the formulation had limitations as it oversimplified available resources, security level, and power processing for each node, which could lead to security and latency concerns. Another work, [20], proposed an RL-based load-balancing algorithm for fog networks. Its goal was to maximize utility while minimizing processing delay and overload probability. However, it had a bias issue because it depended on the exploration policy, affecting the number of tasks offloaded to adjacent fog nodes. The authors of [21] presented a deep RL approach for scheduling IoT applications in a fog computing environment. The approach aims to optimize task scheduling and resource allocation to improve overall fog computing performance while reducing time and cost.

Overall, RL-based scheduling and resource allocation algorithms have demonstrated encouraging outcomes in enhancing the performance and efficiency of distributed systems, especially in fog computing environments. Nevertheless, certain challenges persist, including the requirement for ample training data, fine-tuning of hyperparameters and reward functions, and ensuring scalability and adaptability of the proposed methods across diverse scenarios and applications.

4. METHODOLOGY

In this study, the research design focuses on developing the following algorithms concerning scheduling and migration. By providing suitable and precise inputs for each algorithm, the study aims to generate reliable outputs that can be utilized in subsequent algorithms, thereby ensuring the validity and reliability of the results.

4.1. Scheduling

The primary scheduling task as outlined in Algorithm 1. The algorithm takes various inputs and generates essential scheduling outputs. The schedule begins by retrieving the number of tasks from the scheduler's task queue. Depending on the scheduling method, it either uses the ***RandomScheduler()*** method or the ***RLScheduler()*** method with specific parameters (e.g., reward type, scheduling tasks, sorting method, time counter, and servers). Next, the algorithm invokes the ***schedule()*** function of the scheduler's algorithm, using the scheduling tasks and time counter as inputs and produce a list containing selected servers along with the tasks assigned to them as output.

Algorithm 1 Scheduling task

Input:
(1) schedulingMethod
(2) migrationMethod
(3) rewardType
(4) nTask: tasks number to be scheduled
(5) sortingMethod: in the case of multi-objective-reward

Output:
(1) selectedServers: the selected servers and their newly assigned tasks.
(2) serversToMigrateFrom: the selected servers to migrate from.

Start Algorithm
1: schedulingTasks = get nTask task from self.tasksQueue
2: if schedulingMethod == 'Random' then

```
3:    self.algorithm = RandomScheduler()
4:  else
5:    self.algorithm = RLScheduler(rewardType,
      sortingMethod, timeCounter, servers)
6:  end if
7:  selectedServers =
    self.algorithm.schedule(schedulingTasks)
8:  serversToMigrateFrom =
    self.Migration(migrationMethod)
End Algorithm
```

Next, the algorithm calls the ***Migration()*** method of the scheduler, using the migration method, time counter, and servers as inputs. This method generates a list of servers to migrate from. Finally, the algorithm produces the output, which includes the list of selected servers with their newly assigned tasks and the list of servers to be migrated from.

4.2. Migrator

Algorithm 2 describes the migration task, which takes several inputs and produces information related to server migration. It begins by sorting the task queue to give priority to delayed tasks. Depending on the migration method specified, the algorithm uses either the ***RandomMigrator()*** method or the ***RLMigrator()*** method as the migrator method of the scheduler.

Algorithm 2 Migration task

```
Input:
(1) migrationMethod
(2) timeCounter
(3) servers

Output:
(1) serversToMigrateFrom: the selected servers and their
    newly assigned tasks

Start Algorithm
1:  self.tasksBuffeer.sort('momentGeneration')
2:  if migrationMethod == 'Random' then
3:    self.migrator = RandomMigrator()
4:  else if migrationMethod == 'RL' then
5:    self.migrator = RLMigrator()
6:  end if
7:  serversToMigrateFrom =
      self.migrator.migrate(timeCounter, servers,
      delayPeriod=0, self.bandwidth)
End Algorithm
```

Next, the algorithm invokes the ***migrate()*** function of the migrator, passing the time counter, servers, delay period (set to 0 for no delay), and scheduler bandwidth as inputs. The function produces a list that includes selected servers to migrate from and their newly assigned tasks. Finally, the algorithm produces the output, consisting of the list containing selected servers to migrate from along with their newly assigned tasks.

