A Context-Aware Approach to Scheduling of Multi-Data-Source Tasks in Mobile Edge Computing

Jifeng Chen and Yang Yang(✉)

College of Computer and Information Science College of Software, Southwest University, Chongqing, People's Republic of China
chenjifeng@email.swu.edu.cn, yycia@swu.edu.cn

Abstract. The multi-data-source tasks are prevalent in the Mobile Edge Computing environment due to the distributed data storage of continuous data streams sent by various mobile devices. Many existing studies on context-aware task scheduling in MEC mainly aim to reduce energy consumption and improve performance by computation offloading among the MEC nodes. However, task context implies not only the locations of data sources but also the receivers of the result of task execution. Therefore, in this paper, we build a context-aware model for the multi-data-source tasks to analyze the objectives of task execution and data management in MEC and propose a related approach to scheduling the multi-data-source tasks. In the proposed approach, the task and environment contexts are used to determine the data sources involved, generate the task execution strategy, and resolve the target receivers. We discuss the task scheduling scheme in detail, including its architecture, metadata and data management, context-aware scheduling algorithm, and task offloading. We evaluate the feasibility and effectiveness of the proposed approach on a dataset of taxi trajectories in a city. The results illustrate that the proposed approach can effectively schedule the context-aware multi-data-source tasks in a MEC environment.

Keywords: Multi-data-source task · Context-aware scheduling · Mobile edge computing

1 Introduction

Internet of Everything is ascendant with the rapid development of mobile networks and 5G technology. Numerous sensors carried by various mobile devices, such as mobile phones, portable computers, and vehicles, send massive data via mobile internet. There are two inevitable severe challenges if all the data are sent and stored in a CDC (Cloud Data Center). First, it will put high force on network bandwidth and network stability due to the continuous data stream to the cloud data center via mobile internet. Second, the CDC will face a vast computing load when processing massive data, and then the performance of data processes is hard to be guaranteed [3,10].

H. Gao et al. (Eds.): CollaborateCom 2022, LNICST 460, pp. 193–212, 2022.
https://doi.org/10.1007/978-3-031-24383-7_11

To address the above issues, MEC (Mobile Edge Computing) arose at the right moment as a supplement to cloud computing and has a pivotal place in the 5G ecosystem. In the MEC paradigm, numerous edge servers are deployed at the edge of the mobile network [1]. On the one hand, they are placed with base stations side by side to serve for network access of mobile devices. On the other hand, since they store and process data close to the data source, they can effectively reduce the networking overhead, lower the demand for the stability of network access, and improve data processing performance by sharing the workload of centralized processing in the cloud.

In MEC scenarios, data generated and sent by mobile devices will be stored in multiple edge servers because the mobile devices will switch the connections to different base stations as they move. To reduce the load on bandwidth, edge servers usually don't transparent forward the data to the CDC in real-time. Instead, edge servers will periodically send the statistical data or the compressed data of the raw data to the CDC according to the business requirements. Consequently, the real-time tasks need to be executed on edge servers to obtain the correct results on the latest data. For example, when a vehicle is turning at a cross, the edge servers should send warnings to other vehicles to avoid traffic collision based on the velocity, direction, and driving track of the vehicles, especially for the vehicles beyond the sights of their drivers. This task is a typical multi-data-source one in MEC. It accesses the data stored in multiple distributed edge servers, locally processes them, and sends the results to targets connected with multiple base stations according to the task context. This kind of task is considerable prevalent in practice.

There has been many studies on task scheduling in MEC. For example, the various studies on task offloading focused on the number of users and computing nodes and the optimization objectives [2,7]. However, most of the studies haven't specialized on the multi-data-source tasks. Such a task should be accomplished through the cooperation of multiple edge servers and CDC in which the data are stored.

Therefore, we propose a context-aware approach to scheduling the multi-data-source tasks in MEC. The proposed approach uses the task context and the real-time load of edge servers to determine the data sources involved and generate the task execution strategy. The result of data processing will be sent to a specific range of edge servers determined by task context and then forwarded to the necessary mobile devices.

The contributions of this paper can be summarized as follows.

- We propose a context-aware model for the multi-data-source tasks to analyze the objectives of task execution and data management in MEC.
- We propose a context-aware approach to scheduling the multi-data-source tasks based on the task contexts and the real-time status of edge servers. If necessary, the computation will be offloaded among edge servers.

2 Related Works

Many distributed data storage services have been proposed for MEC in recent years. For example, Garg *et al.* [8] offered an edge time-series data storage service named TorqueDB based on ElfStore, which executes distributed queries on time-series data in edge-local storage. Zhou *et al.* [16] proposed a collaborative edge-edge data storage service called DECS, which can offload storage or computation among the MEC nodes and proactively replicate or migrate data by predicting data's popularity. Wang *et al.* [15] proposed an enhanced version of DECS to reduce the data access latency in the cold start and workload burst situations. Oyekanlu *et al.* [12] discussed that the real-time results of data processing could be obtained by deploying the computing on edge because such a way can filter most of the useless data, reduce resource consumption, and protect data privacy. In summary, the mainstream data storages are cloud-edge collaborative ones in which the data are distributed in multiple nodes, including the cloud data center, and migrated among the nodes. Hence, the multi-data-source task scheduling is of great research significance because the ubiquitous real-time tasks must be multi-data-source.

Concerning task scheduling in MEC, many types of research focus on optimizing energy consumption, execution performance, or resource utilization. For example, Saleem *et al.* [13] put forward a method to minimize the latency by D2D-enabled partial computation offloading. Huang *et al.* [9] studied on the energy-efficient offloading decision-making in vehicular networks. Bi *et al.* [4] proposed energy-optimized partial computation offloading with genetic simulated-annealing-based particle swarm optimization. Chen *et al.* [5] proposed a deep-reinforcement-learning-based context-aware online offloading strategy to reduce the overhead caused by user mobility. In general, these studies focused on generalized task scheduling, which aims to reduce energy consumption and improve performance by computation offloading among the MEC nodes. However, before assigning tasks to MEC nodes, we need to decide which MEC nodes will process their locally stored data. Moreover, we need to decide which target users will receive the result of data processing. Both the two decisions are made in the task contexts.

