AVE: Autonomous Vehicular Edge Computing Framework with ACO-Based Scheduling

Jingyun Feng, *Student Member, IEEE*, Zhi Liu, *Member, IEEE*, Celimuge Wu, *Member, IEEE*, and Yusheng Ji, *Member, IEEE*

***Abstract*—With the emergence of in-vehicle applications, providing the required computational capabilities is becoming a crucial problem. This paperproposes a framework named *autonomous vehicular edge* (AVE) for edge computing on the road, with the aim of increasing the computational capabilities of vehicles in a decentralized manner. By managing the idle computational resources on vehicles and using them efficiently, the proposed AVE framework can provide computation services in dynamic vehicular environments without requiring particular infrastructures to be deployed. Specifically, this paper introduces a *workflow* to support the autonomous organization of vehicular edges. Efficient job caching is proposed to better schedule jobs based on the information collected on neighboring vehicles, including GPS information. A scheduling algorithm based on *ant colony optimization* is designed to solve this job assignment problem. Extensive simulations are conducted, and the simulation results demonstrate the superiority of this approach over competing schemes in typical urban and highway scenarios.**

***Index Terms*—Ant colony optimization, cloud computing, edge computing, job scheduling, vehicular network.**

I. Introduction

With the development of smart vehicles, an increasing number of computer applications are being used in vehicular environments. This presents a new challenge for vehicles, i.e., how to satisfy the computational requirements of these applications. Upgrading the on-board computers is one option, but the cost is high. Cloud computing [1] is another possible solution, in which networked remote servers are used to satisfy the computational requirements. However, traditional centralized cloud computing suffers from long latency and unstable connections in vehicular environments. This greatly degrades the quality of the experience. For example, Augmented Reality (AR) can provide helpful information and warnings through a *heads-up display* (HUD) in vehicles [2] or can provide a better field of view [3]. However, AR has high computational requirements, which are typically beyond the computational capability of a single vehicle; moreover, because AR also requires swift processing, the direct use of traditional cloud services for AR on the road is unrealistic because of the high latency. Other examples of computationally intensive applications that may be useful in vehicular environments include speech recognition and natural language processing, which are expected to become more widely utilized in applications for assisting drivers and passengers.

Edge computing [4] is a more suitable solution for enhancing computational capabilities in vehicular environments. Communication between vehicles can be realized based on *dedicated short-range communications* (DSRC) or device-to-device communication [5], which has enabled great improvements in communication quality. Moreover, vehicles are not running computationally intensive applications at all times. Therefore, a low-latency edge cloud environment to support these applications can be made possible by efficiently managing the vehicles' resources. Several related studies have been conducted with regard to cloud computing systems for vehicular environments, such as *Datacenter at the Airport* [6], *Pics-on-wheels* [7] and *Cloud Transportation System (CTS)* [8]. However, these schemes have various limitations, such as only being usable in static scenarios, being dedicated to specific applications, or requiring infrastructure-based centralized controls. The questions of how to efficiently utilize vehicles' resources in highly dynamic vehicular environments and how to provide a universal framework for all applications are non-trivial and have not been formally studied, to the best of the authors' knowledge.

To solve the aforementioned problems, we propose an *Autonomous Vehicular Edge* (AVE) framework for edge computing on the road. In this framework, we introduce a *workflow* to support the autonomous organization of a vehicular cloud. Efficient job caching is proposed to better schedule jobs based on the information collected on neighboring vehicles, including GPS information. A scheduling algorithm based on Ant Colony Optimization (ACO) [9] is designed to solve the assignment problem. Extensive simulations are conducted using simulated traffic and an *IEEE* 802.11p network. The main contributions of this work are as follows:

1) We propose a modular architecture for AVE to support generalized task offloading. In our framework, the requirements imposed on the applications are kept minimal.
2) We design procedures to allow offloading and scheduling without the need for centralized control, so they can be applied with a minimal deployment cost.