4.3. RL Scheduler

Algorithm 3 presents the RL scheduler algorithm, which aims to optimize the scheduling process. The scheduler takes multiple inputs and generates servers for assigning new tasks, as oulined in the algorithm. The algorithm begins by obtaining the current state of the RL environment using the RLScheduler's ***getState()*** function. If the current time counter exists in the list of Q-table updates, the algorithm updates the Q-table using the RLScheduler's ***updateQtable()*** function, incorporating inputs such as the current server status, time counter, reward type, and sorting method. The resulting reward is then added to the overall rewards of the RLScheduler.

Algorithm 3 RL Scheduler task

```
Input:
(1) rewardType
(2) schedulingTasks : tasks to be scheduled
(3) timeCounter
(4) servers: snapshot of the servers' status.

Output:
(1) selectedServers: the selected servers and their newly
assigned tasks

Start Algorithm
1:  currentState = self.getState()
2:  if timeCounter in self.QtableUpdates then
3:    reward = self.updateQtable(servers, timeCounter,
        rewardType)
4:    self.overAllRewards.add(reward)
5:  end if
6:  for task in schedulingTasks do
7:    capableServers = any server that can still receive
        tasks
8:    if randomNumber < self.epsilon then
9:      action = Random(capableServers)
10:   else
11:     action =

argmax(self.Qtable[currentState][capableServers])
12:   end if
13:   update servers' status after the action
14:   self.QtableUpdates[timeCounter +
        self.delayPeriod].add(currentState, action)
15:   selectedServers.add(task, action)
17: end for
18: self.epsilon = max(self.epsilon * self.decay,
      self.minEpsilon)
End Algorithm
```

For each task in the scheduling tasks, the algorithm determines the servers capable of handling the task. If a randomly generated number is less than the RLScheduler's epsilon value, the algorithm chooses a server randomly from the eligible ones.

Otherwise, it selects the server with the highest Q-value for the current state and eligible servers. The algorithm updates the server status by executing the chosen action and includes the current state and action in the list of Q-table updates for the next time step.

After that, the algorithm adds the chosen server and task to the list of selected servers with their newly assigned tasks. Then, the algorithm updates the RLScheduler's epsilon value by picking the larger value between the product of the current epsilon value and the decay factor and the minimum epsilon value. This updated epsilon value is then used in the server selection process for the next scheduling iteration.

Algorithm 4 UpdateQtable task

```
Input:
(1) servers
(2) timeCounter
(3) rewardType

Start Algorithm
1:  nextState = self.getState()
2:  reward = self.Reward(rewardType, servers)
3:  for state,action in self.QtableUpdates[timeCounter] do
4:     if rewardType == 'Multi-Objective' then
5:        for objective in self.numberObjectives) do
6:           TD = reward[objective] + self.gamma *
                  max(self.QtableMO[nextState][objective]) –
                  self.QtableMO[state, action][objective]
7:           self.QtableMO[state, action][objective] +=
                  (self.learningRate * TD)
8:        end for
9:     else
10:       TD = reward + self.gamma *
               max(self.QtableMO[nextState])-
               self.QtableMO[state, action]
11:       self.Qtable[state, action] += (self.learningRate *
TD)
12:   end if
13: end for
14: if rewardType == 'Multi-Objective' then
15:    self.sortQtable(state, self.sortType)
16: end if
End Algorithm
```

Algorithm 4 handles the Q-table updating process. The algorithm modifies the Q-table based on the current state, selected action, reward type, and sorting method (if multi-objective reward is involved). The algorithm starts by retrieving the next state of the environment from the RL scheduler instance. It then calculates the reward for the current state and adds it to the Q-table, using either the single-objective or multi-objective approach.

In the multi-objective case, it computes the temporal difference (TD) for each objective by considering the maximum Q-value for the next state and the learning rate. It then updates the Q-value for the current state-action pair for each objective. For the single-objective case, it also computes the TD using the maximum Q-value for the next state and the learning rate and updates the Q-value for the current state-action pair. After updating the Q-table, the algorithm checks if the reward type is multi-objective. If it is, the Q-table is sorted using the specified sorting method.

4.4. Reward Update

Algorithm 5 describes the reward update process. First, the algorithm checks the ***rewardType*** to determine the specific reward calculation needed. If the type is migration-reward, it calculates the difference between the old-utilization and new-utilization for servers that were overloaded and required migration to balance the load. On the contrary, if multi-objective reward, the algorithm computes a list of multiple reward values which include the sigmoid function of the server standard utilization deviation and queue occupation, as well as the server utilization median and queue occupation. If the ***rewardType*** is neither "migration-reward" nor "multi-objective reward," the algorithm calculates the sigmoid function of the server's utilization standard deviation.

