LARS: A Latency-aware and Real-time Scheduling Framework for Edge-enabled Internet of Vehicles

Shihong Hu, Guanghui Li, and Weisong Shi, Fellow, IEEE

Abstract—With the development of the Internet of Things and mobile computing, the explosive proliferation of latency-sensitive applications raises high computation demands for mobile devices. To this end, offloading computation of applications to edge-enabled Internet of Vehicles (IoV) has emerged as an effective solution. However, most of the existing studies on this issue assume that IoV can be easily formed in the practical environment, and neglect the dependency relationship between tasks of the offloading application. In this paper, we first give several observations based on the analysis results of the real traffic dataset to verify the feasibility of aggregating vehicular resources in the real world. Then, we design a Latency-aware Real-time Scheduling Framework for the edge-enabled IoV, named LARS, in which mobile users can offload applications to LARS, and the offloading tasks can be scheduled to the appropriate vehicular resources in real-time. First, we propose a clustering-based algorithm to generate Herds, which treats connected vehicles as edge computation resources to provide cooperative computing services. Second, considering the dependency relationship between tasks in the job, we present a greedy-based task scheduling algorithm for offloading jobs. The experiments on testbed include two case studies demonstrate that the superiority of the proposed scheme compared to baselines, in terms of latency and resource utilization.

Index Terms—Computation Offloading, Task Scheduling, Internet of Vehicles (IoV), Edge Computing.

1 INTRODUCTION

With the development of mobile communication technology, the rapid proliferation of high-performance intelligent terminals makes smart applications increase at a surprising rate. 4G promotes information integration and industry convergence, and intelligent applications are more widely used in industries such as finance, education, transportation, medical care, and entertainment. However, the ever-increasing demand for computing and communication of applications makes it difficult for intelligent terminals with limited resources to meet the quality of experience. In the beginning, traditional cloud computing has been widely regarded as a feasible paradigm [1]; however, the crowded transmission and long latency make it hard to meet the needs of modern applications. Fortunately, edge computing [2], [3] has emerged as a promising paradigm. In recent years, academia and industry have done a lot of research and practical applications on edge computing [4], [5], [6], [7], [8]. Undoubtedly, edge computing has efficiently solved the limitations of latency, bandwidth, energy consumption, security in traditional cloud computing [2]. However, to establish an edge computing platform for mobile users, what needs attention is that we have to deploy and maintain a tremendous number of high-cost edge servers in a large-scale area, which inevitably leads to large capital expenditures and business expenditures. Also, the intensive deployment of edge servers will lead to idle and waste of resources during off-peak hours, considering the user’s dynamic time-varying demand for resources.

The emerging concept of Internet of Vehicles (IoV) [9] is employed to collect real-time traffic conditions for transportation control systems. Remarkably, the powerful on-board computer, high-capacity storage, and more advanced communication module configuration make future vehicles more intelligent. Hence, the idea of edge-enabled IoV is to treat connected vehicles as edge computation resources to form a cooperative computing cluster and make full use of the tremendous resources that are under-utilized. Furthermore, without deploying additional servers, the concept of edge-enabled IoV can also efficiently alleviate the congestion of the network during peak time. For example, Zhang et al. [10] exploited the fog capability of parked vehicles to help the latency-sensitive computing services. Therefore, the edge-enabled IoV as a new paradigm is an excellent complement to traditional edge computing.

Although there have been a large number of studies on IoV [11], [12], [13], [14], most of them are based on the assumption that IoV can be easily formed in the practical environment, ignoring the feasibility of IoV in the real world. Given such an observation, we evaluate the feasibility of IoV formation in a realistic environment by analyzing the real-world vehicle dataset. In this paper, we consider an edge-
enabled IoV in a fixed geographical region, composed of several dynamic Herds. Besides, the mobility of vehicles and the dynamic variability of resources make when and where to form Herds in an edge-enabled IoV become a challenging issue. Therefore, we design an edge computing framework for the edge-enabled IoV, which can form Herds periodically and provide computing service for latency-aware applications dynamically. In the previous research on computation offloading of applications, most of the applications were simulated as a single task or multiple unrelated tasks [15], [16], [17]. However, in practical applications, the data relationship between tasks can not be ignored. For example, in an object recognition application, object detection can be performed after the feature extraction task is completed. Therefore, we considered the task dependencies between tasks while designing the task scheduling algorithm. Moreover, most works focus on optimizing the offload strategy itself based on the assumption that computing is offloaded directly to the edge. It is worth pondering whether offloading computing from users to the edge will achieve performance gains, such as reducing latency. In addition, applications from users are continuously generated and dynamically submitted to the edge-enabled IoV for real-time or near-real-time processing. Naturally, fast scheduling response is needed to keep up with the dynamic environment, which further exacerbates the difficulty of task scheduling in real-time systems.

Accordingly, the aforementioned observations and challenges motivate us to design an edge computing framework for IoV, named LARS, aiming to form Herds periodically to collaborate to provide computing service for applications. First, we use the realistic traffic dataset of Shenzhen city to analyze the feasibility of forming Herds in a practical environment. Then, based on the mobility and resource state of vehicles, we employ a classic clustering algorithm K-means to generate Herds, named GetHerds. In LARS, an application from the mobile user need to be partitioned into several tasks to generate a task dependency graph. Considering the characteristics of different tasks and the real environment, we design an accelerated evaluation to form an offloading job. To reduce the latency of jobs and maximize the resource utilization of Herds, we propose a Greedy-based Task Scheduling Algorithm, namely GBTSA, to schedule tasks to the appropriate workers and sort the tasks on each worker. The main contribution of this study can be summarized as follows:

- The feasibility of forming Herds in an edge-enabled IoV is verified by analyzing the real-world traffic dataset. The analysis results summarize the peak time of the day and the hot spots where vehicles gathered, which provides strong realistic conditions and a basis for the design of the LARS platform.
- In one edge-enabled IoV, according to the mobility state of vehicles such as speed, location, and trajectory, as well as the resource state such as resource amount and available time of resource, we propose a clustering algorithm GetHerds to generate Herds to coordinate to provide computing service.
- LARS, as an edge computing platform, is designed to manage the Herds in the covered region of an edge-enabled IoV and decide whether the task should be offloaded to Herds. Besides, we also propose a task scheduling algorithm called GBTSA for offloading jobs, the objective of which is to minimize the total latency of job as well as maximize the resource utilization of Herds.
- We conduct a simulation experiment based on the real traffic dataset, and the results show that Herds generated by GetHerds can maintain a stable period to provide computing service. To evaluate the performance of GBTSA, we build the testbed and conduct two case studies in a real system. The experimental results demonstrate that the proposed GBTSA is superior to baselines in terms of reducing the latency of and maximizing resource utilization.