As for multi-data-source tasks, we also can find some related studies. For example, Chen *et al.* [6] proposed a non-cooperative game-theoretic offloading framework for multi-data-source task accomplishment. However, the MEC nodes either process data locally or fully offload computation to other nodes in this framework. There is no partial offloading allowed, which is a rather rigid restriction. Many studies on the application of MEC also involve distributed data processing. For example, Sonmez *et al.* [14] offered a machine-learning-based workload orchestrator for vehicular edge computing. Mehrabi *et al.* [11] put forward multi-tier cloudvr to Leverage edge computing for remote render of virtual reality. Existing studies on multi-data-source task scheduling have proposed some scenario-specific approaches. However, the more generic approach still needs to be developed for more scenarios.

3 System Model and Problem Formulation

3.1 A Scenario of Multi-Data-Source Task

The early warning of safe vehicle traveling is a typical context-aware multi-data-source task in MEC. In Fig. 1, all the vehicles send their status data to the connected base stations. After receiving the raw data, base stations store them on the local edge servers and send their digest as the metadata to CDC. The edge servers periodically compress and send the raw data to CDC at a relatively low frequency to reduce the bandwidth overhead. Hence, the real-time data are always held in edge servers. When performing real-time analysis, CDC assigns subtasks to base stations by the context of the analysis task. The base stations send the processing results back to CDC, and then the latter reduces all the results to determine what early warnings will be sent to which vehicles.

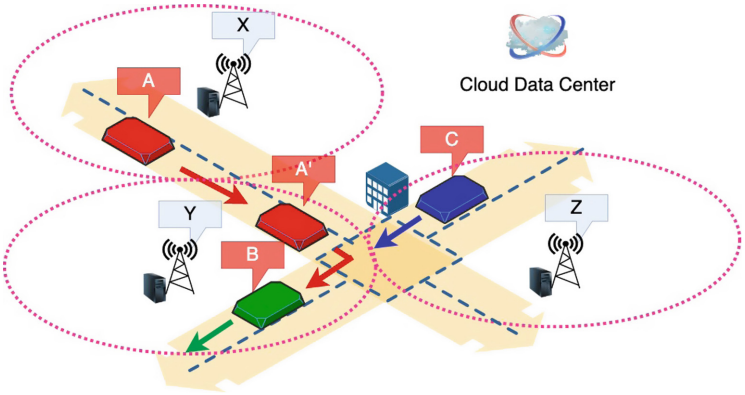

Fig. 1. A scenario of multiple data sources (Color figure online)

Car A, represented by the red flat cuboid, is traveling on the road in the direction of the red arrow. It enters the coverage area of base station Y after traveling through the coverage area of base station X. Then, car A will turn right at the cross while A and car C (represented by the blue flat cuboid and in the coverage area of base station Z) can't observe each other due to the high building at the corner. However, C must be early warned that it should take some risk avoidance measures because there is a possibility of collision with A. Meanwhile, car B(represented by the green flat cuboid) will not receive any warnings though B and C are almost the same far away from A and both B and A are in the coverage area of Y because the distance between A and B will not be shortened since B's traveling direction will not conflict with A.

Each base station with the collocated several edge servers forms a MEC node for data collecting, storing, and processing. The CDC continuously sends data analysis tasks to MEC nodes in the above scenario. The status of A, including

the speed, traveling direction, current location, destination, and the geographical positions of other cars, will be used to determine the target cars of the early warnings. Consequently, the early warnings depend on the context-aware data from multiple sources, including the status data of A in a short period held in X and Y and the locations of cars in the coverage areas of geographically adjacent MEC nodes, such as Z.

It is inevitable for some MEC nodes to be incapable of completing the assigned data analysis tasks due to their overhigh real-time load. Accordingly, they have to offload part or all of the computation to directly connected MEC nodes to ensure the tasks will be accomplished. In addition to involving multiple data sources, the scheduling of such tasks should also take the collaboration among MEC nodes into account.

It is noticeable that even if the geographically adjacent MEC nodes are interconnected, a coordinator is still necessary to coordinate the collaboration among them because the peer-to-peer pattern is inefficient for the real-time multi-data-source tasks. Consequently, the reasonable way is to reduce the results returned by data processing MEC nodes in CDC, which will send the warnings to the target MEC nodes by their geographical locations.

Besides the early warning of the safe vehicle traveling, there are many other similar scenarios of the context-aware multi-data-source tasks in the real world, such as autonomous driving and mobile service recommendation. For this reason, it is well worth designing approaches to scheduling such tasks in MEC.

3.2 System Model

Suppose the application platform is composed of a remote cloud data center and N MEC nodes like X, Y, and Z shown in Fig. 1. The MEC nodes are represented as $E = \{e_i\}, i \in [0, n]$, where n is the number of MEC nodes and e_0 represents CDC. Each MEC node connects to CDC and its adjacent nodes. Thus, it is feasible for the computation to be effectively offloaded among the MEC nodes to achieve the global load balance.

In MEC, numerous data will be uploaded by mobile users and distributedly stored in the MEC nodes connected to them along the trajectories of users. When a user establishes a connection to a MEC node and closes the connection for switching to another MEC node, the MEC node will send a real-time message to CDC, and the latter will store the message as a piece of metadata. The message is represented as a quadruple $m = (e_{id}, u_{id}, a, t)$, where e_{id} is the MEC node ID, u_{id} is the user ID, a is the action of establishing or closing the connection, and t is the timestamp.

The MEC nodes will periodically compress and upload the raw user data to CDC, but this process is transparent to users. That is, users do not need to know the exact location of data storage. Once they connect to a MEC node, they can access all the data authorized to them.

CDC will maintain the metadata to locate the storage location of the users' data in the specified time slot. Each piece of the metadata is represented as $h = (u_{id}, e_{id}, t_e, t_l)$, where u_{id} is the user ID, t_e and t_l are the start and end

timestamps of the time slot, e_{id} is the ID of the MEC node where stores the user's data generated in the time slot from t_e to t_l. When the MEC node uploads the stored data to CDC, the metadata will be updated to reflect the real-time change in the storage location. Moreover, CDC will merge the metadata by the contiguity of time slots to reduce the size of metadata. The initial values of t_e and t_l are the timestamps when the user enters and leaves the coverage area of e_{id}, but their final values are e_0, which represents CDC because the data are finally uploaded to it. Note that the storage location obtained by querying metadata is at the granularity of the MEC node or CDC. As for the specific storage mechanism, it depends on the implementation of the application platform, for example, using a time-series database to store the raw data.