Algorithm 5 Rewarding task

```
Input:
(1) rewardType
(2) servers

Start Algorithm
1:  if rewardType == 'migration-reward' then
2:     reward = old-utilization - new-utilization for all the
          servers that were overloaded and we perform
          migration to.
3:  else if rewardType == 'Multi-objective' then
4:     reward = [sigmoid(std(servers.Utilization)),
sigmoid(std(servers.queueOccupation),
5:     median(servers.Utilization),
          median(servers.queueOccupation)]
6:  else
7:     reward = sigmoid(std(servers.Utilization))
8:  end if
End Algorithm
```

Algorithm 6 outlines the task of the ***sortQtable*** function. This algorithm is essential in the RLScheduler to arrange the Q-table based on the provided state using the specified ***sortingMethod*** function. If the ***sortingMethod*** is set to 'NDS' (non-dominated sorting), the function employs the "nonDominatingSorting" method from the ***RLScheduler*** (Algorithm 3) instance. This function assigns ranks to each row in the Q-table using non-dominated sorting, a technique useful for handling

multi-objective optimization problems. Non-dominated sorting groups solutions into different levels of dominance, assisting in identifying the best solutions. Alternatively, if the function is set to any other value, it uses the "lexographicalSorting" method of the ***RLScheduler*** instance. This method assigns ranks to each row in the Q-table using lexicographic sorting, which is a technique for sorting elements based on multiple criteria in order of importance.

After determining the ranks, the function updates the Q-table for the given state with the new ranks. This process enhances the efficiency and effectiveness of the RLScheduler by organizing the Q-table based on the chosen sorting method.

Algorithm 6 SortQtable task

Input:
(1) state
(2) sortType

Start Algorithm
1: if sortType == 'NDS' then
2: ranks = self.nonDominatingSorting(self.QtableMO)
3: else
4: ranks = self.lexographicalSorting(self.QtableMO)
5: end if
6: update self.Qtable[state] with the new ranks
End Algorithm

4.5. Migrate Agent

The Migrate RL algorithm aims to handle changing server workloads by moving tasks between servers. In Algorithm 7, the process starts by obtaining the current environment state and checking if it's time to update the Q-table. If an update is needed, the algorithm calculates the reward for the previous action and adds it to the overall reward. Next, the algorithm selects a set of servers with tasks to migrate and picks an action using the Q-table. If a randomly generated number is less than the exploration probability, a random action is chosen; otherwise, the action with the highest Q-value is selected. This chosen action indicates the server from which a task will be moved, and the heaviest task is selected for migration.

Afterward, the algorithm updates the Q-table with the current state and chosen action for this period. The servers and devices requiring migration are stored in a data structure for future use. Finally, the exploration probability is reduced using a decay factor to ensure the algorithm eventually selects the best action. This process continues in a continuous background loop, where the algorithm makes decisions based on the current environment state, updates the Q-table, and selects the optimal action for task migration.

Algorithm 7 Migrate RL task

Input:
(1) timeCounter
(2) servers : snapshot of the servers status.
(3) delayPeriod: when to get the reward.
(4) bandwidth: to calculate the migration cost

Output:
(1) serversToMigrateFrom: the selected servers and the device that we will migrate its task

Start Algorithm
1: currentState = self.getState()
2: if timeCounter in self.QtableUpdates then
3: reward = self.updateQtable(servers, timeCounter, rewardType='migration-reward')
4: self.overAllRewards.add(reward)
3: end if
4: for i in len(servers)/3 do
5: capableServers = any server that has a task to migrate
6: if randomNumber < self.epsilon then
7: action = Random(capableServers)
8: else
9: action = argmax(self.Qtable[currentState][capableServers])
10: end if
11: device = action.getHeaviestTask()
12: self.QtableUpdates[timeCounter + delayPeriod].add(currentState, action)
13: serversToMigrateFrom.add(action, device)
14: end for
*15: self.epsilon = max(self.epsilon * self.decay, self.minEpsilon)*
End Algorithm

5. EXPERIMENTAL WORKS AND RESULTS

The experimental section is divided into two parts. In the first part (5.1), we present the experimental design, and in the second part (5.2), we present the experimental results and analysis.

5.1 Experimental Design

For simulation the training was executed on Windows 11 OS with core i7 10 gen and RAM 16 G. The parameters that were used are depicted in Table 1. The experiment discussed evaluates four different scheduling methods in the context of fog cloud computing optimization RL schedulers and RL migrators. These methods are GR_c (Greedy based on CPU), GR_r (Greedy based on Reliability), RLRM (Random Multi-objectives), and RLRLM (RL Multi-objectives). The performance of these

methods is assessed using metrics such as average utilization, number of tasks completed, serve rate, and delay.