The remainder of this paper is organized as follows. Section 2 gives a comprehensive review of related work. In Section 3, we present the observation of data analysis based on a real-world dataset and introduce the system model. Section 4 elaborates on the design of the LARS framework. The specific dynamic scheduling includes GetHerds and GBTSA are illustrated in Section 5. Section 6 provides the results of the simulation experiment, as well as the experiments on the testbed, including two case studies. Finally, Section 7 concludes our work.

2 Related work

Cyber foraging [18] coined by Satyanarayanan is a pervasive computing technique where resource-poor mobile devices offload some of their heavy work to stronger surrogate machines in the vicinity, which is the origin of computation offloading. Since then, computation offloading in cloud computing [19], [20] and mobile computing [21], [22] has started to affect the industry. In recent years, to cope with the proliferation of real-time applications, such as video analytics [23] and autonomous driving [24], computation offloading in edge computing has attracted widespread attention. As a new computing paradigm, edge computing pushes the computing from the centralized cloud to the decentralized edges, close to users, and thus effectively reduces communication latency [3] and network traffic of the core network. By deploying a large number of edge infrastructures, such as base stations (BSs), roadside units (RSUs), cloudlets [25], micro data centers (DCs), etc., to form a hierarchical computing architecture: the cloud center, the edge, and users. Therefore, offloading computation requests of latency-sensitive applications to edge-centric computing has become a hot research topic. Some studies developed the computation offloading problem as a non-linear optimization program (NLP) or a mixed-integer (MILP) program [26], [27], [28], [29]. The authors in [28] considered the energy constraints of mobile devices and dynamic network conditions to form a NLP and transformed the NLP into an equivalent integer linear program (ILP). Differently, Alameddine et al. [29] jointly combined the task offloading and scheduling problem into a MILP, based on the technique of the logic-based benders decomposition, they also designed a thoughtful decomposition to solve the problem. Particularly, many computing offloading strategies based on deep learning have emerged in the past two years.
The authors in [31] jointly considered server selection and data transmission mode. They employed a deep Q-learning approach to design an optimal offloading scheme, aiming to optimizing system utilities and improving offloading reliability. Besides, Huang et al. [32] proposed a distributed deep learning-based offloading (DDLO) algorithm to conserve energy and maintain quality of service, and they used multiple parallel DNNs to generate offloading decisions. In [33], the authors exploited the optimization problem of the content placement and content delivery in vehicular edge computing and networks. They designed a deep reinforcement learning (DRL)-based cooperative caching scheme to provide low-complexity decision making and adaptive resource management.

To compensate for the deficit resources of edge servers, edge-enabled IoV uses the existing underutilized vehicular resources in urban areas to provide computing services. It is noted that many studies use parked vehicles to enable IoVs [34], [35], [36], [37]. Li et al. [36] investigated a parked vehicular cloud paradigm, based on the contract; they designed an incentive mechanism to make parked vehicles contribute idle resources. Similarly, the basic idea in [34], [35], [37] is to aggregate potential vehicular resources in the parking lot to form a stationary cloud to provide users with computing resources. Moreover, many works have been done on investigating task offloading in mobile vehicular edge computing in recent years. To deal with the big data on IoVs in the smart city, the authors in [38] proposed a regional cooperative fog-computing-based intelligent vehicular network architecture, aiming to provide low-latency coordination services. Qiao et al. [14] presented a collaborative task offloading and output transmission mechanism for vehicular edge networks to guarantee low latency as well as application-level performance. Besides, by leveraging deep reinforcement learning, Ning et al. [17] built an intelligent offloading system for edge-enabled IoV, and they formulated the task scheduling and resource allocation problems to be a joint optimization problem. Differently, the work in [39] proposed an optimization model taking account of the communication and computing budgets as well as the failure probability. This is the first work to consider the offloading failure probability in heterogeneous vehicular edge computing. In summary, the studies on task offloading in IoV always can be formulated as an optimization problem, and there exist many approaches to address this problem, such as matching theory [40], [41], Lyapunov optimization [42], game theory [43], [44], [45], and deep learning-based [17], [46]. However, most of the existing works about task offloading focus on single task offloading or ignore the dependencies between tasks. Furthermore, the researchers put their attention on designing task offloading strategies based on the assumption that the realistic conditions of IoV are satisfied.

Particularly, to verify the feasibility of forming vehicular micro clouds (VMC) under practical road conditions, Higuchi et al. [47] firstly analyzed a realistic vehicle probe dataset of Luxembourg city, and the results showed that the connected VMCs could be formed at many locations throughout the road networks. Inspired by this work, we use the traffic dataset in Shenzhen city to analyze the feasibility of forming Herds and hot pot areas in the urban area.

The difference is that we also design the K-means-based algorithm to generate Herds, which makes the foundation for the design of the LARS framework. More importantly, to minimize the total latency of the job as well as maximize the resource utilization of Herds, the proposed task scheduling algorithm considers the task dependency of an offloading job, as well as the dynamic real-world system conditions.

3 OBSERVATION AND SYSTEM MODEL

This section presents the observation of data analysis based on a real-world dataset and introduces the system model.

3.1 Observation

Before introducing the system model, we analyze a realistic traffic dataset of Shenzhen, China [48]. The dataset covers the traffic data of more than 10000 taxis for 24h on October 22, 2013, and collects data about every 30s. First, we count the number of vehicles at each hour to analyze the peak time of the day. As shown in Fig. 1, from 24:00-4:00, the number of active vehicles gradually decreases; this is because people are in the sleep period. From 5:00, as people start a busy day, the number of vehicles increases apparently. Furthermore, we can observe that the peak time of a day is from 18:00 to 20:00 with the largest number of active vehicles in this period, as this period is the rush time for off-work and nightlife. In this case, a large number of application requests from mobile users will be generated, and the gathered resources of mobile vehicles on the road can be used to process application requests. Then, we select one period from 18:00-18:05 to count the hot pots where the vehicles gather.