3.3 Task Model

Each multi-data-source task, such as the prediction of traveling direction and speed of the cars in the safe vehicle traveling, is represented as $s = (u_{id}, f, \hat{t}_e, \hat{t}_l)$, where u_{id} is the user ID, f is the business function like prediction of traveling direction, $\hat{t}_e$ and $\hat{t}_l$ indicate the start and end timestamps of the time slot required by f.

When scheduling the task s, CDC will query the metadata to determine a set of MEC nodes $M = \{e_i\}$, $M \subset E$ which hold the data in the time slot from $\hat{t}_e$ to $\hat{t}_l$. Then, CDC assigns f to all the nodes in M, receives their return results, and reduces the results to obtain the final result. If s is a single-data-source task, its associated M contains only a single element.

The business function f will predict the mobile features of u_{id}, including the location, speed, and direction of u_{id}. CDC will send the warnings to some target users based on the prediction, just like car C receives the warning message about car A in Fig. 1. CDC determines the set of target users according to the geographical location and the mobile features of the users.

Therefore, context awareness of multi-data-source task scheduling has two aspects: on the one hand, the set of MEC nodes to which the task is assigned is determined by the context of the task. Thus, the data will be processed close to where they are generated and stored to save the network overhead and improve the processing performance. On the other hand, CDC will send the reduced result to a set of MEC nodes according to the task context. Then, the MEC nodes forward the result to the specified users.

3.4 Problem Formulation

In MEC, when CDC performs multi-data-source task scheduling, its target is the set of tasks $S = \{s_j\}, j \in [0, k]$ where k is the number of the tasks. The context-aware task scheduling has the following aspects:

Locating Data Sources and Assigning Tasks. For each s_j, suppose D_j is the data of u_{id}^j in time slot from $\hat{t}_e^j$ to $\hat{t}_l^j$. CDC will select the appropriate MEC

nodes to form M_j by querying the metadata. Assuming $d_{e_i}^j$ is the data stored in the MEC node e_i, where $e_i \in M_j$. D_j and $d_{e_i}^j$ will satisfy the following condition:

$$D_j \subset \bigcup_{e_i \in M_j} d_{e_i}^j \tag{1}$$

That is, the data stored in all e_i of M_j chosen by CDC covers the required data D_j of s_j.

Reducing and Forwarding Result. Each s_j in S will return the result of f to CDC which includes the function-specific result, such as the evaluation of environment or the available services, and the prediction of the mobile features of u_{id}^j denoted as $r_j = (u_{id}^j, l_j, v_j, d_j)$, where l_j, v_j, and d_j respectively represent the location, speed, and direction of u_{id}^j at the next moment.

Suppose $U = \{u_{id}^j\}, j \in [1, k]$ contains all the users involved in S, CDC will create the set U' which contains all the users who are geographically away from some user in U less than a given threshold of y. For all the users in $\hat{U} = U \cup U'$, CDC will create a set of user pairs $P = \{p_i\}$, where $p_i =< u_f^i, u_s^i >$ for any pair of users between which the distance is less than the dangerous threshold y_h. For each pair, u_f^i and u_s^i will receive the warnings sent by CDC, and the warnings will contain the predicted mobile features of the other party and the function-specific result of f.

Offloading Computation. When some MEC nodes are overloaded, they can partially offload the tasks to other low-load or high-performance edge nodes. Considering the overhead and delay of data transmission caused by the computation offloading, we specify that a MEC node should only offload the computation to its directly connected MEC nodes. We require just the regional load balancing but not the global load balancing of all MEC nodes.

For each M_j of s_j in S, there is a set M_j' containing all the MEC nodes directly connect to any e_i^j in M_j. Suppose $\hat{M}_j = M_j \cup M_j'$ and L_{e_i} is the load of e_i, $e_i \in \hat{M}_j$, the optimization objective of the offloading decision is formulated as:

$$min(\sum_{e_i \in \hat{M}_j} (L_{e_i} - \overline{L})^2) \tag{2}$$

where $\overline{L}$ is the average load of all the edge nodes. Namely, our objective is to minimize the fluctuation of workloads of all the MEC nodes in $\hat{M}_j$.

4 Task Scheduling Scheme

Figure 2 is the overall architecture of the proposed scheduling scheme for multi-data-source tasks, which includes three functional modules: data management, metadata management, and context-aware task scheduling. We will discuss the specific design of the functional modules in the following subsections.

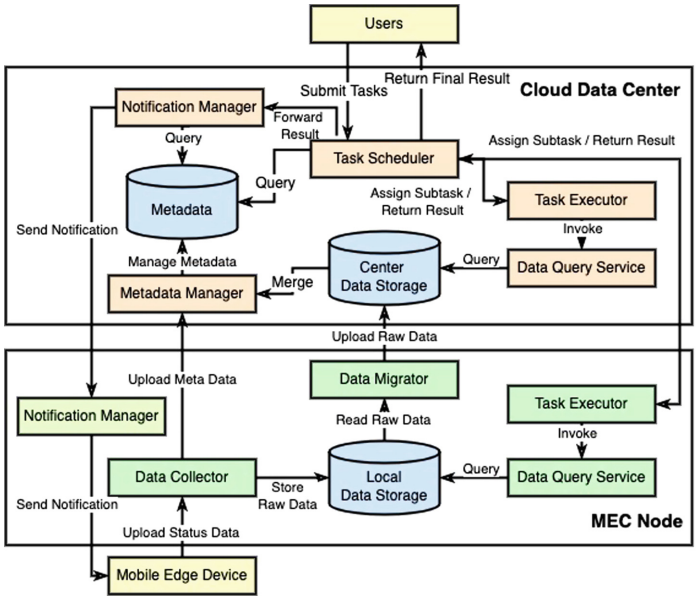

Fig. 2. The overall architecture of task scheduling scheme

4.1 Metadata Management

As mentioned in Sect. 3.2, the Data Collector of the MEC node will collect the data sent by MEDs(Mobile Edge Devices) and send two messages of $m = (e_{id}, u_{id}, a, t)$ to the Metadata Manager of CDC when u_{id} connects and disconnects with e_{id}. If a in m denotes connecting, Metadata Manager will create a new record in the metadata as $h = (u_{id}, e_{id}, t_e, t_l)$, where t_e is t in m and t_l is NULL indicating u_{id} hasn't disconnected to e_{id}. If a in m denotes disconnecting, Metadata Manager will search the records in metadata which matches u_{id} and e_{id} in m and whose t_l is NULL, then update t_l with t in m.

