Mobile Networks and Applications
https://doi.org/10.1007/s11036-022-01974-y

An Optimized Collaborative Scheduling Algorithm for Prioritized Tasks with Shared Resources in Mobile-Edge and Cloud Computing Systems

Amira A. Amer[1,2] · **Ihab E. Talkhan**[2] · **Reem Ahmed**[3] · **Tawfik Ismail**[1,4]

Accepted: 13 December 2021

Abstract
Mobile edge computing (MEC) is a promising technology that has the potential to meet the latency requirements of next-generation mobile networks. Since MEC servers have limited resources, an orchestrator utilizes a scheduling algorithm to decide where and when each task should execute so that the quality of service (QoS) of each task is achieved. The scheduling algorithm should use the least possible resources required to meet the service demands. In this paper, we develop a two-level cooperative scheduling algorithm with a centralized orchestrator layer. The first scheduling level is used to schedule tasks locally on MEC servers. In contrast, the second level resides at the orchestrator and assigns tasks to a neighboring base station or the cloud. The tasks serve in accordance with their priority, which is determined by the latency and required throughput. We also present a resource optimization algorithm for determining resource distribution in the system in order to ensure satisfactory service availability at the minimum cost. The resource optimization algorithm contains two variations that can be employed depending on the traffic model. One variant is used when the traffic is uniformly distributed, and the other is used when the traffic load is unbalanced among base stations. Numerical results show that the cooperative model of task scheduling outperforms the non-cooperative model. Furthermore, the results show that the suggested scheduling algorithm performs better than other well-known scheduling algorithms, such as shortest job first scheduling and earliest deadline first scheduling.

Keywords Mobile edge computing · Scheduling algorithms · Quality of service · Power consumption · Optimization · Cloud computing

1 Introduction

Internet of things (IoT) applications have grown rapidly in recent years, and as digital transformation accelerates, more smart devices are projected to be connected to next-generation networks. Furthermore, the growth of IoT devices has resulted in significant advances in industrial processes, transforming how people interact with and govern the physical environment [1] . However, these smart devices have computational and energy limitations, preventing them from executing computationally intensive operations required by current applications such as augmented reality (AR) and artificial intelligence (AI) [2]. Therefore, IoT devices send the data collected to processing centers via the internet [3]. Cloud computing offers powerful servers that can be used for offloading IoT tasks. However, due to the extreme remoteness of cloud servers, cloud computing has high latency responses, which might not fulfill the requirements of upcoming low latency applications [4]. Furthermore, due to

✉ Tawfik Ismail
tismail@cu.edu.eg

Amira A. Amer
amira.Amer@nu.edu.eg

Ihab E. Talkhan
italkhan@aucegypt.edu

Reem Ahmed
reem@sci.cu.edu.eg

1 Wireless Intelligent Networks Center (WINC), Nile University, Giza 12677, Egypt

2 Department of Computer Engineering, Faculty of Engineering, Cairo University, Giza 12613, Egypt

3 Department of Mathematics, Faculty of Science, Cairo University, Giza 12613, Egypt

4 National Institute of Laser Enhanced Sciences, Cairo University, Giza 12613, Egypt

Published online: 20 April 2022

bandwidth limitations, the high traffic load predicted in next-generation network applications cannot be fully offloaded to the cloud.

Mobile edge computing (MEC) is seen as a feasible alternative approach in order to solve the challenges that cloud computing might have. MEC moves computing and storage resources from a centralized cloud to the edge of the network, where they are closer to end devices. Because of its closeness to the task origin, the MEC can significantly reduce latency, making it the key to offering delay-sensitive services. Serving tasks at edge devices also reduces the amount of data transferred through the network backbone, decreasing the probability of network congestion and preserving data privacy. However, the MEC resources are much more constrained than cloud resources, which limits the number of tasks that could be served on the MEC layer.

A hybrid edge-cloud architecture can take advantage of low latency in MEC and powerful cloud resources. MEC will complement the cloud by serving low latency tasks while the cloud serves the rest tasks. For example, the MEC can aggregate and preprocess data before sending it to the cloud servers, which decreases the workload and traffic destined to the cloud [5]. Furthermore, the hybrid architecture can be used to optimize the number of completed tasks in the system which meet the requirements [6]. The performance of the overall system could be further enhanced by sharing resources between edge nodes within the same cluster which increases the utilization and capacity of the system. To enable this hybrid architecture, the ITU recommends having a system responsible for orchestrating resources in the network [7].

The orchestrator manages the computation resources and network resources by scheduling when and where each task should be served. The orchestrator considers the computational requirements, latency, and task priority when building a scheduling strategy. The scheduling strategy should focus on maximizing the number of completed tasks and minimizing the average waiting time of tasks, the system's energy consumption, and the monetary cost. Another issue facing the task scheduling problem is user mobility. For example, a user placing a request to one MEC server might move away from this server, making the estimated communication delay change. A policy for handling tasks during handover is needed so that user mobility does not affect the system's reliability.

Various works were directed to solve the task scheduling problem in edge computing, however a large portion of these works considered task scheduling only in an offline setting making them impractical for real-time systems. Recent works introduced online scheduling solutions that can be deployed in a real-time edge environment [8]. Nevertheless, current online scheduling solutions ignore some system constraints, such as network link capacity, and are also tested for tasks with deadlines much more relaxed compared to 5G specifications. Moreover, no research was directed towards designing the resource distribution across edge nodes in the network. All works assume the available resources arbitrary without considering the consequences of having these resources on the system cost and energy consumption. Increasing the number of resources in a system can help satisfy the QoS of more tasks but will increase the initial system cost and the energy consumption of the MEC layer. In order to satisfy the goal of energy minimization in next-generation networks, the resource design optimization problem should not be ignored. This paper introduces an online scheduling algorithm in a cooperative edge environment. The scheduling algorithm considers network link constraints as well as computation resource constraints and task QoS constraints. The paper also introduces a resource design optimization algorithm that can operate on different network traffic models These algorithms aim to a) maximize the system availability while maintaining a reasonable quality of service (QoS) satisfaction rate for all tasks. b) minimize the total system cost. c) decrease the average waiting time of tasks.

The rest of the paper is organized as follows. Section 2 discusses the previous work done on task scheduling. Section 3 introduces the system architecture on which the scheduling algorithm is based. In Section 5, the details of the cooperative scheduling algorithm are explained. Section 4 formulates the resource optimization problem and explains how particle swarm optimization was used to solve this problem. In Section 6, the cooperative scheduling algorithm is implemented and compared to other scheduling techniques. Finally, we conclude and discuss future research directions in Section 7.

2 Literature Survey

This section presents alternative orchestrator designs discovered in literature and previous studies on task scheduling. Additionally, we also discuss the benefits and challenges associated with various approaches.

2.1 Orchestrator Architecture

The orchestrator is the system that allocates and schedules the resources and tasks. In many cases, the orchestrator can be used as a standalone system, or its functionality can be distributed among edge nodes. Both centralized and decentralized approaches were described in the literature.

2.1.1 Decentralized Orchestration

A decentralized approach for orchestration was presented in [9] to mitigate the single point of failure problem found in centralized systems. The disadvantage of decentralization is overloading the MEC servers due to the additional orchestration function and reducing the system security by sharing more data amongst edge nodes. The system also needs additional links between the edge nodes to facilitate data exchange. Blockchain technology was suggested by Tuli et al. [10] to reduce the security risk associated with decentralization, however no solutions were suggested for data exchange overheads. To avoid using a massive number of control and reduce security vulnerabilities, a centralized orchestration approach can be adopted instead.

2.1.2 Centralized Orchestration

Fadahunsi and Maheswaran [11] suggested a centralized resource management system placed in the cloud for optimizing the load balance across edge nodes. The disadvantage of placing the orchestration function at the cloud is the latency introduced by transmitting each task arriving to the cloud and waiting for the allocation response. Alternatively, the orchestration function can be placed at an edge cluster controller as proposed in [12, 13], and [14]. The system uses clusters close to the edge nodes to reduce the latency compared to cloud orchestration since the cluster controller is geographically near the edge nodes. Our work uses an edge cluster orchestration similar to [12, 13], and [14] to reduce the orchestration latency. The main challenge with employing an orchestration layer in a centralized manner is introducing a single point of failure yo the system. This could be resolved using a cloud controller that assigns one of the edge nodes as the cluster head and provides it with the most current state of the previous cluster head in case of failure.

2.2 Resource Allocation and Task Scheduling Algorithms

Many efforts have been made to develop and realize efficient resource allocation and scheduling algorithms. Each algorithm aims to meet the quality of service (QoS) requirements of the tasks in the system. Furthermore, these algorithms may also seek to optimize additional aspects of the system, such as response time, energy consumption, utilization rate, and monetary cost.