![Fig. 1: The mobile vehicle distribution at different time of a day](image1)

![Fig. 2: Hot pots in the city during 18:00-18:05.](image2)
TABLE 1: Summary of Key Notation

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M$</td>
<td>Set of Herds</td>
</tr>
<tr>
<td>$J$</td>
<td>Set of offloading jobs from users</td>
</tr>
<tr>
<td>$G_j$</td>
<td>Task dependency graph</td>
</tr>
<tr>
<td>$L_j$</td>
<td>The dependency between the tasks of job $j$</td>
</tr>
<tr>
<td>$M_v(\tau)$</td>
<td>Mobility state of vehicle $v$ at time slot $\tau$</td>
</tr>
<tr>
<td>$R_v(\tau)$</td>
<td>Resource state of vehicle $v$ at time slot $\tau$</td>
</tr>
<tr>
<td>$\theta_v$</td>
<td>Available time of resource in vehicle $v$</td>
</tr>
<tr>
<td>$I_j$</td>
<td>Set of tasks that can/cannot be processed in parallel</td>
</tr>
<tr>
<td>$T_{local}$</td>
<td>Local computation latency of task $j$</td>
</tr>
<tr>
<td>$w_i$</td>
<td>Workload of task $i$</td>
</tr>
<tr>
<td>$p_j$</td>
<td>Local processor speed of user $j$</td>
</tr>
<tr>
<td>$l$</td>
<td>Round trip time overhead</td>
</tr>
<tr>
<td>$N_{worker}(m)$</td>
<td>Number of workers in $m$</td>
</tr>
<tr>
<td>$T_p$</td>
<td>Tolerant latency of job $j$</td>
</tr>
<tr>
<td>$N_{worker}(j, m)$</td>
<td>Number of workers in $m$ occupied by job $j$</td>
</tr>
<tr>
<td>$T_{start}(t_i)$</td>
<td>Ideal start time of task $t_i$</td>
</tr>
<tr>
<td>$T_{send}(t_i)$</td>
<td>Time of sending the results from task $t_i$ to task $t_i$</td>
</tr>
<tr>
<td>$T_{pro}$</td>
<td>Processing time of task $t_i$</td>
</tr>
</tbody>
</table>

TABLE 2: Number of vehicles in different hot points.

<table>
<thead>
<tr>
<th>Hot point</th>
<th>Location(latitude, longitude)</th>
<th>Number of vehicles</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>(114.143951, 22.554733)</td>
<td>212</td>
</tr>
<tr>
<td>B</td>
<td>(114.062286, 22.531733)</td>
<td>195</td>
</tr>
<tr>
<td>C</td>
<td>(114.017915, 22.544110)</td>
<td>169</td>
</tr>
<tr>
<td>D</td>
<td>(114.064072, 22.531475)</td>
<td>162</td>
</tr>
<tr>
<td>E</td>
<td>(114.059547, 22.526449)</td>
<td>178</td>
</tr>
<tr>
<td>F</td>
<td>(114.278999, 22.724501)</td>
<td>181</td>
</tr>
</tbody>
</table>

Fig. 3: The distance between vehicles.

As a result, how to coordinate so many vehicles in the edge-enabled IoV to form Herds is challenging, and we will introduce the proposed method of forming Herds in Section 5.

3.2 System model

We consider an edge-enabled IoV in this paper, as shown in Fig. 4. All vehicles equipped with powerful edge servers in the communication range of controller (BS or RSU) form a vehicular network, and several different vehicles aggregate into different Herds in one edge-enabled IoV. The controller is responsible for vehicle information collection, resource coordination, and task assignment in its edge-enabled IoV. During the peak period, once the computing request from the mobile user approaches the processing peak of the controller, the controller will periodically trigger the vehicles in the edge-enabled IoV to form several Herds to provide computing resources for mobile users. The details for how to form Herds will be illustrated in Section 5.
to get high QoS. We use $J$ denote the set of offloading jobs from users, and a certain job can be formulated as $J_j = \{ t_{\text{arr}}^j, G_j \}$, where $t_{\text{arr}}^j$ and $G_j$ represent the arrival time and task dependency graph, respectively. Moreover, $G_j = (T_j, L_j)$ is modeled as a directed acyclic graph [51], [52], [53], $T_j = \{ t_1, t_2, ..., \}$ is the task set of job $j$, and $L_j$ is the dependency between the tasks. For example, $l_{k,i} \in L_j$ means that the input data of task $t_i$ is based on the results of task $t_k$; that is, $t_i$ is the precursor of $t_k$. We use $pr (t_i)$ denote the set of precursors of task $t_i$, and $su (t_i)$ represents the set of successors of $t_i$. Fig. 5(a) shows a directed acyclic graph example for a job $j$. Job $j$ contains six tasks, $T_j = \{ t_1, t_2, ..., t_6 \}$. From Fig. 5(a), we know that the precursor of $t_6$ is $pr (t_6) = \{ t_3, t_4, t_5 \}$ and the successor of $t_1$ is $su (t_1) = \{ t_3, t_4 \}$. More specifically, the task graph of Fig. 5(a) represents a traffic monitoring job. The video frames collected by roadside surveillance cameras are input into $t_1$ for motion detection. At the same time, the images are input into $t_2$ for preprocessing. Then, the results of $t_2$ will be sent to $t_3$ for pedestrian detection and $t_4$ for vehicle recognition. And, the results of $t_2$ will be sent to $t_5$ for plate detection. Finally, the results of $t_3$, $t_4$, and $t_5$ will be aggregated into $t_6$ for result generation. Besides, we also consider more complicated task graphs, the alternative structure. For example, as shown in Fig. 5(b), $t_3$ and $t_5$ are mutually parallel optional tasks, which represent “or” nodes and are marked in blue. Only one “or” node of the same color can be selected. Thus, in this task graph, the input of last task $t_6$ has two choice, one is $t_3$ and $t_4$, the other is $t_4$ and $t_5$.

Fig. 5: Different structures of task graph.

4 LARS Framework Design

LARS is designed as an edge computing framework for edge-enabled IoVs. LARS periodically forms Herds, which provide computing service for latency-aware applications dynamically. The main components are applications generated by mobile users and Herds composed of mobile vehicles. At the peak time, when the computing demands from users are close to the controller’s load, the controller will periodically schedule the vehicles within the range of the edge-enabled IoV to form several Herds, thereby cooperating to provide computing services for mobile users. The proposed architecture of the LARS framework is shown in Fig. 6.