Manuscript received October 21, 2016; revised March 27, 2017; accepted May 22, 2017. Date of publication June 12, 2017; date of current version December 14, 2017. This work was supported in part by China Scholarship Council no. 201206340036, and in part by JSPS Kakenhi Grants 16H02817 and 15K21599. The review of this paper was coordinated by the Guest Editors of the Connected Vehicle Series. *(Corresponding author: Yusheng Ji.)*

J. Feng and Y. Ji are with the National Institute of Informatics and SOKENDAI (the Graduate University for Advanced Studies), Tokyo 101-8430, Japan (e-mail: fjy@nii.ac.jp; kei@nii.ac.jp).

Z. Liu is with Shizuoka University, Hamamatsu 432-8011, Japan (e-mail: liu@ieee.org).

C. Wu is with The University of Electro-Communications, Tokyo 182-8585, Japan (e-mail: clmg@is.uec.ac.jp).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TVT.2017.2714704

TABLE I
VEHICLE APPLICATIONS

	Examples	Priority	Offload
Crucial Applications (CAs)	Vehicle control, system monitoring, accident prevention	Highest	No
High-Priority Applications (HPAs)	Navigation, information services,optional safety applications	High	Yes
Low-Priority Applications (LPAs)	Multimedia, passenger entertainment	Medium to Low	Yes

3) We further propose an ACO-based scheduling algorithm to solve the *NP-hard* job-assignment problem with fast convergence, such that over 90 percent of the optimal results are achieved within 10 iterations.
4) Finally, we conduct extensive evaluations using real traffic data for both highway and urban scenarios. Our simulation results demonstrate the proposed framework's advantages over existing schemes.

The remainder of this paper is organized as follows. Section II analyzes the motivation for this framework and presents an overview of proposed job offloading methodologies. Section III defines the components of the proposed framework. Section IV introduces the workflow and describes the job assignment problem. Section V explains the ACO-based scheduling algorithm for solving the assignment problem. The framework is then evaluated through simulations, as described in Section VI. Section VII introduces previous related studies, and we conclude this paper in Section VIII.

II. PROBLEM ANALYSIS

A. Motivation: Vehicle Applications

Vehicle applications can be classified into three levels according to their characteristics: *crucial applications* (CAs), high-priority applications (HPAs) and low-priority applications (LPAs), as shown in Table I. CAs are core applications of the vehicle system or safety-related applications. Because of their importance to the vehicle and the passengers, CAs have the highest priority and must be flawlessly executed without relying on unstable connections in the vehicular environment. Moreover, because CAs are generally developed by the vehicle manufacturer, the vehicle's on-board system should always be designed to have sufficient–if not abundant–capacity to meet their resource demands. Hence, CAs are considered to be executed completely outside of AVE.

The remaining applications are divided into high-priority applications (HPAs) and low-priority applications (LPAs) depending on their purposes. HPAs include driving-related applications and optional safety-enhancing applications, e.g., routing and information services for drivers. These applications are important but not obligatory, which means that failures and delays are allowed, although inconvenience will be caused to the driver. Typical HPAs include heads-up displays (HUDs) [2], field-of-view enhancement [3], and road sensing [10]. An increasing number of new cars are equipped with such applications. To accommodate these emerging HPA services, manufacturers are designing some computational capacity margins into their on-board systems.

LPAs are the class of applications that are of lesser importance to drivers and passengers. For example, speech recognition [11] is an interesting application that allows a driver to issue various commands without being distracted from driving. This concept has been adopted in Apple's CarPlay [12], whose implementation relies on an attached smart phone and cloud computing via the cellular network. Other multimedia applications, such as video processing [13], have also been proposed for vehicles. With the emerging trend of self-driving vehicles, users are becoming able to shift their focus from driving to other activities, such as entertainment. Cloud-based video games would provide a better travel experience for passengers or a free driver. With the further development of smart vehicles, an increasing number of HPAs and LPAs will emerge. Note that HPAs and LPAs are interchangeable; i.e., HPAs can be LPAs for some vehicles, and LPAs can be HPAs for some vehicles.

The inherent problem is how to meet the ever-increasing computational demands posed by LPAs and HPAs in vehicles. Instead of replacing equipment or even the entire vehicle, a solution based on the efficient utilization of the existing computational resources can ease the burden of the deployment of new vehicular applications.