2.2.1 Offline Task Allocation Algorithms

Chen et al. [15] used a concave-convex procedure (CCCP) based algorithm to choose the device that minimizes the energy consumed constrained by the maximum acceptable delay, minimum SINR, computing capacity, and energy. Zhang et al. [16] use an iterative search algorithm that combines interior penalty function with D.C. programming (IPDC) to minimize the weighted sum of both latency and power consumption. To reduce the number of parameters used in optimization, the problem was divided into four subproblems. First, the algorithm allocates the optimum CPU cycle frequency of the mobile device that optimizes the energy-time tradeoff in the local computation case. Second, the channel that minimizes the interference is allocated for the requesting device. Third, the optimum transmission power is chosen. Finally, the offloading device that minimizes the weighted cost function is chosen. The disadvantage of the two previous methods is that devices burden the orchestrator with their local scheduling and more information needs to be collected from each connected device. Wang et al. [17] suggested calculating the cost based on future requests rather than considering just the current state of the system. The goal is to choose an assignment that minimizes the delay cost of the current request and future requests within a defined look-ahead window constrained by the resource capacity. Bahreini et al. [18] proposed a solution for resource allocation in edge-cloud environments that uses an auction-based technique. Users request several resources from the different VM types available in the system. The value gained by a user is affected by the request importance and the allocation decision. The resource allocation problem formulated aims to maximize the total value gained by all users and solved using a proposed linear program based approximation technique and a greedy technique. The proposed techniques showed near-optimum performance at a much lower execution time. Huang et al. [19] suggested a heuristic for allocating virtual machines (VMs) on physical machines in a cloud environment, which closely resembles allocating tasks on edge resources. Their heuristic aims to decrease the energy consumption in the system while increasing the CPU utilization rates. The heuristic successfully balanced energy consumption, CPU utilization, and service-level agreement violations compared to other algorithms.

Wang et al. [14] used a deep reinforcement learning (DRL) resource allocation scheme that aims to balance allocated resources and minimize the average service time. Resource balancing is achieved by minimizing the variance in loads on network resources and computational resources. They built a deep Q network, where states represent incoming requests. Actions represent the resource assignment given to the requests and rewards depending on the average service time and resource balancing of the chosen action. Zhao et al. [20] used a multi-agent DRL technique based on double-Q strategy and dueling architecture to solve the joint problem of user association and resource allocation.

Although offline task allocation algorithms found in literature consider various system constraints, however these algorithms assume that all tasks arrive together and optimized in one go. This assumption does not hold in a real-time environment where tasks arrive asynchronously. Therefore, an online scheduling technique is needed to solve the scheduling problem for real-time systems.

2.2.2 Online Scheduling Algorithms

Lin et al. [21] presented Petrel, an application aware distributed scheduling technique for edge nodes that aims to provide load balancing and improve the system performance. The algorithm serves tasks on each node in a first come first serve manner. To decrease the latency, the algorithm checks if the edge node has idle virtual machines (VMs) to assign the task to. If the edge node has no idle VMs, then the algorithm chooses two random neighbour nodes and selects the node with the lower load as a candidate node. According to the application type, the scheduling policy is determined. For latency-sensitive applications, a greedy approach is taken where the task is assigned to either the local edge or the candidate neighbour edge according to which has an earlier expected finish time. For latency-tolerant tasks, a best effort approach is taken where the task is assigned to the candidate neighbour if it has idle VMs otherwise the scheduling of the task is delayed within the latency bound to be scheduled later. Results showed that their proposed approach increased the average task speedup and the edge node throughput compared to other scheduling techniques. Unlike our work, the effect of network link constraints was not taken into consideration while assigning the tasks. The authors also stated that the cloud acts only as a backup for task execution and did not exploit the added advantage of scheduling tasks on the cloud.

Chunlin et al. [22] used a scheduling algorithm that aims to improve the utilization of resources and minimize the response latency. First, they classify the tasks arriving and resources available in each edge cluster. When assigned to a cluster, the tasks are placed in the queue corresponding to their type and are scheduled in a first in, first out fashion within the chosen cluster. Next, they use a neural network to predict the expected task execution time for each task type on each edge cluster. Using genetic algorithms, the tasks are assigned to an edge cluster or the cloud, where the deadline is not exceeded. Thus, the latency of all tasks and resource utilization is optimized in the system. Although the utilization and execution time are improved, the QoS satisfaction rates are relatively low for tasks with short deadlines. They also considers a non-cooperative scenario only and do not explore the advantages of sharing resources between different edge nodes. In contrast, our work explores a cooperative edge scenario and provides a balance in QoS satisfaction rates between tasks with short deadlines and tasks with long deadlines. Xiang et al. [23] suggested a Lyapunov optimization framework to minimize the average response time of requests at a reasonable cost in an edge-cloud environment. The optimization problem is solved at each time slot to decide where the tasks found in the current time slot should be allocated. Their algorithm showed a good balance between response time and monetary cost. Ma et al. [24] used a centralized orchestrator to decide which queue should a task be admitted to in a cooperative edge-cloud setting. The task scheduling decision aimed to minimize the average task response time while keeping the cost of cloud resources used within budget. The decision is also constrained by the stability of the edge and cloud queues. A Lyapunov optimization framework is used to transform the problem into subproblems solved at each time slot. A water filling-based algorithm is used to solve the scheduling subproblem dynamically at each time slot. The water-filling based dynamic task scheduling algorithm achieved near-optimum results in much less time than optimal solving techniques. In both [23] and [24] the deadline of tasks are not taken into consideration, making it hard to assess the QoS offered by this algorithm. Our proposed algorithm considers the deadlines of tasks while allocating the appropriate resources to ensure QoS satisfaction of different services.

Adhikari et al. [25] proposed a scheduling algorithm dependent on multilevel feedback queues. Arriving tasks are assigned a priority according to their deadline and resource requirement. Tasks are then placed in the queue corresponding to their priority. Tasks in the highest priority queue are served first, followed by those in the lower priorities. Tasks are served within each queue on a first-come, first-serve basis and are assigned to the first device to fulfill its requirements. Tasks at lower priority queues are raised to a higher priority level after a given time to avoid starvation. Results showed the effectiveness of this method. However, the effect of network bandwidth and receiving a large number of tasks was not explored. The model proposed in this paper accounts for the network bandwidth constraints and the system was tested for a larger scale of users compared to [25].

2.3 Mobility Management

Bao et al. [26] suggested a handover protocol for tasks in an edge network. When a connection is redirected to another edge node, the old edge node forwards all processed tasks to the new node. If the old edge node receives a request before the handover is completed, the old node processes the task and sends the final result to the new node. When the user re-requests the tasks from the new edge node, the data is sent to the user without further processing. The protocol did not support the migration of unprocessed tasks to avoid unnecessary communication overhead in false handover alarms. Ouyang et al. [12] used a system where different edge nodes support different services. They discuss a method to decide the migration of services during user mobility. They try to minimize the communication and computing delays for mobile users while keeping the long-term total cost of service migrations below a certain threshold. The optimization algorithm decides which server a task should be assigned and if a service should be transferred to each server. Zhu et al. [27] proposed a network selection scheme for mobile users using multi-attribute decision theory and the fuzzy logic theory. The network selection decision is made based on the user's requirement for QoS, the cost of network association, and the network's load.

3 System Model and Constraints

This section introduces the proposed system model and the constraints that rule the task scheduling problem. The key notations used in the system and their descriptions are summarized in Table 1.

3.1 System Model

We consider a hierarchical system with four layers (end users, base stations (BS) equipped with CPUs, an orchestrator, and the cloud). Figure 1 illustrates the architecture of the hierarchical system used. The end user layer consists of N users, each of whom is associated with the BS with the most robust signal strength. Due to the limitation of computation resources in the end user layer, tasks are offloaded by users to their associated BS in the BS layer. The BS layer consists of M base stations, each BS schedules the received tasks individually on its available computation resources. Tasks that fail to be scheduled by the BS are sent to the orchestrator layer for global scheduling. An orchestrator is connected to all base stations in the BS layer in a tree topology. The orchestrator is also connected to the cloud layer, which is assumed to provide unlimited computation resources that can be used for task execution. Tasks are categorized into service types which reflect the priority and requirements of the task. Task categorization could be done using techniques such as that proposed in [28]. The orchestrator schedules a task to a neighbour BS with free computational resources or to the cloud according to the task deadlines, service type and the network conditions. Figure 2 illustrates the service-based system used and shows the interactions between the system entities and the order of scheduling events. The system operates in a discrete-time framework, with requests being planned at the beginning of each time slot. The system updates user-BS associations every crucial cycle, which consists of m time slots to capture user mobility. It is assumed that each user would only send one service request at a time. The user is free to request a new service once a request has been fulfilled or blocked. A task is assumed to be non-preemptive and to occupy only one CPU at a time similar to [29], but it may hold the CPU for more than one time slot.

Table 1 Notations used in system

Notation	Definition
M	Number of base station in the system
N	Number of users in the system
C_{cloud}	Link capacity between orchestrator and cloud
C_{BS}	Link capacity between orchestrator and base station
L_{cloud}	Length of cable between orchestrator and cloud
L_{BS}	Length of cable between orchestrator and base station
CPU_i	Number of CPUs on base station i
C_j	Throughput required by task j
t_{serving_j}	CPU holding time needed by task j
t_j^{wait}	Maximum waiting time allowed for task j
T_{prop_i}	Propagation delay from base station i to orchestrator
$T_{\text{prop}_{\text{cloud}}}$	Propagation delay from orchestrator to cloud
T_{comm_j}	Communication delay of task j
T_{queue_j}	Queuing time taken by task j
t_{slot}	Duration of one time slot
m	Number of time slots in a crucial cycle

3.2 System Constraints

A base station i has a set number of CPUs (CPU_i), representing the number of concurrent tasks that the BS can perform each time slot rather than the number of cores or servers employed. The total of available and occupied CPUs at every time slot in BS_i should be equal to CPU_i. The link between a BS and the orchestrator has a limited capacity of C_{BS}. The entire amount of incoming and outgoing traffic on the base station link should less than C_{BS}. The amount of traffic that can be offloaded to the cloud is limited by the capacity link between the orchestrator and the cloud. The capacity of the cloud link is equal to C_{cloud}.