4.1 Controller

As shown in Fig. 7, the controller will collect the mobility state and resource state information of all vehicles periodically. Mobility state information includes the location, the current speed, and the trajectory based on the GPS navigation system. Resource state information informs the available resource amount and the available time of the vehicle. Especially, the resource amount is represented as the number of workers in this study. Each vehicle that enters or gets out the covered region of the edge-enabled IoV will send a register or released message to the controller. Based on the collected information, the controller decides how to generate Herds and estimates the available virtualized resource pool.

4.2 Application

In LARS, as shown in Fig. 6, an application from the mobile user need to be partitioned as several tasks to generate a task dependency graph. The techniques of application partition are beyond the scope of this study. An accelerated evaluation needs to be performed before task offloading due to the characteristics of different tasks and the real environment. Then, the offloading controller will submit the job composed of a set of tasks to the framework by user API. According to the information of the offloading job, the job assigner first selects the most appropriate Herd. Then, the task scheduler calls the resource service through the internal APIs to give the optimal task provisioning message, as shown in Fig. 6 and Fig. 8. In particular, if the Herd resource becomes unavailable before the task is completed, the task will be migrated. In this regard, the controller of LARS must call another Herd to take over the task. The problem of task migration is beyond the scope of this paper.

5 Dynamic Scheduling

In this section, we give the detailed algorithm of generating Herds, problem formulation, and the task scheduling scheme.

5.1 Herds generator

The available resources and available time of each vehicle are very different. To coordinate vehicle resources to provide services to users, forming a Herd as a cluster with a similar amount of resources and similar available time can maximize resource utilization and reduce task failure and migration rates. In this study, we use the classic clustering algorithm K-means to generate Herds. Algorithm 1 describes the procedure of the specific process.
The proposed GetHerds aims to generate Herd clusters in the edge-enabled IoV every interval time \( \tau \). Note that the input information includes the mobility state \( M_v(\tau) = \langle L_v, S_v, \rho_v \rangle \) and available resource state \( R_v(\tau) = \langle C_v, \theta_v \rangle \) of all vehicles in the edge-enabled IoV. For Procedure HerdGenerator, firstly, each vehicle forms an information array \( F_v \) consisting of the location \( L_v \), the amount of computing resource \( C_v \), and the available time of resource \( \theta_v \) (Lines 2-4). Then, the vehicle data matrix \( D_R \) composed of the information array of all vehicles has been obtained (Line 5). Based on vehicle data matrix \( D_R \), Procedure HerdGenerator calls for Procedure K-meansBasedCluster to get the final Herd at time \( \tau \) (Line 6). Specially, the location \( L_v \) of vehicle \( v \) needs to call for Procedure UpdateVehicleState to get the real-time location information (Lines 8-11). Procedure K-meansBasedCluster first randomly selects one cluster center and then select other \( k-1 \) centers based on the maximum distance among vehicles (Lines 13-21). Finally, Procedure K-meansBasedCluster uses the classic cluster algorithm k-means to generate Herds (Lines 22-23).

5.2 Offloading controller

Before task offloading, we designed an acceleration evaluator to evaluate whether it is necessary to offload each task in the task set \( T \) after the application partition. Computing acceleration refers to the speedup in latency of processing a task. Fig. 5 shows two instances of task graph, and we know acceleration refers to the speedup in latency of processing a task in the task set \( T \) after the application partition. Computing acceleration refers to the speedup in latency of processing a task.

Fig. 6: The architecture of LARS framework.

Fig. 7: Herd generator.

Fig. 8: Request process.
that some tasks have no dependencies between each other can be processed in parallel, while some tasks need to wait for all its dependent tasks to complete before they can be processed. For example, tasks 3, 4 and 5 can be processed in parallel, and task 6 can be processed after all tasks have been processed. Therefore, based on these two cases, we give two ways to evaluate the task’s computing acceleration, which are expressed as (1) and (2).

\[ C_{acc}^i = \frac{T_{local}^i}{T_{offload}^i} = \frac{w_i/p_j}{d_i/r_j + w_i/p_0 + l}, \]  

\[ C_{acc}^{I'} = \frac{T_{local}^{I'}}{T_{offload}^{I'}} = \frac{\sum_{m \in I'} w_i/N_{worker}^{local} p_j}{d_i/r_j + \max_{j \in I', m \in M} w_i/N_{worker}^{m} p_0 + l}, \]  

where \( C_{acc} \) is defined as the computing acceleration of a single task and \( C_{acc}^{I'} \) represents the computing acceleration of parallel processed tasks. For Eq. (1), \( w_i \) denotes the workload of task \( i \), \( d_i \) is the input data size of task \( i \), \( p_j \) represents the local processor speed of user \( j \), \( p_0 \) stands for the processor speed of vehicles, \( r_j \) is the transmission rate from user \( j \) to the controller, and \( l \) represents the round trip time overhead. For Eq. (2), \( I' \) is the set of tasks that can be processed in parallel, \( N_{worker}^{local} \) is the number of workers locally, \( N_{worker}^{m} \) denotes the number of workers in \( Herd \ m \) and \( M \) is defined as the set of \( Herds \). Apparently, \( C_{acc} > 1 \) and \( C_{acc}^{I'} > 1 \) mean that the latency of local computation is greater than the total latency of the computation offloaded to the control center; that is, the computing acceleration speedup can be obtained by offloading. The offloading latency includes the transmission delay, the task processing delay, and the round trip time overhead. The acceleration evaluator will evaluate each task in the task graph of the application according to Eq. (1) and Eq. (2). Then the offloading controller will make a decision and form an offloading job \( T_{offload}^{ij} = \{t_1, t_2, ..., t_n\} \) based on the result of the acceleration evaluator.