Table 1. Parameters for the experimental design

Parameters	Value
ε	0.99
DECAY	0.999
ε_{min}	0.1
γ	0.5
α	0.6
DELAY PERIOD	2

Each of them is explain as below:

i. ***GR_c*** prioritizes tasks by considering the CPU utilization of servers.

ii. ***GR_r*** prioritizes tasks based on the reliability of servers.

iii. ***RLRM*** employs RL for scheduler and random migration for migrator with multi-objective reward model.

iv. **RLRS** – employs RL for scheduler and random for migrator. It uses single objective function for reward.

v. ***RLRLM*** utilizes RL to optimize multiple objectives scheduler and migrator. It uses multi-objective function for rewarding.

5.2 Results and Analysis

The evaluation is summarized in Table 2, which provides details of the performance assessment used in this study. To assess the effectiveness of the methods, the experiment examines the average server utilization in scenarios with varying CPU capacities. Additionally, the performance of these methods is evaluated using a reward-based approach, where the reward is determined based on either migration or multiple objectives.

Here are the performance metrics observations:

i. **Average utilization**: RLRLM achieved the highest average utilization at 88.63%, followed closely by RLRM at 83.40% and RLRS at 83.71%. In comparison, GR_c had an average utilization of 80.25%, while GR_r had a much lower average utilization of only 51.84%.

ii. **Tasks completed**: RLRLM achieved the highest number of completed tasks at 9981, followed by RLRM with 9607, and RLRS with 9671. In comparison, GR_c completed 8962 tasks, while GR_r had a much lower completion rate.

iii. **Serve rate:** Higher values indicating that tasks were completed more quickly. RLRLM had the highest serve rate at 0.725, followed by RLRS at 0.703, and RLRM at 0.698. In comparison, GR_c had a serve rate of 0.651, while GR_r had a serve rate of only 0.361.

iv. **Delay:** Shows the average delay between when a task was submitted and when it was completed, with lower values indicating that tasks were completed more quickly. GR_r had the lowest delay at 100.215, followed by RLRLM at 58.071, and RLRS at 62.687. In comparison, RLRM had a delay of 62.700, while GR_c had a delay of 71.777.

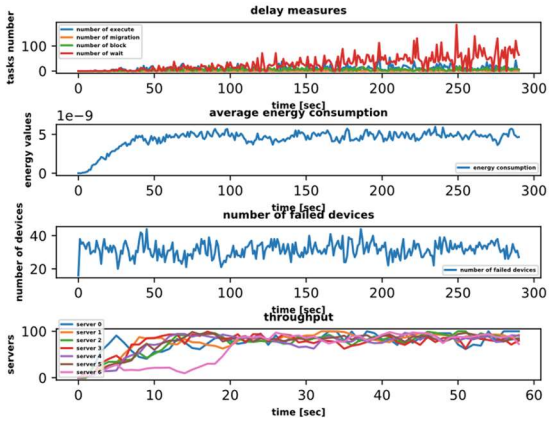
Table 2. Performance metric for different scheduler types

Methods	Avg Utilization (%)	Task	Serve Rate (task/ ms)	Delay (ms)
RLRM	83.40	9607	0.70	62.70
RLRLM	88.63	9981	0.73	58.07
RLRS	83.71	9671	0.70	62.69
GR_r	51.84	4961	0.36	100.22
GR_c	80.25	8962	0.65	71.78

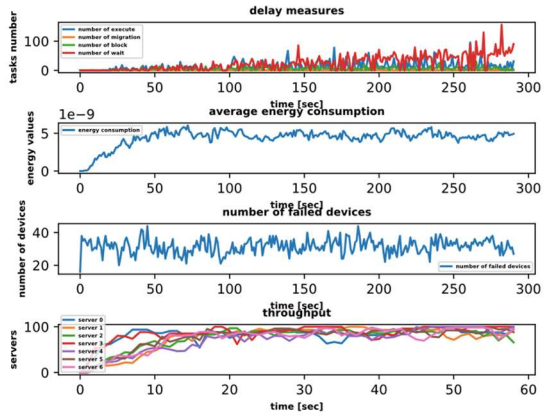
From the numerical values, it is evident that the RL-based methods generally performed better than the greedy-based methods concerning average utilization, the number of completed tasks, serve rate, and delay. Specifically, RLRLM, utilizing RL for multiple objectives, outperformed RLRM, which used reinforcement learning for a single objective. The results also indicate that the choice of migration method and reward type can influence performance. As mentioned earlier, both RLRM and RLRLM use RL for scheduling with the difference of employing random migrators for RLRM.