For example. In Fig. 3, when A enters the coverage area of X, X will send a message $m_1 = (X, A, Enter, t_1)$ to Metadata Manager. The latter will create a new record $h_1 = (A, X, t_1, NULL)$ in metadata. When A leaves X and enters the coverage area of Y, X and Y will respectively send the message $m_2 = (X, A, Leave, t_2)$ and $m_3 = (Y, A, Enter, t_2)$ to Metadata Manager, and the latter will update $h_1 = (A, X, t_1, t_2)$ and create a new record $h_2 = (A, Y, t_2, NULL)$.

When MEC nodes upload local data to the Central Data Storage of CDC, Metadata Manager will merge some records to reduce metadata size. The data uploaded by Data Migrator are divided into blocks by users. Each block contains all the data of a specific user and with a message $h = (u_{id}, e_{id}, t_e, t_l)$ in the same format as metadata. Metadata Manager will search the record exactly matching u_{id}, t_e, and t_l and update its e_{id} with e_0, indicating that these data have been migrated to CDC. Next, Metadata Manager will filter out all the records related

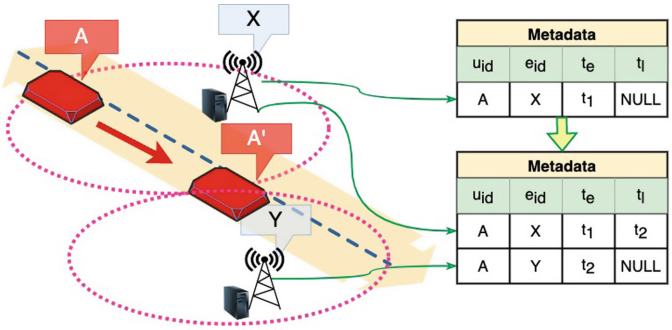

Fig. 3. An example of metadata management

to u_{id} whose e_{id} fields are e_0 and merge the records whose the time slots denoted by t_e and t_l are adjacent. For example, suppose A left the coverage area of Y at t_3, and both X and Y have uploaded A's data to CDC. Metadata Manager will merge the two records in Fig. 3 into one record as $h_1 = (A, e_0, t_1, t_3)$.

Besides the metadata of the dynamic data sent by MEDs, the static data of MEC nodes are also maintained by the Metadata Manager, including the geographical locations and the coverage area of the MEC nodes, and the connections among the MEC nodes. These data will be used in context-aware task scheduling.

4.2 Data Management

The Data Collector of the MEC node will collect the data sent by MEDs and store them in the local data storage. The Data Migrator will divide the local data into blocks by users and periodically migrate them from the local data storage of the MEC node to the central data storage of CDC. The application platform will store the migrated data in its specific database management system. The Metadata Manager will maintain the metadata of data locations in the way given in Sect. 4.1 when data are collected or migrated.

Every piece of data sent by MEDs to MEC node is denoted as $\mathbf{c} = <c_p, c_m>$, including the data specific to the application platform c_p and the mobile features c_m. c_m is denoted as (u_{id}, l, v, d) in the way given in Sect. 3.4 to represent the location, speed, and direction of u_{id}.

The local data storage of MEC nodes and the central data storage of CDC will build the 2d spatial index on users' locations l in c_m to support the search of users by the geographical distance. The positioning system of MEDs can obtain users' locations in the form of $< longitude, latitude >$. Many Database Management Systems support indexing on spatial data. For example, MySQL supports an R-tree structure index on spatial data.

4.3 Context-Aware Task Scheduling

When a user submits task $s_j = (u_{id}^j, f, \hat{t}_e^j, \hat{t}_l^j)$ to CDC, Task Scheduler will locate the set of MEC nodes M_j in which the required data D_j of s_j are stored at first. The Task Scheduler will achieve this goal by executing the following query in the metadata:

$$M_j = \Pi_{e_{id}}(\sigma_{u_{id}=u_{id}^j \wedge (t_l > \hat{t}_e^j \vee t_e < \hat{t}_l^j)}(metadata)) \tag{3}$$

Task Scheduler will select all the records which match u_{id}^j and whose t_e are earlier than $\hat{t}_l^j$ and t_l are later than $\hat{t}_l^e$. That is, the time slots of these records are fully or partly in the time period from $\hat{t}_e^j$ to $\hat{t}_l^j$. All the e_{id} projected from these records forms M_j.

For each e_i in M_j, $d_{e_i}^j$ is the subset of its stored data which is related to s_j. According to Eq. 3, the union of $d_{e_i}^j$ must cover D_j and therefore satisfy the constraint of Eq. 1.

Task Scheduler will obtain M_j' which contains all the MEC nodes directly connect to any e_i in M_j by querying the connections among MEC nodes in metadata and make decision on the policy of task execution for $\hat{M_j} = M_j \cup M_j'$. Then, Task Scheduler will send s_j along with the computation offloading decision to the MEC nodes of M_j to execute the task. We will discuss the specific algorithm of computation offloading in Subsect. 4.4.

Every MEC node and CDC has a Task Executor and a Data Query Service. The former executes the tasks sent by Task Scheduler and offloads computation to other nodes if necessary. The latter extracts required data from local data storage for the task execution. Since the function f of s_j operates on a distributed dataset, it should be a MapReduce job. Consequently, the Task Executor will execute $f.map()$ on the local dataset, and if necessary, it will offload partial computation to its directly connected nodes and execute $f.reduce()$ on the results they returned. Then, the Task Executor will send the final local result to Task Scheduler.

The special MEC node e_h in M_j is the one to which u_{id}^j is currently connected. The metadata record of this node is $(u_{id}^j, e_{id}, t, NULL)$. This node will send the locally-stored location l of u_{id}^j to the MEC nodes to which it directly connects. e_h and its adjacent nodes will find all the users who are geographically away from u_{id}^j less than a given threshold of y by searching in the local 2d spatial indexes. e_h will collect the found users in the set U_j'.

The Task Scheduler will obtain a set of the MEC nodes M_j^a in which each MEC node is geographically less than the threshold y away from and not directly connected to e_h. All the u_{id} currently connected with any nodes in M_j^a and geographically less than the threshold y away from e_h are added into U_j'. We calculate the distance between u_{id} and e_h instead of the one between u_{id} and u_{id}^j because the nodes in M_j^a can't obtain the location of u_{id}^j from e_h as they are not interconnected. That's why we set a larger threshold y at first to obtain potential targets and then use a smaller threshold y_h to filter out the final targets. Finally,

the MEC nodes will send U'_j and their mobile features in the set $C_m^{U'_j}$, then send the two sets to Task Scheduler.