5.3 Job assigner

In an edge-enabled IoV, the controller manages several \( Herds \). The differences in computing resources and available time of different \( Herds \) make the controller need to be weighed in assigning a job to the appropriate \( Herd \). In this study, we assume that each job seeks to find the \( Herd \) with the most computing acceleration and the lowest cost. The maximum computing acceleration speedup is defined as \( C_{acc}^{I',Max} \) and the total cost of leased computing resources is denoted as \( T_{tole}^{ij} F_m \), where \( T_{tole}^{ij} \) represents the tolerant latency of job \( j \) and \( F_m \) denotes the unit price of computation of \( Herd \ m \). The trade-off of the two objectives is investigated, and the joint objective is defined as follows:

\[ P1: \max \alpha C_{acc}^{I',Max} + \frac{\beta}{T_{tole}^{ij} F_m}, \forall m \in M, \]  

\[ s.t. \; N_{worker}^{j,m} = \frac{T_{tole}^{ij} - \max(T_{local}^{ij}, \sum_{i \in I', p_0})}{\max_{i \in I'} w_i/p_0}, \]  

\[ \alpha, \beta \in [0, 1], \]  

\[ N_{worker}^{j,m} \leq N_{worker}'^{j,m}, \forall m \in M, \]  

\[ T_{tole}^{ij} \leq \theta, \forall m \in M, \]  

where \( \alpha, \beta \in [0, 1] \) are two scalar weights. The constraint (3a) defines the number of \( workers \) of \( Herd \ m \) \( N_{worker}^{j,m} \) occupied by \( job \ j \), and the constraint (3b) indicates that \( N_{worker}^{j,m} \) cannot exceed the total number of \( workers \) in \( Herd \ m \). The constraint (3c) implies the available time \( \theta \) of \( Herd \ m \) should be bigger than the tolerant latency \( T_{tole}^{ij} \) of \( job \ j \). 

Proof: See the Appendix.

5.4 Task scheduling

5.4.1 Problem formulation

For a job \( J_j = \{T_{arr}^{j}, G_j\} \), \( T_{arr}^{j} \) is the arrival time of \( j \) and \( G_j \) is the task dependency graph of \( job \ j \). Each task \( t_i \in G_j \) needs to wait until all tasks in its precursors have been completed and sent the results to it before starting running.

**Definition 1.** The ideal start time of task \( t_i \) is defined as:

\[ T_{start}^i = \max_{t_k \in pr(t_i)} (T_{end}^k + T_{send}^i(k,i)), \]  

where \( T_{end}^k \) denotes the end time of task \( t_k \), \( T_{send}^i(k,i) \) indicates the time of sending the results from task \( t_k \) to task \( t_i \). The end time of \( job \ j \) is determined by the maximal finish time of its all tasks, which can be defined as:

\[ T_{end}(J_j) = \max_{t_i \in P_j}(T_{end}^i(t_i)), \]  

where \( T_{end}(J_j) \) represents the end time of \( job \ j \). For mobile users, the objective is to minimize the total completion time of an offloading \( job \ j \), thus obtaining near-real-time results. The total completion time of \( job \ j \) is the difference between the end time and the arrival time, which is expressed as:

\[ \min T_{end}(J_j) - T_{arr}^{j}. \]

For the computing service providers, they seek to maximize resource utilization to reduce energy-related consumption. For a \( Herd \), its resource utilization refers to the actual average number of served workers.

5.4.2 Theoretical analysis

It is impossible to obtain optimal solutions because the problem of task scheduling in dynamic environments is NP-hard [54], [55]. Two key steps for task scheduling on the LARS framework, one is to map tasks to the appropriate worker, and the other is to perform a reasonable execution order of tasks on the same worker, thereby minimizing the delay of jobs and maximizing the resource utilization of \( Herds \). In this study, refer to [56], we apply the minimum earliest start time first rule (MESTF) to sort the tasks on the same worker.

**Theorem 1.** Assume all the tasks in task set \( T = \{t_1, t_2, ..., t_k\} \) are parallel and their ideal start time satisfy that \( T_{start}^{1} \leq T_{start}^{2} \leq ... \leq T_{start}^{k} \), the total completion time of task set \( T = \{t_1, t_2, ..., t_k\} \) on the same worker can be minimized by the MESTF rule.

**Proof:** See the Appendix.

Fig. 9 gives an example of Theorem 1, the ideal start time of tasks in task set \( T = \{t_1, t_2, t_3\} \) is \( T_{start}^{1} = 0, T_{start}^{2} = 20 \) and \( T_{start}^{3} = 35 \), respectively. Besides, assume the processing time of tasks is \( T_{local}^1 = 15, T_{local}^2 = 10 \) and \( T_{local}^3 = 10 \). If we apply the MESTF rule to sort these tasks in worker 1,
as shown in Fig. 9(a), the processing order is $t_1 \rightarrow t_2 \rightarrow t_3$, and the total completion time is $45s$. However, if we change the order of $t_2$ and $t_3$, the total completion time is $55s$, as shown in Fig. 9(b). As a result, we can know that the processing order by the MESTF rule can effectively reduce the completion time of tasks on the same worker.

**Theorem 2.** Let $pr(t_i) = \{t_1, t_2, ..., t_m\}$ and $T^{\text{end}} + T^{\text{send}}(l_{1,i}) \geq T^{\text{end}}(l_{2,i}) \geq ... \geq T^{\text{end}}(l_{m,i})$, task $t_i$ start time can be minimized as $T^{\text{star}}(t_i) = \min_{1 \leq i \leq k} \{ MT(l_{1,i}) + T^{\text{end}}(l_{i+1}) \}$, where $MT(l_{j,i}) = \min_{t_j \in \mathcal{T}} \{ T^{\text{end}}(t_j) \}$ refers to the maximal completion time of the task set \{\$t_1, t_2, ..., t_i\}$ based on MESTF rule.

**Proof:** See the Appendix.

**Corollary.** According to Theorem 2, we can obtain the following corollary: some tasks in the precursors $pr(t_i)$ of $t_i$ may not be in the same worker. If there is a task $t_m$ satisfies $T^{\text{send}}(l_{m,i}) > T^{\text{pre}},$ duplicating the task to the worker where $t_m$ is located can be minimized the total completion time of $t_i$.

As illustrated in Fig. 10(a), the sample task graph consists of five tasks \{\$t_1, t_2, t_3, t_4, t_5\}. For task $t_4$, its precursors $pr(t_4) = \{t_1, t_2, t_3\}$. The time of sending results from precursors to the task is written on the arrow between them, and the processing time is also marked around the tasks. For example, the processing time of task $t_3$ is $T^{\text{pre}} = 5$ and the time sending the result from task $t_3$ to task $t_4$ is $T^{\text{send}}(l_{3,4}) = 15$. From Fig. 10(b), we know that task $t_4$ has to wait for all its precursors have been processed and all the results have been sent to it. Since it takes time for the result of task $t_3$ to be sent from worker 1 to worker 2, the start time of task $t_4$ is delayed to 40s. As shown in Fig. 10(c), we duplicate the task $t_3$ to worker 2 and the start time of task $t_4$ can be advanced from 40s to 30s.