A task j is defined with maximum allowed waiting time (t_j^{wait}), a required throughput (C_j), a priority, and a CPU serving time (t_{serving_j}). The serving time t_{serving_j} is assumed to be the same in all CPUs whether located at base stations or at the cloud. Therefore, the processing time is not considered during task delay minimization. The task delay is calculated as the sum of queuing time (T_{queue_j}) and communication delay (T_{comm_j}). The communication delay of a task j at BS i can be calculated as follows:

$$T_{\text{comm}_j} = \begin{cases} 0 & \text{if scheduled at BS } i \\ 2 * (T_{\text{prop}_k} + T_{\text{prop}_i}) & \text{if offloaded to BS } k \neq i \\ 2 * (T_{\text{prop}_i} + T_{\text{prop}_{\text{cloud}}}) & \text{if offloaded to cloud} \end{cases} \tag{1}$$

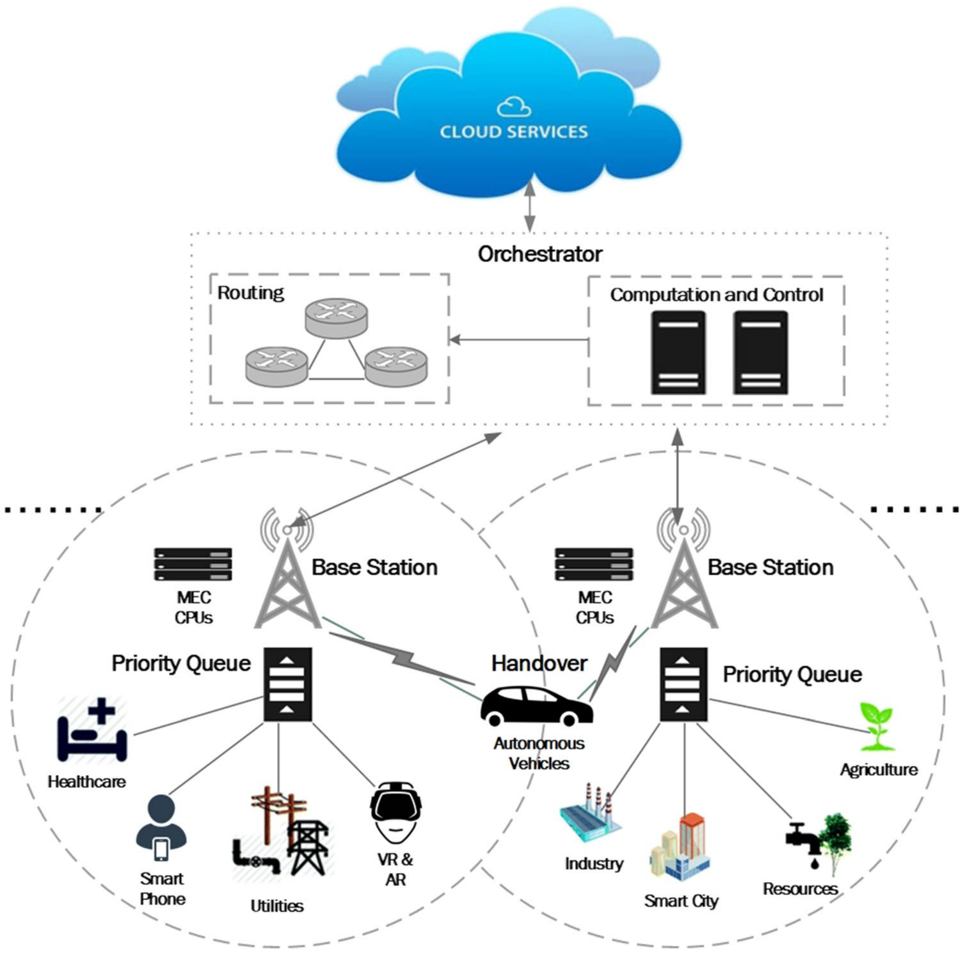

Fig. 1 Generic network architecture and topology

where T_{prop_i} is the propagation delay on link from BS i to the orchestrator. $T_{\text{prop}_{\text{cloud}}}$ is the propagation delay on link from orchestrator to the cloud. Since both links are assumed to be fiber optic links, the propagation delay per kilometer is approximately 5 μ s [30]. Thus, propagation delay on any link in ms can be calculated as $L \times 5 \times 0.001$ where L is the length of the link in km. The task delay is constrained by the maximum allowed waiting time, which can be formulated as:

$$T_{\text{queue}_j} + T_{\text{comm}_j} \leq t_j^{\text{wait}} \tag{2}$$

A task that fails the constraint in Eq. 2 gets blocked.

3.3 Mobility Management

Due to the mobility of users, a task can be placed inside the queue of one base station while the user moves to the coverage area of another base station. All user data is passed to the target base station during handover, including task results and unscheduled tasks. If a task j was scheduled before the user moves from BS i to BS m, the results will be forwarded to the orchestrator to route it to the base station where the user is located. The expression for communication delay stays the same as defined in Eq. 1 if task j was offloaded to BS m, but in case the task was offloaded elsewhere the communication delay will be:

$$T_{\text{comm}_j} = \begin{cases} T_{\text{prop}_i} + T_{\text{prop}_m} & \text{if scheduled at BS } i \\ T_{\text{prop}_i} + 2 * T_{\text{prop}_k} + T_{\text{prop}_m} & \text{if offloaded to BS } k \neq i \\ T_{\text{prop}_i} + 2 * T_{\text{prop}_{\text{cloud}}} + T_{\text{prop}_m} & \text{if offloaded to cloud} \end{cases} \tag{3}$$

The change in communication delay might cause a task failure if the actual delay was higher than the expected delay. However, the probability that a handover is performed near a task deadline is low since the deadline is in order of milliseconds, so even a high-speed vehicle is unlikely to cover enough distance that causes a handover.

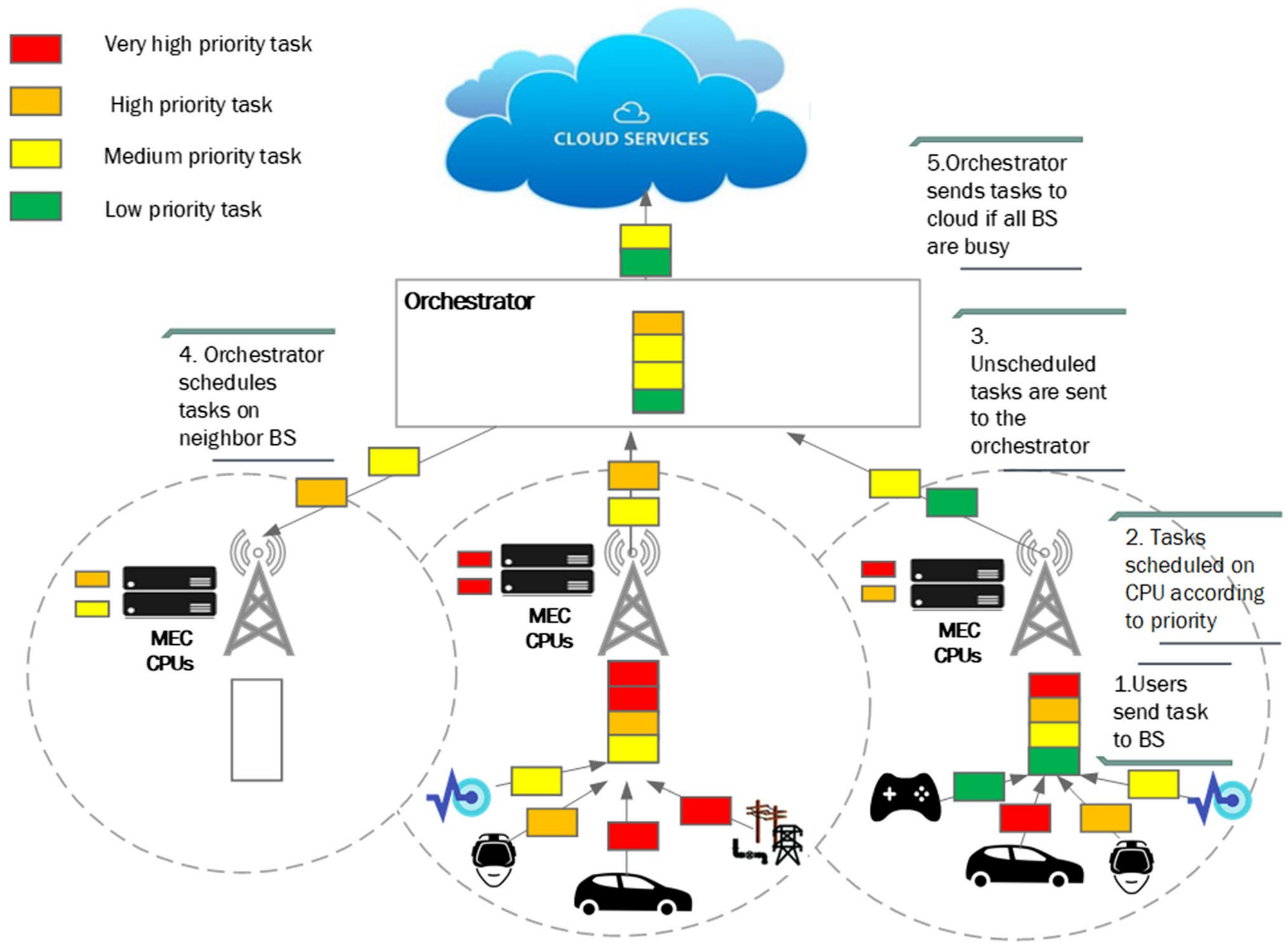
Very high priority task
High priority task
Medium priority task
Low priority task
CLOUD SERVICES
5.Orchestrator sends tasks to cloud if all BS are busy
Orchestrator
4. Orchestrator schedules tasks on neighbor BS
3. Unscheduled tasks are sent to the orchestrator
2. Tasks scheduled on CPU according to priority
1.Users send task to BS
MEC CPUs
MEC CPUs
MEC CPUs