For a more detailed analysis, we provide the time series data for each metric, including delay, average energy consumption, number of failed devices, and throughput, for both RLRM and

RLRLM. The results indicate that the number of waiting tasks in RLRLM is lower than in RLRM. Additionally, the energy consumption in RLRLM is slightly lower by approximately 5 × 10^{-9} watt compared to RLRM, but it is still equivalent. The throughput remains stable for both RLRLM and RLRM, consistently reaching between 90% to 100%.



a) RLRM



b) RLRLM

Figure 1 Time series of evaluation metrics for developed RLRM and RLRLM

6. DISCUSSION

Both RLRM and RLRLM consistently outperform the greedy-based methods in terms of average utilization, completed tasks, serve rate, and delay. Notably, RLRLM, which optimizes multiple objectives, exhibits even better performance than RLRM, which focuses on a single objective. These findings support existing literature advocating for the effectiveness of reinforcement learning-based methods in fog cloud computing optimization. Moreover, this study goes further by demonstrating the clear advantage of multi-objective optimization. The significant performance improvement of RLRLM emphasizes the potential of RL-based methods in this field, filling a crucial gap in the literature where the advantages of such methods have not been thoroughly explored.

7. CONCLUSION AND FUTURE WORKS

This study introduces a new load-balancing system that optimizes fog-cloud computing using the RL approach. It employs two RL agents—one for task allocation to fog or cloud nodes and another for task migration between fog nodes, aiming for fairness and higher throughput. Unique models for state, action, and reward functions enable effective collaboration in the load-balancing process. The study explores three types of rewards for the RL agents: single-objective, multi-objective with non-dominated sorting, and multi-objective with lexicographical sorting.

Experimental results reveal that RL-based scheduling methods outperform greedy-based methods across various performance measures. However, this study has limitations. The experimental setup focuses primarily on fog-cloud computing systems and may not fully generalize to other distributed systems. The RL agents' specific state, action, and reward models might be optimal only for certain scenarios. There are also knowledge gaps in exploring reinforcement learning techniques' applications and limitations in fog-cloud computing optimization.

Future work can delve into additional optimization metrics like security and privacy and investigate scalability and adaptability with different fog-cloud computing setups. Moreover, exploring more robust and adaptable state, action, and reward models for the RL agents can further enhance the system performance. This research's significance lies in its potential to improve efficiency in the rapidly growing domain of fog-cloud computing and provide a strong foundation for future investigations into RL-based load balancing solutions in this field.

REFERENCES

[1] H. Abreha, C. Bernardos, … A. O.-… J. of A. H., and undefined 2021, "Monitoring in fog computing: state-of-the-art and research challenges," *inderscienceonline.com*, vol.

36, no. 2, p. 114, 2021, doi: 10.1504/ijahuc.2021.113384.

[2] H. F. Atlam, R. J. Walters, and G. B. Wills, "Fog computing and the internet of things: A review," *Big Data and Cognitive Computing*, vol. 2, no. 2, pp. 1–18, Jun. 2018, doi: 10.3390/BDCC2020010.

[3] N. Mohamed, J. Al-Jaroodi, S. Lazarova-Molnar, and I. Jawhar, "Applications of integrated iot-fog-cloud systems to smart cities: A survey," *Electronics (Switzerland)*, vol. 10, no. 23, Dec. 2021, doi: 10.3390/ELECTRONICS10232918.

[4] M. M. Kamruzzaman, B. Yan, M. N. I. Sarker, O. Alruwaili, M. Wu, and I. Alrashdi, "Blockchain and Fog Computing in IoT-Driven Healthcare Services for Smart Cities," *J Healthc Eng*, vol. 2022, 2022, doi: 10.1155/2022/9957888.

[5] P. Singh, R. K.-I. Software, and undefined 2022, "A software-based framework for the development of smart healthcare systems using fog computing," *Wiley Online Library*, 2022, doi: 10.1049/sfw2.12081.

[6] B. Jamil, M. Shojafar, I. Ahmed, A. Ullah, K. Munir, and H. Ijaz, "A job scheduling algorithm for delay and performance optimization in fog computing," *Concurr Comput*, vol. 32, no. 7, Apr. 2020, doi: 10.1002/CPE.5581.