**Definition 2.** The effective end time of task $t_i$ is defined as:

$$T^{\text{end}}(t_i) = \begin{cases} \min_{t_k \in \text{su}(t_i)} \left( \max_{t_k \in \text{pr}(t_i)} \right. T^{\text{end}}(t_k) - T^{\text{send}}(l_{q,i}), & \text{su}(t_i) \neq \varnothing \vphantom{3} \vphantom{3}, (7) \\ 0, & \text{su}(t_i) = \varnothing. \end{cases}$$

**Theorem 3.** If task $t_i$ satisfies $T^{\text{end}} \leq T^{\text{end}}$, then the end time $T^{\text{end}}$ of $t_i$ has no effect on its successors’ start.

**Proof:** See the Appendix.

To explain Theorem 3 effectively, we give an instance, as shown in Fig. 11. From Fig. 11, we know task $t_1$ and $t_2$ have been processed, and the end time is 60s and 80s, respectively. Task $t_3$ is unscheduled and its successors are $t_4$ and $t_5$, the time of sending results from $t_3$ to $t_4$ and $t_5$ is 15s and 40s. Based on Eq. 7, the effective end time of $t_3$ can be calculated as $T^{\text{end}} = \min \{ t_k \in \text{pr}(t_4), t_k \in \text{pr}(t_5) \} \left( \max \left( T^{\text{end}}(l_{4,3}) \right) - T^{\text{send}}(l_{3,4}) \right) = \min \{ 60 - 15, 80 - 40 \} = 40$. If the task $t_3$ is processed and the actual end time $T^{\text{end}} \leq 40$, which means the task $t_3$ has no effect on its successor task $t_4$’s start time.

**Fig. 9:** An example of Theorem 1.

**Fig. 10:** An example of Corollary.

**Fig. 11:** An example of Theorem 3.
Algorithm 2 GBTSA
1: WorkerS ← Get all the available workers in Herd m
2: WaitQ ← ∅; ExecuS ← ∅
3: Procedure Preparation
4: while a new job arrives do
5:   Add all the tasks of the new job into WaitQ
6:   while WaitQ! = ∅ do
7:     ProT = \{ti | pr(t_i) = ∅ ∪ pr(t_i) ⊆ ExecuS\}
8:     for each task t_i ∈ ProT do
9:       if t_i^opt ∉ ∅ or t_i^opt ∉ ExecuS then
10:          [selWorker, reTaskS] = Schedule(t_i)
11:          ExecuS(selWorker) ← ExecuS ∪ reTaskS ∪ \{t_i\}
12:       else
13:          delete t_i from WaitQ
14:       end if
15:     end for
16:   end while
17: end Procedure
18: Procedure Schedule(t_i)
19: selWorker ← NULL; reTaskS ← ∅; MinT ← ∞
20: Compute T^end(t_i) of task t_i
21: for each worker w_k ∈ WorkerS do
22:   tempRT ← ∅
23:   Use MESTF rule and get T^end(t_i) of task t_i on worker w_k
24: while true do
25:   if T^end(t_i) < MinT then
26:      selWorker = w_k;
27:      MinT = max(T^end(t_i), T^end(t_k));
28:      reTaskS ← tempRT;
29:      if T^end(t_i) < T^end(t_k) then
30:        break;
31:      end if
32:   end if
33: end if
34: St ← argmax{t_k ∈ pr(t_i)}(T^end(t_k) + T^end(l_{k,i}))
35: if St! = NULL & St ∉ tempRT & St ∉ ExecuS(w_k) then
36:   tempRT ← tempRT ∪ \{St\}
37: Assuming that all the tasks in tempRT are re-executed on w_k
38: Update T^end(t_i) of task t_i
39: else
40: break;
41: end if
42: end while
43: end Procedure
44: EndProcedure

the optimal worker selWorker and the set of tasks reTaskS associated with it that need to be duplicated. Finally, the task and its reTaskS will be added into ExecuS(selWorker) of selected worker (Lines 8-10). Procedure Schedule gives a detailed process of task scheduling based on Theorem 1, Theorem 2 and Theorem 3. Note that selWorker denotes the selected worker for task t_i, reTaskS is used to record the tasks needed to be re-executed, and MinT stands for the minimal end time of task t_i. Firstly, Procedure Schedule computes the effective end time of task t_i (Line 18). Then, Procedure Schedule tries to schedule task t_i to a worker satisfying T^end(t_i) < T^end(t_k) (Lines 25-26), which greatly reduces the probability of task migration. If it is infeasible, the worker with the minimal end time of task t_i will be recorded (Lines 21-24). Using the advantage of Theorem 2, Procedure Schedule strives to advance the start time of task t_i by re-executed some tasks in its precursors (Lines 16-35). Besides, the selected tasks are added into tempRT and the iteration until the start time of task t_i can no longer be advanced.

6 EXPERIMENTS AND EVALUATION

In this section, we aim to evaluate the performance of two proposed algorithms, including GetHerds and GBTSA. First, we carry on a simulation experiment for the proposed GetHerds. Second, we introduce the experimental hardware and environment setup and present evaluation results under two study cases for GBTSA.

![Fig. 12: The snapshot of vehicles at time 18:00.](image)

Table 3: The results of 10 generated Herds. Acronyms used in this Table: Average Distance (AD), Differences of Speed (DoS), Differences of Resource Amount (DoRA), Differences of Available Time (DoAT).

<table>
<thead>
<tr>
<th>Herd</th>
<th>AD</th>
<th>DoS</th>
<th>DoRA</th>
<th>DoAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>297</td>
<td>3.8678</td>
<td>1.1662</td>
<td>7.9095</td>
</tr>
<tr>
<td>2</td>
<td>286</td>
<td>3.2619</td>
<td>0.8944</td>
<td>6.2097</td>
</tr>
<tr>
<td>3</td>
<td>231</td>
<td>3.7202</td>
<td>1.1662</td>
<td>5.831</td>
</tr>
<tr>
<td>4</td>
<td>255</td>
<td>2.0591</td>
<td>0.8944</td>
<td>10.4881</td>
</tr>
<tr>
<td>5</td>
<td>272</td>
<td>4.4091</td>
<td>0.4899</td>
<td>6.7646</td>
</tr>
<tr>
<td>6</td>
<td>210</td>
<td>3.9699</td>
<td>1.0198</td>
<td>6.5605</td>
</tr>
<tr>
<td>7</td>
<td>283</td>
<td>2.7568</td>
<td>0.4899</td>
<td>8.7316</td>
</tr>
<tr>
<td>8</td>
<td>266</td>
<td>2.4166</td>
<td>0.9798</td>
<td>6.5238</td>
</tr>
<tr>
<td>9</td>
<td>247</td>
<td>2.5612</td>
<td>0.4899</td>
<td>6.8348</td>
</tr>
<tr>
<td>10</td>
<td>295</td>
<td>4.4452</td>
<td>0.7483</td>
<td>8.0895</td>
</tr>
</tbody>
</table>