Fig. 2 Case example of services-based segmentation

4 Resource Optimization

The distribution of CPUs across base stations is one factor that impacts the number of blocked requests in a network. Works in literature either distribute CPUs randomly while ignoring task blocking rates or assign a high number of CPUs to provide a non-blocking system. However, ignoring the task blocking rate can cause the user experience to drop and using a high number of CPUs increases the system's cost and energy consumption. Therefore, both aspects should be considered while designing the CPU distribution. Since there is no reference method to obtain CPU distribution considering both the blocking rate and the system cost, we proposed to model the CPU distribution as an optimization problem. Particle swarm optimization was a fast, practical solution to find the optimum CPU distribution instead of the impractical exhaustive search method.

The amount of resources necessary to provide acceptable blocking rates might be used to evaluate scheduling algorithms as well. An efficient algorithm should use fewer resources to provide the desired level of service availability.

4.1 Optimization Problem

The amount of resources needed in the system can be modeled as an optimization problem. The optimization problem aims to minimize the number of CPUs installed per BS (R) and have an acceptable blocking rate for incoming requests. The optimization problem can be formalized as:

$$\min(R) \text{subject to: } p_{\text{blocking}} < th_{\text{blocking}} \quad (4)$$

where p_{blocking} is the blocking probability of each service in the system under the current number of CPUs, and th_{blocking} is a defined blocking threshold for each service. If the blocking probability exceeds the threshold, the availability of the service will fall below the required QoS. The key terms used in describing and solving the optimization problem are summarized in Table 2.

4.2 Particle Swarm Optimization

Particle swarm optimization (PSO) is a population-based stochastic optimization technique first proposed in 1995 by Kenndy and Eberhart [31], inspired by the social behaviour of bird flocking or fish schooling. The basic idea of the PSO is to simulate the hunting behaviour of a swarm of birds looking for a food source. The PSO considers each bird as a particle, and the food source as the solution of an optimization problem [32]. In PSO, the potential solutions are proposed by the different particles. Two main values indicate each particle; its position and its velocity. Initially, all the positions and velocities of the particles are initialized randomly. Furthermore. all particles have fitness values obtained from the evaluation of the optimization problem. The particles explore the searching space of the optimization problem, hopefully, to find the optimal solution. Each particle updates its information based on two main position values, the own best position (pbest) and the global best position (gbest). Moreover, each particle keeps updating its position and velocity by following its own best solution until the current iteration and the best solution found by the whole swarm. Figure 3 illustrates the basic format for each iteration. The algorithm finishes it task after it satisfies the termination conditions.

In a D dimensional search space, the position and velocity of particle i are represented as $x_i = [x_{i1}, \dots, x_{iD}]^T$ and $v_i = [v_{i1}, \dots, v_{iD}]^T$, respectively. The best position previously reached by the particle is referred to as the particle's local best and is denoted by x_i^{best}. The best position among all particles is called the global best and is denoted by B. At each iteration, the position and velocity of particles are updated using the following equation [33]:

$$\begin{aligned} v_{i+1} &= \omega . v_i + c_1 r_1 \left(x_i^{\text{best}} - x_i\right) + c_2 r_2 (B - x_i) \\ x_{i+1} &= x_i + v_{i+1} \end{aligned} \tag{5}$$

where ω, c_1 and c_2 are constants, and r_1 and r_2 are random numbers uniformly distributed in the interval [0,1]. PSO was chosen for its robustness, quick convergence [34] and easy implementation [35].

In this problem, a particle represents the CPU distribution among base stations. Since the number of CPUs has to be within the positive range of numbers, the boundary mode

Table 2 Notations used in optimization

Notation	Definition
p_{blocking}	Blocking probability for each service
th_{blocking}	Blocking threshold for each service
R	CPUs per base station
$R_{\max}$	Maximum allowed number of CPUs per base station
$n_{\min}$	Minimum number of active users per base station
$n_{\max}$	Maximum number of active users per base station
w_1	Cost weight of one CPU per base station
w_2	Cost weight of exceeding the blocking threshold by one service
I	Number of simulation iterations
x_i	Position of particle i
v_i	Velocity of particle i
x_i^{best}	Local best position of particle i
B	Global best position in the swarm
ω	Particles' inertia weight
c_1	Cognitive weight parameter
c_2	Social weight parameter
I_{PSO}	Number of particle swarm iterations
$n_{\text{particles}}$	Number of particles used in the swarm

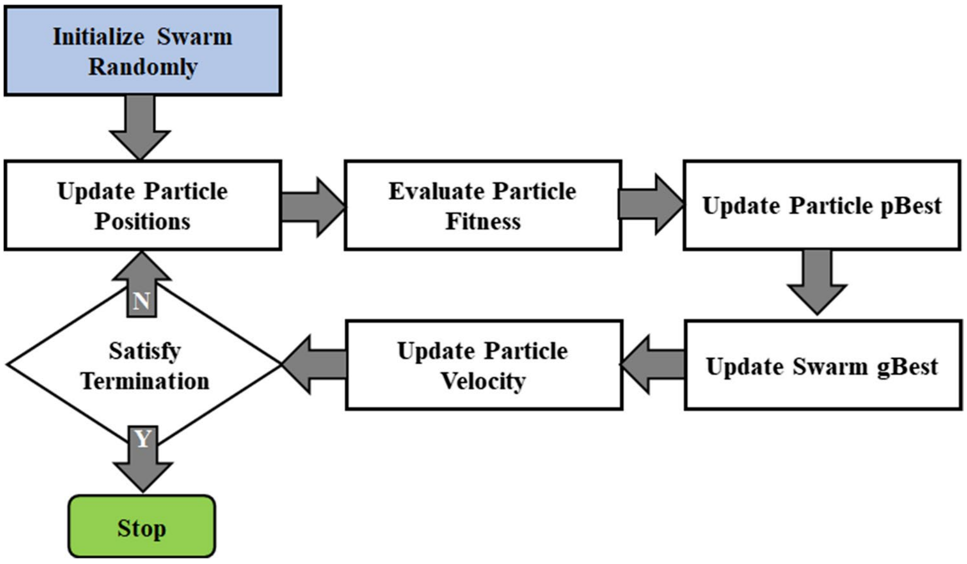

Fig. 3 Basic PSO algorithm flowchart

proposed in [36] can be used to keep the position of the particle between an upper boundary u and lower boundary l. The following check is applied after each update for each element x_{id} in the position vector of particle i:

$$x_{id} = \begin{cases} l & \text{if } x_{id} < l \\ u & \text{if } x_{id} > u \\ x_{id} & \text{o.w} \end{cases} \tag{6}$$

According to Eq. 4, the objective function should minimize the number of CPUs while having a blocking probability below the determined threshold. The cost of CPUs is represented as $w_1 R$ where R is the average number of CPUs per BS, and w_1 is the cost of one CPU. In order to include the cost of blocking, the number of services with blocking probability higher the threshold are counted and multiplied by w_2. w_2 represents the cost of having one service with a high blocking probability. The values of w_1 and w_2 are chosen such that $w_2 > w_1 R_{\max}$. This condition gives a higher priority to finding a solution that fulfills the condition $p_{\text{blocking}} < th_{\text{blocking}}$ for all services rather than focusing on CPU minimization.

The outcome of PSO is the minimum number of CPUs per BS needed to achieve acceptable blocking rates for the system under study. Algorithm 1 shows the steps of the PSO algorithm done to get R.

4.3 Optimization with Grouping

The PSO output in Algorithm 1 assumes an equal traffic load across all base stations, and therefore, the number of CPUs per BS is the same. In a real-life scenario, base stations within the same system may have unbalanced traffic loads and may need a different number of CPUs to cope with their load. In order to calculate the number of CPUs needed by each BS, the system can be divided into many subsystems. Each subsystem groups base stations that have similar traffic loads. Algorithm 1 runs for each subsystem independently. The total number of users in a subsystem is a percentage of the total number of users in the system N. For a system with G subsystems, this percentage should be estimated and given as an array $\beta = [\beta_1, \dots, \beta_G]$. The number of users in group g can be calculated as $N \times \beta[g]$. The upper and lower boundaries of users in each of the G subsystems are represented as $n_{\max} = [n_{\max_1}, \dots, n_{\max_G}]$ and $n_{\min} = [n_{\min_1}, \dots, n_{\min_G}]$ respectively. The number of base stations in each subsystem is also given as an array $M = [M_1, \dots, M_G]$. Algorithm 2 shows the steps to obtain the number of CPUs needed by each group of base stations within the same subsystem.