B. Methodology: Cloud and Edge Computing

By offloading the local workload to a remote cloud, the owner of a device can be freed from the need to upgrade and maintain additional hardware [14]. This concept was quickly transplanted into the mobile environment, where it is also known as mobile cloud computing (MCC) [15]. MCC provides a good solution for mobile devices, which typically exhibit poor performance in terms of computational capability and require a high cost to upgrade. However, in the mobile scenario, offloading faces challenges such as unstable network connections, high latency and network data charges. Some of these problems are even more severe in vehicular environments. The mobility of vehicles is considerably higher than that of hand-held devices, which significantly reduces the link durations with fixed-point infrastructures such as roadside units (RSUs). In addition, unlike hand-held devices, which mostly roam in urban areas, vehicles are often in rural areas, where infrastructures are rarely deployed and Internet connections to the remote cloud are not as good as those in urban areas. Because of these limitations, cloud computing is difficult to use in vehicular environments.

Mobile edge computing (MEC)[16] is another option for addressing these issues. This concept involves placing computational resources at the logical extremes of the network, e.g., access points, RSUs, base stations, or even users' devices, instead of relying only on a centralized cloud. In this way, MEC enables applications to respond in real time. The main difference that distinguishes cloud and edge computing from other methodologies is the ability to "virtualize" resources, meaning that resources are provided not in physical forms (such as CPUs

and HDDs) but rather in a per-time and per-use manner. These methodologies allow more flexible services to be provided with lower upgrade and upkeep costs to consumers.

We target to realize the local "cloud" idea in vehicular environment, based on the following properties:

1) *Shareable resources:* Apart from the computational resources used by CAs, some resources remain available for HPAs/LPAs. When these applications are not running (e.g., navigation applications become idle after completing the computation of routes), this subset of resources is in an idle state and can be shared.
2) *Movement pattern:* Although vehicles have a high speed relative to stationary infrastructures, with which their link durations are short, the speed difference between two vehicles driving in the same direction on the same road is much smaller [17]; hence, a longer link duration between such vehicles becomes possible.
3) *Energy:* A formidable challenge in MEC is the limited battery life of mobile devices. Assisting others is not "free" because it consumes the host device's energy. By contrast, a vehicle has much larger battery storage, and the on-board system's energy cost is trivial compared with the total energy consumed during driving.
4) *Incentives:* Engaging in such a resource-sharing framework is beneficial to both service providers and users. Service providers can reduce their infrastructure deployment costs. Users can potentially enjoy low-latency computational services at the cost of nothing but idle resources. Moreover, service providers may even pay users for their shared resources.

Challenges exist with regard to framework design, the method of self-organization, and scheduling in the distributed environment. Studies have previously been conducted on resource sharing in vehicular environments, such as [18]. However, to the best of the authors' knowledge, no solution has been proposed for general applications, nor one that would operate in a completely distributed manner.

C. Offloading and Virtualization

Offloading is the process of migrating workloads to other machines for processing; we define these workloads as *jobs* in this paper. "Job" in this paper is a generic term; no detailed characteristics are presumed except for those stated in Section III-C. Offloading is one fundamental aspect of the proposed framework. It can be performed at various levels, such as functions, tasks, applications and virtual machines (VMs) [19]. A considerable amount of work has been performed on offloading itself. One typical proposal is *CloneCloud* [20], which focuses on how to split the execution of workloads and move them to another machine. This paper borrows this idea and will not reinvent the wheel. Rather, we will focus on processor discovery and job scheduling. Although the workload concept is generalized in this paper, the following factors are considered:

1) *Size of the transmitted data:* Since the network bandwidth is limited in a vehicular environment, the amount of data to be transmitted is one critical factor to the performance.