After all the Task Executors of the MEC nodes in M_j return the results to the Task Scheduler, the Task Scheduler will execute $f.reduce()$ on the results. For all the tasks in S, the Task Scheduler will generate the union U' of all the sets of U'_j, and further generate $\hat{U}$ of the union of U and U'. The Task Scheduler will iterate the users in $\hat{U}$ to find all the user pairs $p_i =< u_f^i, u_s^i >$ in each of which the distance between the two users is less than the dangerous threshold y_h in the next moment according to their locations, speeds, and traveling direction in $C_m^{U'_j}$. Finally, for each pair p_i, the Task Scheduler will search the MEC nodes where u_f^i and u_s^i are currently connecting in the metadata and send them the prediction of the mobile features of u_f^i and u_s^i. These MEC nodes will forward the predictions to u_f^i and u_s^i.

When determining whether the distance between two users u_F and u_S at the next moment will be less than the dangerous distance y_h, the task scheduler will use the following way to calculate the location $hatl$ of every individual user at the next moment:

$$\hat{l} =< v * \sin\alpha * \Delta t + l.longitude, v * \cos\alpha * \Delta t + l.latitude > \tag{4}$$

In Eq. 4, α is the angle between the traveling direction d and the positive axis of X in the plane coordinate system. The user's speed v is orthogonally decomposed into horizontal and vertical components, which are used to calculate the horizontal and vertical displacements in Δt from the current time to the next moment. The future location $\hat{l}$ is obtained by adding the displacements to the current location l.

When calculating the distance $\hat{y}$ between u_f and u_s at the next moment, the Task Scheduler can delegate the calculation to the navigation application if it is available. Otherwise, the Task Scheduler will calculate the Euclidean distance between the users. If the distance between the pair of users is enlarged and exceeds the threshold y_h, they will not receive the warnings about each other even though they are connecting to the same MEC nodes. On the contrary, if the distance is less than y_h, both the users will receive the warnings even though they are connecting to different MEC nodes.

The complete context-aware task scheduling process is shown in Algorithm 1, which invokes Algorithm 2 to find the geographically adjacent users of e_h and Algorithm 3 to send warnings to the target users.

4.4 Computation Offloading

When s_j is assigned to the MEC nodes in M_j, the sizes of the data related to s_j in all nodes are different. Moreover, the nodes have different workloads and are potentially heterogeneous with each other in computing powers. Thus, it is inevitable that some nodes perhaps can not accomplish the assigned subtask of s_j on demand. Consequently, these nodes need to offload full or partial computation

Algorithm 1: Context-aware Task Scheduling

Input: Task set $S = \{s_j\}_{j=1}^{k}$, The set of MEC nodes $E = \{e_i\}_{i=0}^{n}$, Metadata *metadata*

Output: NULL

Function `TaskScheduling`(S, E, *metadata*)**:**
  $U, U' \leftarrow \{\}$;
  foreach $s_j = (u_{id}^j, f, \hat{t}_e^j, \hat{t}_{lj})$ *in* S **do**
    $U \leftarrow U \cup \{u_{id}^j\}$;
    $M_j \leftarrow \Pi_{e_{id}}(\sigma_{u_{id}=u_{id}^j \wedge (t_l > \hat{t}_e^j \vee t_e < \hat{t}_l^j)}(metadata))$;
    $M_j\prime, result \leftarrow \{\}$;
    foreach e_i *in* M_j **do**
      $M_j\prime \leftarrow M_j\prime \cup \{metadata.getDirectNeighbors(e_i)\}$;
    end
    $\hat{M}_j \leftarrow M_j \cup M_j'$;
    TaskDispatch(s_j, $\hat{M}_j$);
    foreach e_i *in* $\hat{M}_j$ **do**
      $result_i \leftarrow f.map(d_{e_i}^j)$;
      if $e_i == e_h$ **then**
        $U_j' \leftarrow$ FindAdjacentUsers(e_h);
        $result_i$.add($C_m^{U_j'}$);
      end
      $result$.add($result_i$);
    end
    $U' \leftarrow U' \cup U_j'$;
    $f.reduce(result)$;
  end
  $\hat{U} \leftarrow U \cup U'$;
  SendWarnings($\hat{U}$)
End Function

to other nodes. That's why the Task Scheduler builds $\hat{M}_j$ containing the nodes of M_j and their directly connected MEC nodes. Due to the time cost and network overhead caused by computation offloading, $\hat{M}_j$ doesn't include the indirectly connected MEC nodes.

Each MEC node e_i will send the heartbeat to CDC periodically to report its current workload L_{e_i}. Considering the heterogeneity among the nodes, we use $L_{e_i} = d_{e_i}/f_{e_i}$ to denote the workload of e_i, where f_{e_i} is the CPU frequency of e_i. We just consider the CPU frequency but not other factors, such as the memory size, bandwidth, or disk capacity, because the data processing tasks are most affected by CPU frequency, and the reasonable simplified model can reduce the computational complexity.

Task Scheduler will assign the subtasks of s_j to each MEC node e_i in M_j. If the current workload of e_i is L_{e_i}, and the extra workload brought by s_j is $\Delta L_{e_i} = d_{e_i}^j/f_{e_i}$, then the workload of e_i after it accepts the assigned subtask is

Algorithm 2: Finding Ajacent Users

Input: The MEC node e_h, The user u_{id}, Metadata *metadata*
Output: U'_j

Function `FindAdjacentUsers`(e_h,u_{id}, *metadata*):
  $U'_j \leftarrow \{\}$;
  $M^a_j \leftarrow metadata.getAjacentNeighbors(e_h)$;
  foreach u^j_{id} *in* $e_h.getUsers()$ **do**
    if *distance*(u^j_{id},u_{id}) $<y$ **then**
      $U'_j \leftarrow U'_j \cup \{u^j_{id}\}$;
    end
  end
  foreach e_i *in* M^a_j **do**
    foreach u^j_{id} *in* $e_i.getUsers()$ **do**
      if *distance*(u^j_{id},e_h) $<y$ **then**
        $U'_j \leftarrow U'_j \cup \{u^j_{id}\}$
      end
    end
  end
  return U'_j
End Function

$L'_{e_i} = L_{e_i} + \Delta L_{e_i}$. If $L_{e_i}\prime$ is not higher than the threshold L, e_i is not overloaded and it will locally process $d^j_{e_i}$. Otherwise, the Task Scheduler will choose a node directly connected to e_i in $\hat{M}_j$ to fully or partially offload computation.