Algorithm 1 Particle Swarm Optimization

```
Input  : $N, M, I_{\text{PSO}}, I, n_{\text{particles}}, n_{\min}$, and $n_{\max}$
Output: $R$ of the global best
generate $n_{\text{particles}}$ particles of length $M$ with random positions for $I_{PSO}$ iterations do
    for each particle $i$ do
        set CPUs on each BS according to $x_i$ for $I$ simulation iterations do
            distribute $N$ users randomly on base stations bounded by $n_{\min}$ and $n_{\max}$ for crucial cycle length do
                generate traffic  call scheduling algorithm
            end
        end
        $p_{\text{blocking}} \leftarrow$ number of blocked requests per service/number of requests per service  $R \leftarrow \text{ceil}(\text{sum}(x_i)/M)$  $obj \leftarrow w_1 \times R + w_2 \times \sum(p_{\text{blocking}} > th_{\text{blocking}})$ if $obj$ < local best cost then
            $x_i^{\text{best}} \leftarrow x_i$
        end
        if $obj$ < global best then
            $B \leftarrow x_i$
        end
        update $v_i$ according to equation 5  update $x_i$ according to equation 5 & 6
    end
end
```

```
Algorithm 2 Groups CPU Optimization
Input   : $N, M, \beta, n_{\max}$, and $n_{\min}$
Output: $R$ for each group of base stations
for each subsystem group $g$ do
    $N_g \leftarrow N \times \beta[g]$  $R[g] \leftarrow$ call PSO in Algorithm 4.2 with
      $N_g, M[g], n_{\max}[g]$ and $n_{\min}[g]$ as input
end
```

5 Cooperative Scheduling Algorithm

A task *j* arrives at BS *i* is placed in a priority queue which sorts requests according to their priority number. If two requests are of the same priority, then the tasks are placed in a first-come, first-serve manner. Priority queues are implemented using binary heaps instead of sorted arrays to decrease the complexity of insertion from $O(n)$ to $O(\log n)$. At the beginning of the time slot, each base station dequeues all requests from the queue and schedules them on the available local CPUs. If requests in the queue exceed the number of available CPUs, the remaining requests are sent for scheduling at the orchestrator. A BS periodically updates the orchestrator with the number of available CPUs and link capacity available at each time slot. A request is blocked if its deadline was exceeded while waiting for resources or during propagation. Algorithm 3 shows the scheduling part done by the BS. Requests sent to the orchestrator are also placed in a priority queue similar to that found on the base stations. Requests are assigned to the device which will give the least delay time. Therefore, the orchestrator first attempts to schedule a request on a neighboring base station and then the cloud. If the request cannot be scheduled in the current time slot, it is placed in the queue to be scheduled in the next time slot. Algorithm 4 shows the global scheduling algorithm that runs on the orchestrator.

In both local and global scheduling steps, the priority of a service depends on both the allowed waiting time for the task (t_j^{wait}) and the throughput required by the task (C_j). A higher priority is given to tasks with smaller t_j^{wait}. If two tasks have the same t_j^{wait}, the task with a smaller C_j is given a higher priority. If two requests have the same t_j^{wait} and C_j, then the task that has the closest deadline is assigned with higher priority.

5.1 Complexity Analysis

The time complexity of Algorithms 3 and 4 are discussed below.

5.1.1 BS Scheduling

The complexity of line 1 is $O(n)$, assuming the length of the task queue is n. Lines $2-7$ are $O(1)$ each. The complexity of line 8 is $O(\log n)$ because priority queues are implemented using binary heaps. The complexity of Algorithm 3 is hence $O(n \log(n))$.

```
Algorithm 3 Base Station Scheduling
while task queue not empty do
    task ← dequeue task queue  if base station has available CPUs and task
    deadline not exceeded then
        allocate one CPU for task
    end
    else if task deadline is exceeded then
        send failure message to user
    end
    else
        enqueue task in binary heap queue at orchestrator $Q_{orc}$
    end
end
```

Algorithm 4 Orchestrator Scheduling

```
while $Q_{orc}$ not empty do
    $task \leftarrow$ dequeue $Q_{orc}$  if BS associated with the user has enough bandwidth for task then
        for each BS $i$ do
            if BS $i$ has available CPUs and has enough bandwidth for task then
                allocate one CPU for task at $i$  break
            end
        end
        if task not assigned and cloud link has has enough bandwidth for task then
            allocate one CPU for task at cloud  continue
        end
    end
    if task not assigned then
        enqueue task in binary heap queue at BS associated with the user
    end
end
```

5.1.2 Orchestrator Scheduling

The complexity of line 1 is $O(k)$, where k is assumed to be the length of the orchestrator queue. Both lines 2 and 3 are $O(1)$. Line 4 is $O(M)$, where M is the number of base stations connected to the orchestrator. Each line between 5 and 11 is $O(1)$ as each needs a constant execution time. Finally, line 12 is $O(\log(n))$ as line 8 in Algorithm 3. Therefore, the complexity of Algorithm 4 is $O(kM + k\log(n))$.

6 Simulation Results and Discussions

In this section, we perform a simulation to evaluate the performance of the proposed scheduling algorithms under different system conditions.

Table 3 Simulation parameters

Parameter	Value	Parameter	Value
M	20	L_{cloud}	200 km
L_{BS}	20 km	t_{slot}	2.5 ms
m	20	$R_{\max}$	175
$n_{\min}$	5	$n_{\max}$	100
w_1	1	w_2	200
I	1500	ω	0.9
c_1	1.49	c_2	1.49
I_{PSO}	200	$n_{\mathrm{particles}}$	200

6.1 Simulation Setup

Simulations use an architecture with 20 base stations, where each base station is linked to the orchestrator with a fiber link of $L_{\mathrm{BS}} = 20$ km cable, and the orchestrator is linked to the cloud with a fiber link of $L_{\mathrm{cloud}} = 200$ km cable. The slot duration t_{slot} is determined as the greatest common divisor of holding times of all services. The crucial cycle is set as the maximum time a task can be present in the system. The number of time slots per crucial cycle m can be calculated as crucial cycle time divided by t_{slot}. The PSO algorithm is implemented with java based on the pseudocode in Algorithm 1. The position and velocity vectors are modified to take integer values only. The particle positions were initialized using a discrete uniform distribution that follows the bounding range of CPUs. The velocity vector for all particles is initialized by 0. By running various PSO simulations, best results were achieved with 200 particle, 200 PSO iterations, $c_1 = c_2 = 1.49$ and $\omega = 0.9$. The bounding range for the number of CPUs was set to [0,175]. The value of $R_{\max}$ in the bounding range was also obtained

Table 4 Service types used

Service #	Wait Time(ms)	Throughput(Mbps)	Holding Time(ms)
1	2.5	10	2.5
2	2.5	100	25
3	25	10	2.5
4	25	100	25

through simulation. The values of w_1 and w_2 were set to 1 and 200 respectively in order to follow the condition $w_2 > w_1 R_{\max}$ discussed in Section 4. Algorithm 1 runs using each scheduling algorithm to get R needed by each algorithm. The scheduling algorithm runs for 1500 simulation iteration before calculating the blocking probability for each service. Simulation parameters are summarized in Table 3.

We assume that all requests in the system fall under one of four types of services. Each service type has a different combination of allowed waiting time and throughput. Table 4 describes the waiting time allowed and throughput needed by each service. These services are inspired by some services described by the 3GPP standards [37, 38]. The 3GPP specifications defined the services applications are fully described with Enhanced mobile broadband (eMBB), Ultra-reliable and low latency communications (URLLC) and Massive machine-type communications (mMTC). Based on these classifications, we listed the services into four categories [39]. Service 1 represents services characterized by having a low required waiting time and low required throughput. High voltage electricity distribution and V2X messaging for advanced driving are examples of service 1, as they require 3 ms maximum packet delay and maximum user experienced data rate of 10 Mbps. Service 2 is based on low latency enhanced Mobile Broadband applications and augmented reality, requiring low latency and high user experienced throughput rates (about 100 Mbps). Monitoring applications (e.g., health monitoring) require low data rates (around 1 Mbps) and a high end-to-end latency (around 50 ms), which is captured by service 3 requirements. Real-time gaming is an example of service 4 where it requires 30 ms maximum packet delay and high data rates. Live streaming can also be categorized under service 4, but it has a more relaxed maximum packet delay (80 ms).

Users are distributed on base stations using a discrete uniform distribution within the range $[n_{min}, n_{max}]$, such that the total number of users in the system is equal to N. The scheduling algorithms were tested under four different traffic loads $N = [500,1000,1500,2000]$. To generate traffic at each time slot, a service number is given to each free user. The service number is drawn from a discrete uniform distribution in the range [0,4], where 0 means that the user does not have a request in the current time slot and numbers from 1 to 4 represent the service type requested by the user.

Two different combinations of C_{BS} and C_{cloud} were used in the simulation. C_{BS} was chosen based on CPRI options 2 and 3 [40]. Table 5 shows the values of C_{BS} and C_{cloud} in each setting.