[7] M. Haghi Kashani, A. M. Rahmani, and N. Jafari Navimipour, "Quality of service-aware approaches in fog computing," *International Journal of Communication Systems*, vol. 33, no. 8, May 2020, doi: 10.1002/DAC.4340.

[8] M. Al-Khafajiy, T. Baker, H. Al-Libawy, … Z. M.-F. G., and undefined 2019, "Improving fog computing performance via fog-2-fog collaboration," *Elsevier*, Accessed: May 07, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X18331868

[9] G. S. S. Chalapathi, V. Chamola, A. Vaish, and R. Buyya, "Industrial internet of things (iiot) applications of edge and fog computing: A review and future directions," *Advances in Information Security*, vol. 83, pp. 293–325, 2021, doi: 10.1007/978-3-030-57328-7_12.

[10] R. K. Naha, S. Garg, and M. B. Amin, "Fuzzy Logic-based Robust Failure Handling Mechanism for Fuzzy Logic-based Robust Failure Handling Mechanism for Fog Computing," *arXiv preprint arXiv:2103.06381*., Mar. 2021, Accessed: Apr. 20, 2021. [Online]. Available: http://arxiv.org/abs/2103.06381

[11] D. Ha and Y. Tang, "Collective intelligence for deep learning: A survey of recent developments," *Collective Intelligence*, vol. 1, no. 1, p. 263391372211148, Aug. 2022, doi: 10.1177/26339137221114874.

[12] G. Shruthi, M. R. Mundada, B. J. Sowmya, and S. Supreeth, "Mayfly Taylor Optimisation-Based Scheduling Algorithm with Deep Reinforcement Learning for Dynamic Scheduling in Fog-Cloud Computing," *Applied Computational Intelligence and Soft Computing*, vol. 2022, 2022, doi: 10.1155/2022/2131699.

[13] G. Mattia, R. B.-2022 I. I. conference on, and undefined 2022, "On real-time scheduling in Fog computing: A Reinforcement Learning algorithm with application to smart cities," *ieeexplore.ieee.org*, doi: 10.1109/PerComWorkshops53856.2022.9767498.

[14] F. M. Talaat, M. S. Saraya, A. I. Saleh, H. A. Ali, and S. H. Ali, "A load balancing and optimization strategy (LBOS) using reinforcement learning in fog computing environment," *J Ambient Intell Humaniz Comput*, vol. 11, no. 11, pp. 4951–4966, Nov. 2020, doi: 10.1007/S12652-020-01768-8.

[15] A. Orhean, F. Pop, I. R.-J. of P. and D. Computing, and undefined 2018, "New scheduling approach using reinforcement learning for heterogeneous distributed systems," *Elsevier*, Accessed: May 05, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731517301521

[16] J. Santos, T. Wauters, … B. V.-2021 I., and undefined 2021, "Resource provisioning in fog computing through deep reinforcement learning," *ieeexplore.ieee.org*, Accessed: May 07, 2023. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9464049/

[17] S. Swarup, E. M. Shakshuki, and A. Yasar, "Energy Efficient Task Scheduling in Fog Environment using Deep Reinforcement Learning Approach," *Procedia Comput Sci*, vol. 191, pp. 65–75, 2021, doi: 10.1016/J.PROCS.2021.07.012.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

[18] M. Duggan, J. Duggan, E. Howley, E. B.-M. Computing, and undefined 2017, "A reinforcement learning approach for the scheduling of live migration from under utilised hosts," *Springer*, vol. 9, no. 4, pp. 283–293, Dec. 2017, doi: 10.1007/s12293-016-0218-x.

[19] M. M. Razaq, S. Rahim, B. Tak, and L. Peng, "Fragmented Task Scheduling for Load-Balanced Fog Computing Based on Q-Learning," *Wirel Commun Mob Comput*, vol. 2022, 2022, doi: 10.1155/2022/4218696.

[20] J.-Y. Baek, G. Kaddoum, S. Garg, K. Kaur, and V. Gravel, "Managing Fog Networks using Reinforcement Learning Based Load Balancing Algorithm," pp. 15–18, 2019.

[21] P. Gazori, D. Rahbari, and M. Nickray, "Saving time and cost on the scheduling of fog-based IoT applications using deep reinforcement learning approach," *Future Generation Computer Systems*, vol. 110, pp. 1098–1115, 2020, doi: 10.1016/j.future.2019.09.060.