We apply a simple algorithm when determining offloading targets due to the computational complexity. For each node e_i whose L'_{e_i} is greater than L, the Task Scheduler will sort the set of its directly connected MEC nodes M_i and iterate the nodes. At first, it finds the node e^0_i with the lowest workload. The $L'_{e_i} - L$ part of e_i's workload will be offloaded to e^0_i if the offloading doesn't incur the overloading of the latter. Otherwise, the part $L - L_{e^0_i}$ of e_i's workload will be offloaded to e^0_i. Then, the Task Scheduler checks the node e^1_i with the second-lowest workload in turn. The iteration proceeds until all the $L'_{e_i} - L$ part of e_i's workload is offloaded or all the nodes are iterated. The latter case means all the nodes in M_i will be overloaded, and the total computing power is not enough to accomplish the task in time. Consequently, e_i has to process the rest data by itself locally. Once the offloading decision is made, all the workloads of the nodes will be updated. Since each node only offloads the computation to its directly connected neighbors, the network overhead can be ignored in a 5G network. The nodes in M_i will send their processing results back to e_i, and the latter will perform *f.reduce* on them before sending the final result to Task Scheduler. The complete task dispatching and computation offloading process are shown in Algorithm 4.

Algorithm 3: Sending warnings

Input: The user set $\hat{U}$
Output: NULL
Function `SendWarnings`(e_h,u_{id}, $metadata$)**:**
  foreach $< u_f^i, u_s^i >$ *in* $\hat{U}$ **do**
    $\hat{y} \leftarrow distanceAtNextMoment(u_f^i, u_s^i)$;
    if $\hat{y} < y_h$ **then**
      sendWarnings($C_m^{u_f^i}, C_m^{u_s^i}, e_{u_f^i}, e_{u_s^i}$);
    end
  end
End Function

5 Evaluation

5.1 Simulation Setup

We use 2 servers with 32 cores and 128GB RAM to simulate 12 MEC nodes, each simulated with a service container and a MongoDB container. The components in MEC nodes shown in Fig. 2 are running in the service container while the data are stored in the MongoDB container. We also use a PC with 8 cores and 16 GB RAM to simulate the CDC in which we deploy the components of CDC and a MongoDB instance. Additionally, a script is deployed in the PC to simulate the vehicles sending data to the MEC nodes.

We download a data set of taxi trajectories in the city from Kaggle[1] for evaluation. This dataset contains a whole year (from 01/07/2013 to 30/06/2014) of the trajectories for all the 442 taxis running in the city of Porto in Portugal. We extracted a subset of the dataset containing all the taxi trajectories in four hours from 15:30 to 19:30 on July 1^{st}, 2013. Considering that most multi-data-source tasks are real-time tasks, we interpolate data points in the original dataset to reduce the time interval between two successive data from 15 s to 1 s. Since the original has no speed of any taxi, we calculate the taxi speed by dividing distance by time to insert them into the dataset.

The preprocessed dataset is divided into a grid with 12 cells by the latitude and longitude of the taxi, as shown in Fig. 4. The blue dots are the taxi locations, and the solid black lines are the borders of the grid cells. We suppose a MEC node is deployed in each cell to collect the status data of all the taxis in the cell.

5.2 Analysis of Data Storage and Management

We simulate the taxis sending their status data to the MEC nodes. The Fig. 5 and Fig. 6 respectively show the data counts and the data sizes of CDC and MEC nodes at 24 time points with 10 min interval from 15:30 to 19:30 on July

[1] The dataset is available at https://www.kaggle.com/crailtap/taxi-trajectory.

Algorithm 4: Task Dispatching

Input: The task s_j, The set of MEC nodes $\hat{M}_j$
Output: NULL

Function `TaskDispatch(`s_j, $\hat{M}_j$`):`
  foreach e_i *in* M_j **do**
    $\Delta L_{e_i} = d^j_{e_i} / f_{e_i}$;
    $L'_{e_i} = L_{e_i} + \Delta L_{e_i}$
    if $L'_{e_i} > L$ **then**
      $M_i \leftarrow metadata.getAjacentNeighbors(e_i)$;
      foreach e^k_i *in* M_i **do**
        if $(L_{e^k_i} + (L'_{e_i} - L)) < L$ **then**
          $offloadComputation(e_i, e^k_i, L'_{e_i} - L)$;
          $L'_{e_i} \leftarrow L$;
          $L^k_{e_i} \leftarrow L^k_{e_i} + (L'_{e_i} - L)$;
          break;
        else
          $offloadComputation(e_i, e^k_i, L - L^k_{e_i})$;
          $L'_{e_i} \leftarrow L'_{e_i} - (L - L^k_{e_i})$;
          $L^k_{e_i} \leftarrow L$;
        end
      end
    end
  end
End Function

1^{st}, 2013. The Y coordinate values in Fig. 5 are the data count, while the Y coordinate values in Fig. 6 are the number of bytes of the MongoDB blocks storing the data, and each bar in both figures is stacked by the values of CDC and MEC nodes. We set the MEC nodes to migrate their data to CDC every 25 min after the first hour.

We have the following insights about the data storage:

The total data count keeps increasing over time, and the data are migrated from MEC nodes to CDC. At the beginning of the simulation, CDC has no data, and MEC nodes begin collecting data from taxis. Later, CDC has more and more data due to the data migration from MEC nodes. Once a MEC node migrates some data to CDC, its data count will reduce. So the storage loads of all MEC nodes fluctuate in a small range.

The total data size will be reduced after merging migrated data and metadata in CDC. Since MongoDB stores data in blocks, even a single document will occupy a whole block in which some space is wasted. If the data are divided into many slices, much space will be wasted in blocks. Consequently, when CDC merges the data migrated from MEC nodes, the wasted space will be reduced due to the contiguous storage.