Table 5 Link capacity settings

Setting Number	C_{BS} (Gbps)	C_{cloud} (Gbps)
1	1	10
2	2.5	40

6.2 Scheduling Algorithms

The simulation is applied to five scheduling algorithms to compare their efficiencies. The baseline is the non-cooperative scheduling (NCS) technique. Three algorithms are based on the algorithm described in Section 5, namely cooperative scheduling (CS), earliest deadline first scheduling (EDFS), and shortest job first scheduling (SJFS). CS is the implementation of the technique described in Algorithms 3 and 4. EDFS and SJFS are variants of CS that use a different priority system. The last algorithm is the cooperative online scheduling algorithm (Petrel) proposed in [41].

6.2.1 Non-Cooperative Scheduling (NCS)

Non-cooperative scheduling is based on an architecture where the orchestrator is only used for routing and does not participate in the scheduling process. Requests arriving at base stations are placed in a priority queue that uses the same priority system as the one described in Section 5 for cooperative scheduling. Each BS assigns tasks on its local CPUs till no more CPUs are available. The remaining tasks are scheduled to the cloud. If the outgoing link from the BS to the orchestrator or the outgoing link to the cloud cannot carry the task, the task waits till the next slot to be scheduled. If a task exceeds the maximum allowed waiting time, then the task is blocked.

6.2.2 Earliest Deadline First Scheduling (EDFS)

The earliest deadline first scheduling (EDFS) is a dynamic scheduling algorithm for real-time systems introduced in [41], that assigns higher priority to tasks with the nearer deadline. In our simulation, EDFS uses Algorithms 3 and 4 but assigns priorities to tasks according to their deadlines despite the service type of the task. In case two tasks have the same deadline, the system will go back to the priority system used in Section 5 for cooperative scheduling.

6.2.3 Shortest Job First Scheduling (SJFS)

Shortest job first scheduling (SJFS) is a scheduling algorithm that prioritizes tasks with low CPU times to minimize their average waiting time [42]. SJFS also uses Algorithms 3 and 4 but assigns priorities to tasks according to their CPU holding times. In case of two tasks have the same holding time, the task with the nearer deadline is given a higher priority.

6.2.4 Petrel

Petrel is our implementation of the cooperative algorithm described in [21]. To accommodate the algorithm to our system a few modifications were done to the greedy and the best effort policies. In the greedy assignment policy, we assigned the task to the node with the lower expected finish time as stated but if the links are occupied, the task waits more time slots till the links are freed before occupying the CPU. In the best effort policy, the task is assigned to the neighbour node if the neighbour has idle CPUs and the links have enough capacity. The method to calculate the expected finish time after delay was not clearly stated, therefore we delay tasks instead if there are more than one CPU that will be free before the maximum waiting time of the task is reached. We also assign the task to the local node if the result of task delaying is not guaranteed.

6.3 Simulation Results

To check that the PSO algorithm described by Algorithm 1 works correctly, we run a sensibility test on the effect of varying the blocking threshold th_{blocking}. th_{blocking} sets a minimum threshold for QoS satisfaction rate that each service should achieve. When th_{blocking} is lowered, more computation resources will be needed to reach the required QoS satisfaction rate. To check that the algorithm results conform to the theoretical analysis, we run the PSO algorithm using three different blocking threshold ($th_{\text{blocking}} = [10^{-2}, 0.5 * 10^{-2}, 0.25 * 10^{-2}]$) on a system with link capacities described in setting 1. The simulations were done at different traffic loads. Figure 4 shows the number of CPUs needed at each th_{blocking} for different traffic loads. The results show that more CPUs are needed at lower th_{blocking} values as stated by the theoretical analysis.

Since the scheduling algorithm proposed runs at the start of each time slot, the running time needs to be shorter than the duration of the time. The algorithm running time should also be as negligible as possible to minimize the delay experienced by tasks. We measured the average running time of the scheduling algorithm at a high traffic density (100 users per BS) using setting 1. The average running time obtained was 0.058044 ms, which is a negligible time compared to the time slot duration of 2.5 ms.

To compare the five algorithms discussed, we run PSO using each algorithm under different traffic loads. The blocking threshold (th_{blocking}) was set to 10^{-2} for all algorithms to guarantee a fair comparison. We compare the

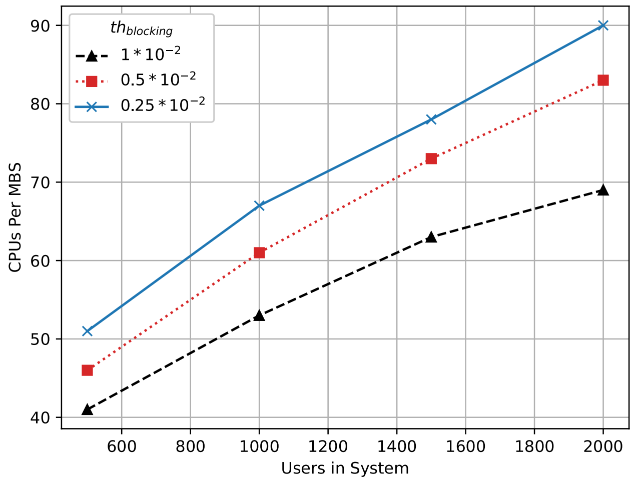

Fig. 4 CPUs needed at different blocking thresholds

algorithms under link capacities described by setting 1 and setting 2. Theoretically, the system should need less number of CPUs per BS at setting 2 because more tasks could be offloaded to the cloud since both C_{BS} and C_{cloud} were increased. Figures 5 and 6 show the number of CPUs per BS (R) needed at different traffic loads obtained by running PSO algorithm using setting 1 and setting 2 respectively. Comparing the two figures, it can be observed that the values of R are indeed less in setting 2 as stated by the theoretical analysis, which serves as another sensibility test for varying C_{BS} and C_{cloud}. The value of R reflects the system cost when using a given algorithm since installing more CPUs increases the system's initial cost and running cost. Therefore, the lower the value of R the more the algorithm is successful at minimizing the system cost.

Results in Figs. 5 and 6 show that CS have the lowest values of R compared to the other four algorithms making

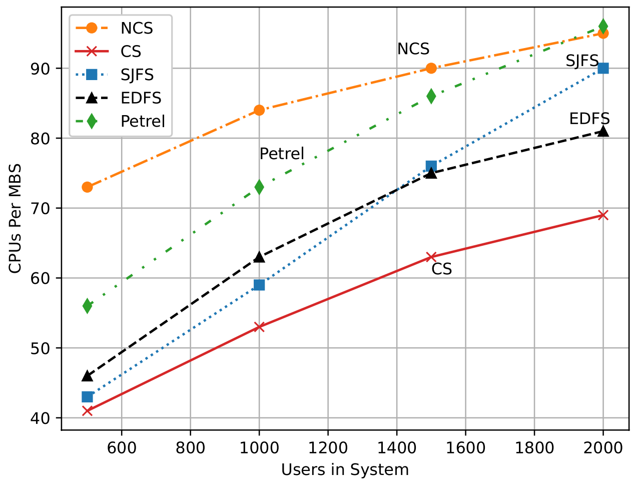

Fig. 5 Simulation results for setting 1

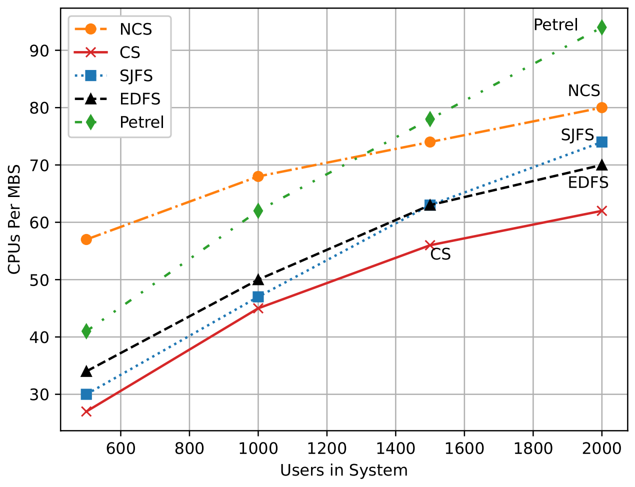

Fig. 6 Simulation results for setting 2

it the best at minimizing the system cost. The CS priority system combines the advantages of both shortest job first and earliest deadline first priority systems. CS gives a high priority to tasks with low allowed waiting time, so they have a higher probability of being scheduled before their deadlines. Also, it schedules tasks with low holding time first, which frees up the CPU and links for other tasks. It can be observed that Petrel has the worst performance between the cooperative techniques because it does not offload any traffic to the cloud, therefore it needs more computing resources at the edge. As the traffic load increases the performance of Petrel degrades more making it worse than the non-cooperative technique. At high traffic loads links are more occupied, exposing the negative effects of ignoring link capacities while assigning tasks. NCS has a higher CPU demand compared to CS, EDFS and SJFS because it does not benefit from idle CPUs found on neighbouring nodes. The results also show that the performance of SFJS degrades as the number of users in the system increases. The performance of SFJS does not scale well because higher priority is given to short tasks while long tasks are kept waiting regardless of their deadline. As the number of users increases, more tasks of types 2 and 4 are kept waiting till they are blocked. Therefore, more CPUs are needed to maintain the task blocking probabilities below the threshold.