6.1 Simulation Results

In this simulation, we use the real traffic dataset of Shenzhen city introduced in Section 3. We select one point (114.143951, 22.554733) as the controller location, and the communication between controller and vehicle is assumed to be LTE. Within a 2km covered region of the controller, all the snapshot locations of vehicles, as well as speed data at time 18:00 on October 22, 2013, are used for simulation, as shown in...
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2021.3106260, IEEE Transactions on Services Computing

Table 4: The average results of different number of Herds. Acronyms used in this Table: Average Distance (AD), Average Differences of Speed (ADoS), Average Differences of Resource Amount (ADoRA), Average Differences of Available Time (ADoAT)

<table>
<thead>
<tr>
<th>Number of Herds</th>
<th>AD</th>
<th>ADoS</th>
<th>ADoRA</th>
<th>ADoAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>264</td>
<td>3.3467</td>
<td>0.8224</td>
<td>7.5743</td>
</tr>
<tr>
<td>15</td>
<td>252</td>
<td>2.8122</td>
<td>0.6421</td>
<td>6.9856</td>
</tr>
<tr>
<td>20</td>
<td>250</td>
<td>2.7723</td>
<td>0.6678</td>
<td>7.5342</td>
</tr>
<tr>
<td>30</td>
<td>244</td>
<td>2.4124</td>
<td>0.7012</td>
<td>6.3251</td>
</tr>
<tr>
<td>40</td>
<td>239</td>
<td>2.4781</td>
<td>0.5908</td>
<td>6.6624</td>
</tr>
<tr>
<td>50</td>
<td>232</td>
<td>2.2912</td>
<td>0.6186</td>
<td>5.9929</td>
</tr>
</tbody>
</table>

Fig. 12. In this paper, the resource state of vehicles is randomly generated, the range of computing resource amount $C_v$ is [4, 6], and the range of available time of resource $AP_v$ is [3, 5] min. To evaluate the stability and cohesiveness of generating Herds by GetHerds, we use Euclidean distance to evaluate the state similarity of all vehicles in each Herd in terms of speed, resource amount, and available time. Table 3 shows the results of the selected 10 generated GetHerds. From Table 3 and Table 4, we can see the average distance between vehicles of each Herd is all below 300m, which satisfies the V2V communication condition. The speed of the real vehicle at peak time is distributed between 20km/h and 40km/h. The third column of Table 3 shows that the speed difference of one Herd is small and Table 4 indicates that the same observation, and most of the Herds are around 3. Besides, the difference between resource amount and the available time of most Herds are below 1 and 10, respectively, which are shown in the fourth and fifth columns of Table 3 and Table 4. In summary, the mobility and resource state of each generated Herds are similar and can maintain a stable period to provide computing service.

Table 5: Hardware Setup

<table>
<thead>
<tr>
<th>Platform</th>
<th>DESKTOP</th>
<th>LAPTOP</th>
<th>RaspberryPi 4B</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Inter i7-4770</td>
<td>Inter i7-5600U</td>
<td>ARM Cortex-A72</td>
</tr>
<tr>
<td>Socket</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Architecture</td>
<td>x86,64</td>
<td>x86,64</td>
<td>ARM</td>
</tr>
<tr>
<td>Memory</td>
<td>16</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>Disk</td>
<td>1TB SSD</td>
<td>256G SSD</td>
<td>64G SD</td>
</tr>
</tbody>
</table>

6.2.2 Baselines

We compare the performance of GBTSA against the following three baselines:

- **Local**: it means all the tasks of jobs are processed locally.
- **Random**: the controller will select a worker of Herd randomly to process the offloaded task.
- **Shortest Transmission Time First (STTF)** [57]: it means that the controller tends to schedule tasks on the worker that has the shortest estimated latency to transfer the tasks. The controller maintains a table to record the latency of transmitting data to each available worker.
- **Shortest queue Length First (SQLF)** [57]: it means that the controller tends to schedule tasks to the worker which has the least number of tasks queued upon the time of query.

Fig. 13: LARS system implementation.

Fig. 14: Different task dependency graph: Sequential structure (SS) or Alternative structure (AS).

6.2.3 Case Study 1: Scene Understanding

The scene understanding application can help users grasp the information of the target environment to make a reasonable judgment. We use ADE20K dataset [58] provided by MIT for scene understanding experiments. We assume that four or five scene understanding tasks form a job, and the dependency of these tasks is predefined in advance. We
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2021.3106260, IEEE Transactions on Services Computing

design two types of task graphs for jobs: sequential structure and alternative structure to impose the task dependency for jobs. Fig. 14 shows the designed task dependency graphs. A job with an alternative task structure means that the task chain to complete the job is optional. As shown in Fig. 14(c) and (d), the blue and green circle nodes are “or” nodes, which means that only one “or” node of the same color can be selected. For example, in Fig. 14(c), we can select \( t_1 \) and \( t_4 \) or \( t_3 \) and \( t_5 \) for a job. Particularly, jobs are generated according to Poisson distribution with the arrival rate \( \lambda \), and the task dependency graph of each job is randomly selected from Fig. 14.

Fig. 15: Case 1: Response time vs. 20 different jobs.

Application side: From the application side, the objective of the proposed GBTSA is to minimize the total latency of an offloading job. In this experiment, the jobs generated by users follow the Poisson distribution, and the arrival rate \( \lambda \) is randomly generated from [2, 5]. To show the superiority of the proposed GBTSA, we conduct comparative experiments with baselines. First, we compare the response time of jobs under different schemes. Fig. 15 shows the results of 20 jobs under five schemes: the proposed GBTSA, local computing, random scheduling, STTF and SQLF. From Fig. 15, in general, we can observe that the response time of jobs under GBTSA is the lowest, followed by SQLF and STTF, then random scheduling, and the response time of jobs is the highest when computed locally. Apparently, the limited computing resources extend the local latency of jobs. Besides, STTF and SQLF perform similarly; overall, SQLF works better than STTF. The STTF scheme tends to schedule a task to the worker with the least transmitting time. Sometimes a lower total latency will be achieved, but it may cause a worker to saturate and increase the response time. Also, the SQLF scheme has a similar problem. Scheduling a task to the worker with the shortest queue without considering the transmission latency may lead to an excessive delay in transmission and increase the response time. We can see that the response time of job 6 and job 20 is abnormally high, and this is because the randomness of scheduling may schedule some tasks to busy workers and thus extending the waiting time in queue. In contrast, the total latency of each job under GBTSA remains stable.