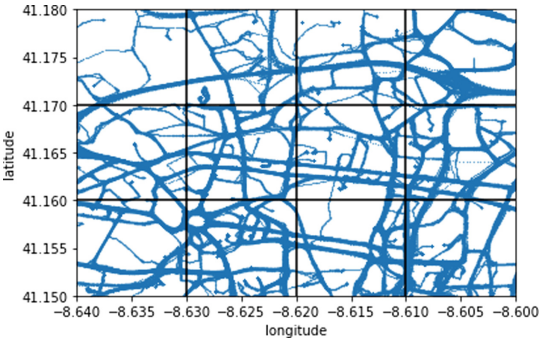

Fig. 4. The extracted dataset and its partitions

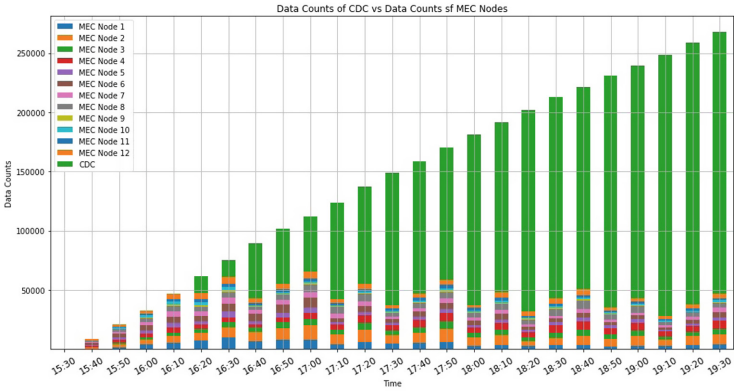

Fig. 5. Data counts of CDC and MEC nodes

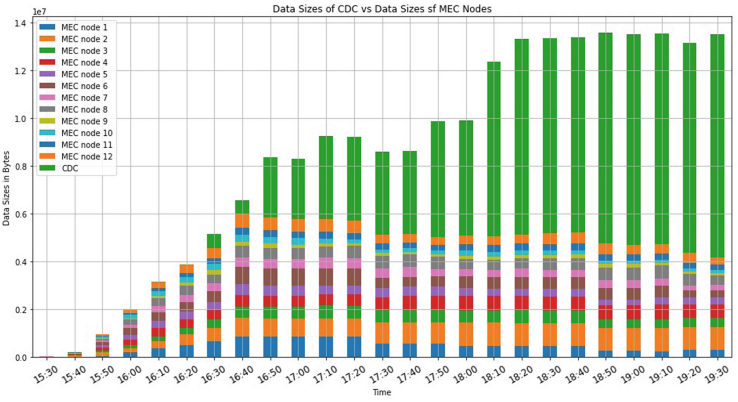

Fig. 6. Data sizes of CDC and MEC nodes

5.3 Validation of Context-Aware Task Scheduling

We use the application of early warning of safe vehicle traveling to illustrate the effectiveness and feasibility of the proposed approach to context-aware task scheduling. We choose the taxi with id 20000649 as the test target, whose location was $< -8.6253012, 41.1645144 >$ at 15:42:00 on July 1^{st}, 2013.

The Task Scheduler assigns subtasks to the MEC node to which taxi 20000649 is connecting and its adjacent MEC nodes. All the MEC nodes return the nearby taxis by initial filtering, and CDC reduces the results and selects all the taxis less than 1 KM away from the target taxi. The final filtering results are shown in Fig. 7. The red dot is taxi 20000649, and the other dots are the taxis after initial filtering, among which the blue dots are the taxis less than 1 KM away from taxi 20000649, while the green dots are the taxis beyond the range of 1KM. All the taxis are in the coverage areas of 4 different MEC nodes represented by the grid. The final set of filtered taxis is [20000499, 20000472, 20000546, 20000007, 20000077].

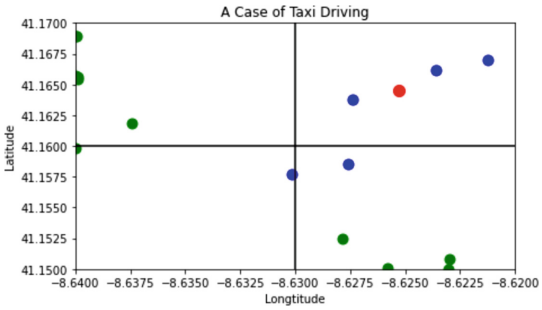

Fig. 7. A case of taxi driving

Then, Task Scheduler receives the trajectories of the filtered taxis from the MEC nodes and reduces them to obtain the complete trajectories. The first subfigure of Fig. 8 shows the trajectories of the 5 filtered taxis denoted as blue dots in Fig. 7. Each trajectory is plotted in a different color, and the black dot is the beginning end of the trajectory. The second subfigure of Fig. 8 shows the data distribution of the taxis. We can find that the trajectories of 2 taxis are respectively reduced with the results from 2 MEC nodes.

Finally, Task Scheduler calculates the distances between the target taxi and the filtered taxis every second in the next 30 s based on the predicted future trajectories and finds the destinations of the warnings. Figure 9 shows the trajectories of the target taxi and the potential destination taxis of the warnings. The blue and orange curves are the current and predicted future trajectories of the taxis, respectively. The green curve is the predicted future of the target taxi 20000649, and its beginning end is denoted by the black dot. According to

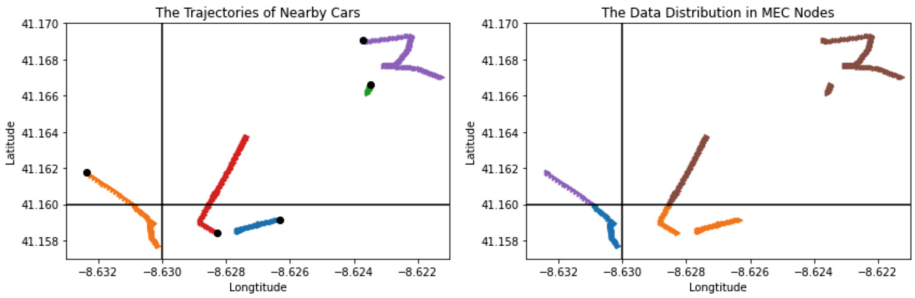

Fig. 8. Trajectories of the sample taxis

the calculated distances, only one taxi, 20000007, will be less than the dangerous threshold of 100 m away from taxi 20000649. Taxi 20000007 will be 89 m away from taxi 20000649 at 15:42:08. So taxi 20000649 and 20000007 receive the warning about each other, while other taxis, 20000499, 20000472, 20000546, and 20000077, won't receive any warnings.