We further compare the performance of the five algorithms at minimizing the average waiting time of tasks and maximizing the system availability. This comparison shows how well each algorithm balances the trade-off between performance and user experience. For a fair comparison, we get the results for each algorithm using an identical system setup. For all algorithms link capacities followed setting 1, R was set to 68 and N was set to 2000. The value of R was chosen as the smallest value shown in the results in Fig. 5 at $N = 2000$. Table 6 shows the average waiting time of tasks and the average blocking percentage (percentage of failed tasks) obtained by simulating each algorithm. NCS has the lowest average waiting time between all five algorithms, however the average blocking percentage indicates that almost half of the tasks are blocked. Therefore, NCS fails to balance between the two goals of minimizing the average waiting time of tasks and maximizing the system availability. Cooperative techniques show a better balance between the two goals. CS achieves the lowest average waiting time and average blocking percentage compared to the other cooperative techniques, showing that CS is the best at balancing between the performance and user experience.

To test the tolerance of CS algorithm to mobility, we chose a percentage of tasks at the end of each crucial cycle randomly and simulated their handover. The tasks can either be unscheduled tasks or the final result of a task that need to be moved from one base station to another. If the task deadline cannot tolerate the communication delay of moving the task or the results, then the task is blocked. It was found that our system can handle a handover rate of 5% of the total number of users without exceeding th_{blocking} and without needing additional CPUs.

The CS algorithm is further tested under a system with unbalanced traffic load. We divided the system into four groups of base stations with similar traffic load. Each group contains 5 base stations. The percentage of traffic allocated for each group of base stations is assumed to be $\beta = [0.1,0.2,0.3,0.4]$. The maximum number of users per BS ($n_{max}[g]$) for a group g is calculated as $N * \beta[g] * 0.21$. The minimum number of users per BS for the first group ($n_{min}[0]$) is set to 1. For other groups $n_{min}[g] = n_{max}[g-1]$. Algorithm 2 ran using $N = [500,1000,1500,2000]$ and the system links follow setting 1. Each subsystem runs with a cloud link proportional to its traffic (i.e cloud link capacity is equal to $10 \times \beta[g]$ Gbps). The number of CPUs needed by each group at different N is shown in Fig. 7. Another uniform distribution run was done to compare the output of the grouping method with the uniform distribution method. For uniform traffic load, we assume $M = 20$, $n_{min} = 5$, and $n_{max} = 0.055 \times N$, where N takes the same values as the groups. The number of CPUs needed by each group versus

Table 6 Average delay and blocking(%) per algorithm

Algorithm	Average Waiting(ms)	Average Blocking %
NCS	0.14	47.65
CS	2.37	0.4
SJFS	2.47	0.66
EDFS	2.67	0.52
Petrel [21]	3.53	6.35

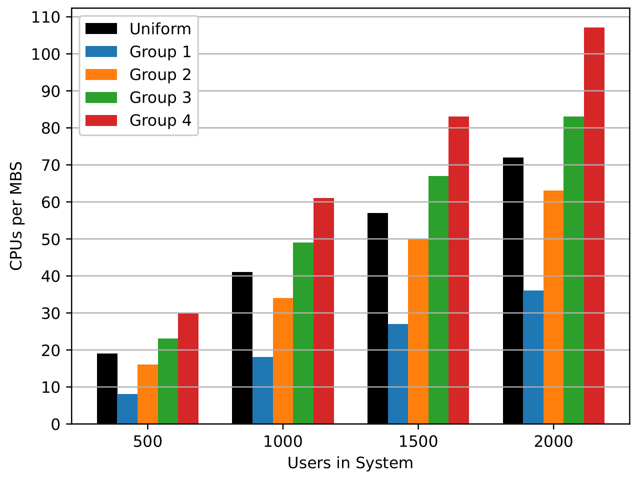

Fig. 7 Distribution of CPUs per Group

the uniform distribution at different N is shown in Fig. 7. Results show that groups 1 and 2 are below the uniform CPU distribution, while groups 3 and 4 always exceed the number of CPUs per BS needed in a uniformly distributed system. This can be explained by investigating the average number of users per BS. In the uniform distribution case, the average number of users per BS is $N/20 = 0.05N$. For the groups, the average number of users per BS is $N * \beta/5$, which corresponds to $0.02N$, $0.04N$, $0.06N$, and $0.08N$ for groups 1,2,3, and 4 respectively. It can be seen that groups 1 and 2 have average users per BS less than the uniform case and therefore need less number of CPUs per BS, while groups 3 and 4 have more average users per BS than the uniform case, so they needed more CPUs per BS.

Comparing the output of each group with the uniform distribution output gives no indication on the performance of these algorithms versus one another. Alternatively, we

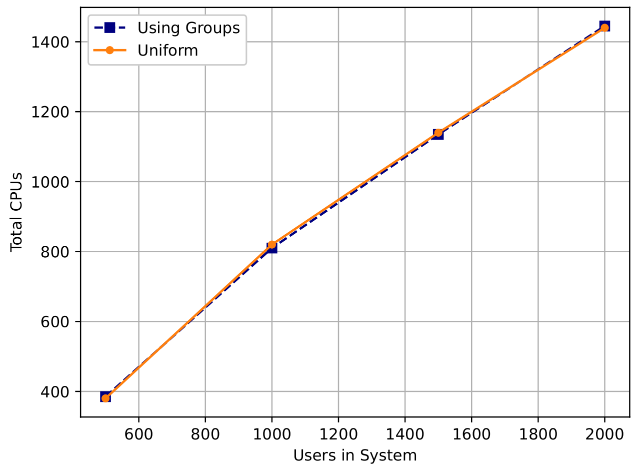

Fig. 8 Total CPUs in Grouped versus Uniform CPU Distribution

compared the total number of CPUs resulting from each algorithm to give an indication about the total cost of the systems designed using each algorithm. The total number of CPUs needed by each algorithm at different N is shown in Fig. 8. The results show that Algorithms 1 and 2 give systems with similar costs. Therefore, both algorithms have the same performance, but should be used according to the use case. If a system has a uniform or unknown load, Algorithm 1 can be used to get the number of needed CPUs per BS. If the system has a known unbalanced traffic, Algorithm 2 should be used to get the optimum CPU distribution.

7 Conclusion and Future Work

In this paper, we introduced a scheduling algorithm for a four-tier architecture system with a centralized orchestrator. Our system used a two-level cooperative technique of scheduling. Tasks are first scheduled on their local base stations until no resources are available. Unscheduled tasks are sent to the orchestrator, assigning the task to a neighbour base station or the cloud. The scheduling algorithm relies on priority queues, prioritizing tasks according to their allowed waiting times and their required throughputs in this order. The system was assessed based on the number of resources needed to reach the required service availability and the average waiting time of tasks. We suggested using a particle swarm optimization-based algorithm to determine the minimum resources needed by a given system to give acceptable performance. Results showed that the cooperative technique outperformed the non-cooperative counterpart. Moreover, the introduced scheduling algorithm minimizes the system cost and the average task waiting time compared to the shortest job first, earliest deadline first, and Petrel cooperative scheduling algorithms. We also suggested a variant of the particle swarm optimization-based algorithm to determine the ideal distribution of resources across base stations in case of an unbalanced traffic model.

In the future, a system that considers which services should be deployed on the MEC server and where to offload requests based on the cost should be explored. The current work considers a system where all services are deployed on the MEC servers. If the number of services required by the users is large, then the MEC server cannot support all services simultaneously due to its limited storage. Moreover, the specifications and offloading cost were considered to be similar for all MEC servers. However, offloading to a MEC server owned by another provider can be of higher cost than the local MEC server, and different servers can have different processing times for the same task. Therefore, the orchestrator might need a different scheduling approach to minimise offloading cost and average task delay. Another aspect to consider is the system utilization when the system is under-loaded. In under-loaded systems, a portion of the computing resources remains idle and consumes energy unnecessarily. A mechanism to utilize idle CPUs and decrease the power consumption of the system should be investigated.

Funding Open access funding provided by The Science, Technology & Innovation Funding Authority (STDF) in cooperation with The Egyptian Knowledge Bank (EKB).

Declarations

We are enclosing herewith a manuscript entitled "An Optimized Collaborative Scheduling Algorithm for Prioritized Tasks with Shared Resources in Mobile-Edge and Cloud Computing Systems" for publication in Mobile Networks and Applications Journal.
With the submission of this manuscript I would like to undertake that:
• All authors of this research paper have directly participated in the planning, execution, or analysis of this study;
• All authors of this paper have read and approved the final version submitted;
• The contents of this manuscript have not been copyrighted or published previously;
• The contents of this manuscript are not now under consideration for publication elsewhere;
• The contents of this manuscript will not be copyrighted, submitted, or published elsewhere, while acceptance by the Journal is under consideration;
• There are no directly related manuscripts or abstracts, published or unpublished, by any authors of this paper.

Conflict of Interests Manuscript title: An Optimized Collaborative Scheduling Algorithm for Prioritized Tasks with Shared Resources in Mobile-Edge and Cloud Computing Systems. The authors whose names are listed immediately below certify that they have NO affiliations with or involvement in any organization or entity with any financial interest (such as honor-aria; educational grants; participation in speakers' bureaus; membership, employment, consultancies, stock ownership, or other equity interest; and expert testimony or patent-licensing arrangements), or non-financial interest (such as personal or professional relationships, affiliations, knowledge or beliefs) in the subject matter or materials discussed in this manuscript.