Generally, the response time of a job consists of transmitting time, scheduling time in the controller, waiting time in queue, and processing time on a worker. To investigate which latency affects the total latency the most, we counted the different latency of each job. As shown in Fig. 16(a), for GBTSA, the transmitting time and processing time account for the largest latency and waiting time also occupies a small part. The scheduling time in the control center is very short. It is worth noting that the transmitting time is a bottleneck factor that causes the total delay, and it will be the focus of our future research work. Fig. 16(b) shows the different latency results of 10 jobs under random scheduling; apparently, the processing time of jobs is the largest part of the total latency. This is because improper scheduling makes task processing time longer, as well as waiting time. Fig. 16(c) and Fig. 16(d) show the different latency results of 10 jobs under STTF and SQLF, respectively. Like the previous analysis, STTF has an advantage in transmitting time compared to SQLF, but SQLF is superior in waiting and processing time. In general, observing the range of the y-axis of Fig. 16, we can find that the average latency of GBTSA is smaller than the other three scheduling schemes.

Fig. 16: Case 1: Different latency vs. 10 different jobs.

Herd side: From the Herd side, the objective is to maximize resource utilization. We use the average served workers to evaluate the resource utilization of different scheduling schemes. From Fig. 17, we can observe that under GBTSA, no matter how many jobs, the number of served workers is almost 4. This means that almost all workers participate in the processing of the job. Followed
by SQLF, the average number of served workers is almost 3, which indicates that the SQLF scheme has a relatively good performance in resource utilization. However, the average number of served workers of STTF and random schemes is less than 3, which means only no more than 3 workers process the job, resulting in low resource utilization. The randomness of the random scheme and the single tendency of the STTF scheme for the shortest transmitting time make them less realizable in terms of resource utilization.

6.2.4 Case study 2: Driving Behavior Detection

The second case study is driving behavior detection, where security monitors in a traffic sensing application use captured vehicular images to detect dangerous behavior. For example, a surveillance camera at the intersection captures the driver on the phone while driving, and then alerts the dangers signal to traffic police. We use Berkeley DeepDrive dataset [59] for driving behavior detection experiments. Similar to case study 1, four or five tasks are assumed to be one job, and the task dependency graph of each job is randomly selected from Fig. 14. Besides, the job arrival process also obeys the Poisson distribution with the arrival rate $\lambda$.

Fig. 18: Case 2: Response time vs. 20 different jobs.

![Graph](image1)

Fig. 19: Case 2: Different latency results of 10 jobs under four scheduling schemes.

(a) GBTSA
(b) Random
(c) STTF
(d) SQLF

Fig. 20: Case 2: Average workers vs. different number of jobs.

**Herd side:** Similarly, we employ resource utilization to compare the performance of these four scheduling schemes. As shown in Fig. 20, we can see that the average served workers of GBTSA are almost 4 under the different number of jobs. Followed by SQLF, the average number of served workers is bigger than 3. In this case study, the gap between STTF and SQLF is not very large, and the average number of served workers under STTF remains at 3. However, for random scheduling, the number of served workers is less than 3. As a result, fewer served workers result in lower resource utilization. In conclusion, the proposed GBTSA scheme can keep high resource utilization of Herd.

7 Conclusion

In this paper, according to observations based on the analysis results of the real traffic dataset in Shenzhen, we verify the feasibility of forming Herds in an edge-enabled IoV. To better manage the computing resources in the vehicular network and provide computing services effectively, we design a dynamic scheduling framework LARS. In LARS, a K-means-based clustering algorithm GetHerds is proposed to generate Herds. Particularly, the task of an application...
should be evaluated to determine whether it can benefit from offloading to Herds. For offloading jobs, the job assigner in the controller first selects the appropriate Herd, and then a greedy-based task scheduling algorithm GBTSA is used to schedule tasks to the appropriate workers and sort the tasks on each worker. The simulation experiment based on the real traffic data shows that Herds generated by GetHerds can maintain a stable period to provide computing service. The experiments on the testbed include two case studies that show that GBTSA has a good performance in reducing the total latency of jobs, as well as maximizing the resource utilization of the edge-enabled IoV. The experimental results from a testbed reveal that the communication time is a bottleneck factor resulting in the total latency of jobs. Based on this observation, we will build an experimental environment based on 5G communications to make computation offloading of applications to the IoV more realistic.

ACKNOWLEDGMENTS

This work was supported in part by the scholarship from China Scholarship Council, in part by the National Natural Science Foundation of China (No. 62072216), Jiangsu Agriculture Science and Technology Innovation Fund (No. CX(19)3087), Wuxi International Science and Technology Research and Development Cooperative Project (No. CZE02H1706), the 111 Project (B12018).

REFERENCES


Shihong Hu received the bachelor’s degree in communication engineering from Jiangnan University in 2016. She is a PhD candidate of the School of Artificial Intelligence and Computer, Jiangnan University. She had been a Visiting Scholar in Prof. Weisong Shi’s MIST Lab for research on resource scheduling in edge computing. She has published over 70 papers in journal or conferences. Her research interests include wireless sensor networks and edge computing.

Guanghui Li received the Ph.D. degree from the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China, in 2005. He is currently a Professor with the School of Artificial Intelligence and Computer, Jiangnan University, Wuxi, China. He has published over 70 papers in journal or conferences. His research interests include wireless sensor networks, fault tolerant computing, and nondestructive testing and evaluation. His research was supported by the National Foundation of China, Zhejiang Provincial Science and Technology Foundation, and other governmental and industrial agencies.

Weisong Shi received the B.S. degree from Xidian University, Xi’an, China, in 1995, and the Ph.D. degree from the Chinese Academy of Sciences, in 2000, both in computer engineering. Weisong Shi is a Charles H. Gershenson Distinguished Faculty Fellow and a Professor of Computer Science with Wayne State University, USA, where he directs the Mobile and Internet Systems Laboratory (MIST) and Connected Autonomous Driving Laboratory (CAR). He is an IEEE Fellow and an ACM Distinguished Scientist.