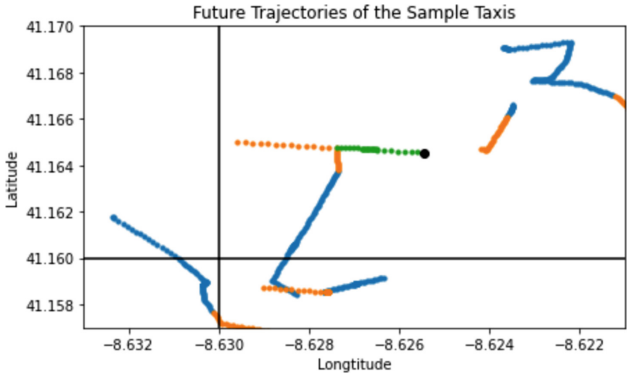

Fig. 9. Future trajectories of the sample taxis

This case demonstrates the two aspects of the proposed context-aware multi-data-source task scheduling. On the one hand, 4 MEC nodes are chosen to execute the early warning task by the task context, and CDC reduces the results of multiple MEC nodes to obtain the complete trajectories. On the other hand, the warnings are sent to the specified taxis according to the task context.

5.4 Discussion on Computation Offloading

In our simulation, all the service instances and MongoDB instances of the 12 MEC nodes are deployed in containers on 2 servers. Hence, all the MEC nodes are interconnected with each other. The data size of the extracted dataset is only

14 MiB, and we simulate they are collected by 12 MEC nodes and migrated to CDC periodically. Thus, the amount of data locally stored in each MEC node is not larger than 1 MiB, and the workload of each MEC node is always relatively low during the simulation. So no computation offloading is necessary for the simulation. However, if the dataset is large enough and a large number of tasks need to be executed, the computation offloading would happen. In our approach, if the computation is fully or partially offloaded from a MEC node to its directly connected nodes, the latter will send the task results back to the former to locally reduce the results before sending them to CDC. Consequently, it is reasonable to believe that computation offloading works in the proposed approach.

6 Conclusion

This paper analyzes the typical scenario of multi-data-source tasks in MEC and proposes a context-aware approach to scheduling this type of task. In the proposed approach, the task and environment contexts are used to locate the data sources involved, generate the task execution strategy, and resolve the target receivers of the task result. We offer the design of the task scheduling scheme, including its architecture, metadata and data management, context-aware scheduling algorithm, and task offloading. The feasibility and effectiveness of the proposed approach are evaluated on a dataset of taxi trajectories in a city.

In future, we should apply the proposed approach to more practical applications to verify its generalization in future work. Meanwhile, we should improve the performance of the proposed approach by optimizing the offloading algorithm and reducing the computational complexity of the algorithms.

References

1. Abbas, N., Zhang, Y., Taherkordi, A., Skeie, T.: Mobile edge computing: a survey. IEEE Internet Things J. **5**(1), 450–465 (2017)
2. Al-Ansi, A., Al-Ansi, A.M., Muthanna, A., Elgendy, I.A., Koucheryavy, A.: Survey on intelligence edge computing in 6G: characteristics, challenges, potential use cases, and market drivers. Future Internet **13**(5), 118 (2021)
3. de Assuncao, M.D., da Silva Veith, A., Buyya, R.: Distributed data stream processing and edge computing: a survey on resource elasticity and future directions. J. Netw. Comput. Appl. **103**, 1–17 (2018)
4. Bi, J., Yuan, H., Duanmu, S., Zhou, M., Abusorrah, A.: Energy-optimized partial computation offloading in mobile-edge computing with genetic simulated-annealing-based particle swarm optimization. IEEE Internet Things J. **8**(5), 3774–3785 (2020)
5. Chen, S., Chen, H., Ruan, J., Wang, Z.: Context-aware online offloading strategy with mobility prediction for mobile edge computing. In: 2021 International Conference on Computer Communications and Networks (ICCCN), pp. 1–9. IEEE (2021)

6. Chen, S., Sun, S., Chen, H., Ruan, J., Wang, Z.: A game theoretic approach to task offloading for multi-data-source tasks in mobile edge computing. In: 2021 IEEE International Conference on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom), pp. 776–784. IEEE (2021)
7. Cong, P., Zhou, J., Li, L., Cao, K., Wei, T., Li, K.: A survey of hierarchical energy optimization for mobile edge computing: a perspective from end devices to the cloud. ACM Comput. Surv. (CSUR) **53**(2), 1–44 (2020)
8. Garg, D., Shirolkar, P., Shukla, A., Simmhan, Y.: *TorqueDB*: distributed querying of time-series data from edge-local storage. In: Malawski, M., Rzadca, K. (eds.) Euro-Par 2020. LNCS, vol. 12247, pp. 281–295. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-57675-2_18
9. Huang, X., Xu, K., Lai, C., Chen, Q., Zhang, J.: Energy-efficient offloading decision-making for mobile edge computing in vehicular networks. EURASIP J. Wirel. Commun. Netw. **2020**(1), 1–16 (2020). https://doi.org/10.1186/s13638-020-1652-5
10. Khan, W.Z., Ahmed, E., Hakak, S., Yaqoob, I., Ahmed, A.: Edge computing: a survey. Futur. Gener. Comput. Syst. **97**, 219–235 (2019)
11. Mehrabi, A., Siekkinen, M., Kämäräinen, T., ylä-Jääski, A.: Multi-tier CloudVR: leveraging edge computing in remote rendered virtual reality. ACM Trans. Multimedia Comput. Commun. Appl. (TOMM) **17**(2), 1–24 (2021)
12. Oyekanlu, E.: Predictive edge computing for time series of industrial IoT and large scale critical infrastructure based on open-source software analytic of big data. In: 2017 IEEE International Conference on Big Data (Big Data), pp. 1663–1669. IEEE (2017)
13. Saleem, U., Liu, Y., Jangsher, S., Tao, X., Li, Y.: Latency minimization for D2D-enabled partial computation offloading in mobile edge computing. IEEE Trans. Veh. Technol. **69**(4), 4472–4486 (2020)
14. Sonmez, C., Tunca, C., Ozgovde, A., Ersoy, C.: Machine learning-based workload orchestrator for vehicular edge computing. IEEE Trans. Intell. Transp. Syst. **22**(4), 2239–2251 (2020)
15. Wang, J., Chen, H., Zhou, F., Sun, M., Huang, Z., Zhang, Z.: A-DECS: enhanced collaborative edge-edge data storage service for edge computing with adaptive prediction. Comput. Netw. **193**, 108087 (2021)
16. Zhou, F., Chen, H.: DECS: collaborative edge-edge data storage service for edge computing. In: Gao, H., Wang, X., Iqbal, M., Yin, Y., Yin, J., Gu, N. (eds.) CollaborateCom 2020. LNICST, vol. 349, pp. 373–391. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-67537-0_23