References

1. Gao H, Qin X, Barroso RJD, Hussain W, Xu Y, Yin Y (2020) Collaborative learning-based industrial iot api recommendation for software-defined devices: The implicit knowledge discovery perspective. IEEE Transactions on Emerging Topics in Computational Intelligence, 1–11. https://doi.org/10.1109/TETCI.2020.3023155
2. Li R, Zhou Z, Chen X, Ling Q (2019) Resource price-aware offloading for edge-cloud collaboration: A two-timescale online control approach. IEEE Transactions on Cloud Computing, 1–1. https://doi.org/10.1109/TCC.2019.2937928
3. Gao H, Zhang Y, Miao H, Barroso RJD, Yang X (2021) Sdtioa: Modeling the timed privacy requirements of iot service composition: A user interaction perspective for automatic transformation from bpel to timed automata. Mobile Networks and Applications, 1–26
4. Lakhan A, Ahmad M, Bilal M, Jolfaei A, Mehmood RM (2021) Mobility aware blockchain enabled offloading and scheduling in vehicular fog cloud computing. IEEE Transactions on Intelligent Transportation Systems
5. Mach P, Becvar Z (2017) Mobile edge computing: A survey on architecture and computation offloading. IEEE Commun Surv Tutorials 19(3):1628–1656
6. Sun H, Yu H, Fan G (2020) Contract-based resource sharing for time effective task scheduling in fog-cloud environment. IEEE Trans Netw Serv Manag 17(2):1040–1053. https://doi.org/10.1109/TNSM.2020.2977843
7. FG-NET2030 (2020) Network 2030 architecture framework. Technical specification, ITU-T
8. Lin T, Qiu J, Fu L (2021) Online learning and resource allocation for user experience improvement in mobile edge clouds. In: ICC 2021 - IEEE international conference on communications. https://doi.org/10.1109/ICC42927.2021.9500905, pp 1–6
9. Mahmud R, Srirama SN, Ramamohanarao K, Buyya R (2019) Quality of experience (qoe)-aware placement of applications in fog computing environments. J Parallel Distrib Comput 132:190–203
10. Tuli S, Mahmud R, Tuli S, Buyya R (2019) Fogbus: A blockchain-based lightweight framework for edge and fog computing. J Syst Softw 154:22–36
11. Fadahunsi O, Maheswaran M (2019) Locality sensitive request distribution for fog and cloud servers. Serv Oriented Comput Appl 13(2):127–140
12. Ouyang T, Zhou Z, Chen X (2018) Follow me at the edge: Mobility-aware dynamic service placement for mobile edge computing. IEEE J Sel Areas Commun 36(10):2333–2345
13. Skarlat O, Nardelli M, Schulte S, Borkowski M, Leitner P (2017) Optimized IoT service placement in the fog. Serv Oriented Comput Appl 11(4):427–443
14. Wang J, Zhao L, Liu J, Kato N (2019) Smart resource allocation for mobile edge computing: A deep reinforcement learning approach. IEEE Transactions on Emerging Topics in Computing
15. Chen X, Cai Y, Li L, Zhao M, Champagne B, Hanzo L (2020) Energy-efficient resource allocation for latency-sensitive mobile edge computing. IEEE Trans Veh Technol 69(2):2246–2262. https://doi.org/10.1109/TVT.2019.2962542
16. Zhang J, Hu X, Ning Z, Ngai EC, Zhou L, Wei J, Cheng J, Hu B (2018) Energy-latency tradeoff for energy-aware offloading in mobile edge computing networks. IEEE Internet Things J 5(4):2633–2645. https://doi.org/10.1109/JIOT.2017.2786343
17. Wang S, Urgaonkar R, He T, Chan K, Zafer M, Leung KK (2016) Dynamic service placement for mobile micro-clouds with predicted future costs. IEEE Trans Parallel Distrib Syst 28(4):1002–1016
18. Bahreini T, Badri H, Grosu D (2022) Mechanisms for resource allocation and pricing in mobile edge computing systems. IEEE Trans Parallel Distrib Syst 33(3):667–682. https://doi.org/10.1109/TPDS.2021.3099731

19. Huang Y, Xu H, Gao H, Ma X, Hussain W (2021) Ssur: An approach to optimizing virtual machine allocation strategy based on user requirements for cloud data center. IEEE Trans Green Commun Netw 5(2):670–681. https://doi.org/10.1109/TGCN.2021.3067374
20. Zhao N, Liang Y, Niyato D, Pei Y, Wu M, Jiang Y (2019) Deep reinforcement learning for user association and resource allocation in heterogeneous cellular networks. IEEE Trans Wirel Commun 18 (11):5141–5152. https://doi.org/10.1109/TWC.2019.2933417
21. Lin L, Li P, Xiong J, Lin M (2018) Distributed and application-aware task scheduling in edge-clouds. In: 2018 14th international conference on mobile ad-hoc and sensor networks (MSN). IEEE, pp 165–170
22. Chunlin L, Jianhang T, Youlong L (2019) Hybrid cloud adaptive scheduling strategy for heterogeneous workloads. J Grid Comput 17(3):419–446
23. Xiang Z, Deng S, Jiang F, Gao H, Tehari J, Yin J (2020) Computing power allocation and traffic scheduling for edge service provisioning. In: 2020 IEEE international conference on Web services (ICWS). https://doi.org/10.1109/ICWS49710.2020.00058, pp 394–403
24. Ma X, Zhou A, Zhang S, Li Q, Liu AX, Wang S (2021) Dynamic task scheduling in cloud-assisted mobile edge computing. IEEE Transactions on Mobile Computing, 1–1. https://doi.org/10.1109/TMC.2021.3115262
25. Adhikari M, Mukherjee M, Srirama SN (2019) Dpto: A deadline and priority-aware task offloading in fog computing framework leveraging multilevel feedback queueing. IEEE Internet Things J 7(7):5773–5782
26. Bao W, Yuan D, Yang Z, Wang S, Li W, Zhou BB, Zomaya AY (2017) Follow me fog: Toward seamless handover timing schemes in a fog computing environment. IEEE Commun Mag 55(11):72–78. https://doi.org/10.1109/MCOM.2017.1700363
27. Zhu A, Guo S, Liu B, Ma M, Yao J, Su X (2019) Adaptive multiservice heterogeneous network selection scheme in mobile edge computing. IEEE Internet Things J 6 (4):6862–6875. https://doi.org/10.1109/JIOT.2019.2912155
28. Yin Y, Cao Z, Xu Y, Gao H, Li R, Mai Z (2020) Qos prediction for service recommendation with features learning in mobile edge computing environment. IEEE Trans Cogn Commun Netw 6 (4):1136–1145. https://doi.org/10.1109/TCCN.2020.3027681
29. Ma X, Xu H, Gao H, Bian M (2021) Real-time multiple workflow scheduling in cloud environments. IEEE Transactions on Network and Service Management, 1–1. https://doi.org/10.1109/TNSM.2021.3125395
30. Wang X, Ji Y, Zhang J, Bai L, Zhang M (2020) Joint optimization of latency and deployment cost over tdm-pon based mec-enabled cloud radio access networks. IEEE Access 8:681–696. https://doi.org/10.1109/ACCESS.2019.2959119
31. Kennedy J, Eberhart RC (1995) Particle swarm optimization. In: Perth A (ed) Proc. IEEE international conference on neural networks, pp 1942–1948
32. Kennedy J, Eberhart R (1995) Particle swarm optimization. In: Proceedings of ICNN'95 - International conference on neural networks. https://doi.org/10.1109/ICNN.1995.488968, vol 4, pp 1942–19484
33. Chopard B, Tomassini M (2018) Particle swarm optimization. Springer, Cham, pp 97–102. https://doi.org/10.1007/978-3-319-93073-2_6
34. Wang D, Tan D, Liu L (2018) Particle swarm optimization algorithm: An overview. Soft Comput 22(2):387–408
35. Sahu A, Panigrahi SK, Pattnaik S (2012) Fast convergence particle swarm optimization for functions optimization. Procedia Technol 4:319–324
36. Zhang W-J, Xie X-F, Bi D-C (2004) Handling boundary constraints for numerical optimization by particle swarm flying in periodic search space. In: Proceedings of the 2004 congress on evolutionary computation (IEEE Cat. No.04TH8753). https://doi.org/10.1109/CEC.2004.1331185, vol 2, pp 2307–23112
37. (2020) 3GPP: System architecture for the 5G System (5GS). Technical Specification (TS) 23.501, 3rd Generation Partnership Project (3GPP). Version 16.6.0. https://www.etsi.org/deliver/etsi_ts/123500_123599/123501/16.06.0060/ts_123501v160600p.pdf
38. (2021) 3GPp: Service requirements for the 5G system. Technical Specification (TS) 22.261, 3rd Generation Partnership Project (3GPP). Version 18.1.1
39. Mahmoud HHM, Amer A, Ismail T (2021) 6g: A comprehensive survey on technologies, applications, challenges, and research problems. Trans Emerging Telecommun Technol e4233:1–14. https://doi.org/10.1002/ett.4233
40. (2015) Common Public Radio Interface: Interface Specification v7.0. http://www.cpri.info/downloads/CPRI_v_7_02015-10-09.pdf
41. Liu CL, Layland JW (1973) Scheduling algorithms for multiprogramming in a hard-real-time environment. J ACM (JACM) 20(1):46–61
42. Shah SNM, Mahmood AKB, Oxley A (2009) Hybrid scheduling and dual queue scheduling. In: 2009 2nd IEEE international conference on computer science and information technology. https://doi.org/10.1109/ICCSIT.2009.5234480, pp 539–543