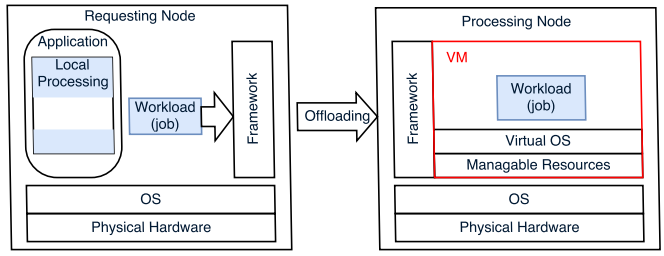

Fig. 1. Illustration of the offloading process. The requesting node splits a workload and submits it to our framework. The framework offloads the job to a processing node, whose framework sets up a VM to process it.

2) *Host requirements:* Some jobs require specific software or capable hardware. In traditional cloud computing, this is not a problem because cloud servers can always satisfy these requirements. However, these requirements may not be satisfied for some nodes.
3) *Importance:* Not all jobs are of equal benefit to users. Some jobs (e.g., jobs from HPAs) have higher priority, and the greatest effort should be made to complete them.

Virtualization is the abstraction of computational resources, typically as virtual machines (VMs) with associated storage and networking connectivity. For better virtualization (i.e., better utilization of the available computational resources), the processing of an offloaded job is performed in a VM on a processing node. This VM has two functionalities: first, it includes the required support for the workload, e.g., operating system (OS) and runtime libraries; second, it virtualizes or manages the remaining resources, excluding those occupied by CAs, so that the execution of CAs is isolated from that of LPAs/HPAs and the priority of the CAs can be guaranteed. The virtualized resources of this VM are what will be scheduled using the framework proposed in this paper. The aforementioned offloading process is illustrated in Fig. 1.

III. System Overview

A. Roles of Nodes

We refer to each vehicle participating in the framework as a *node*. Each job is generated by one node and processed by one node. These two nodes can be identical; i.e., that node that generates the job may also process the job. In this case, we say that the job is processed *locally*. If the node that generated the job cannot directly reach the node that is helping to process the job, then other nodes will be needed to help forward the data between them. For convenience, these three types of nodes are called *requesters*, *processors* and *forwarders*, respectively, in this paper. Note that this classification is not permanent and only applies for a particular job. A vehicle may be the processor for one job while also being the requester for another job that it does not have sufficient capacity to process.

B. Architecture of Proposed Framework

Like other client-server architectures [20], [21], the software in this framework includes client-side applications and

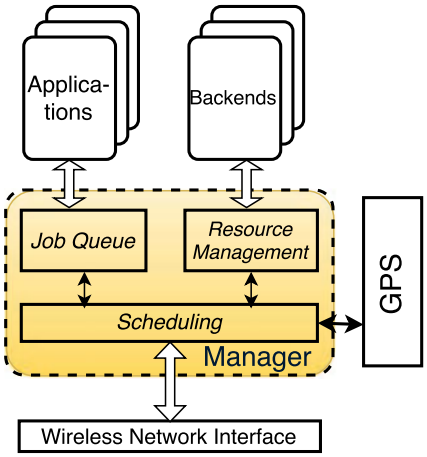

Fig. 2. Illustration of the architecture of proposed framework.

corresponding server-side applications, referred to as *application* modules and *backend* modules, as shown in Fig. 2. *Applications* are run on the native operating system, which manages their priorities and available resources; *backends* are run in a VM, which is managed by the AVE framework. We introduce a *manager* module (highlighted in Fig. 2) as a middleware to collect information, jobs (as explained in Section III-C) and results from the other two modules and to make the offloading and job assignment decisions.

Components of this framework are deployed on each vehicle before that vehicle begins traveling on the road. The *manager* module is pre-installed on each vehicle, preferably by the on-board system vendor. *Applications* are chosen and installed by users. A typical means of doing so is to install the applications from the Internet; however, how and in what way the applications are installed is beyond the scope of this paper. *Backends* can be installed together with the corresponding applications (e.g., a user may want an application that can be run *locally* on the vehicle) or deployed independently (e.g., companies may deploy backends on users' vehicles to provide services on the road).

The *manager* module consists of three submodules: a *job queue* module, a *resource management* module and a *scheduling* module. The *job queue* module provides a job offloading interface for applications, collects jobs and places them in the scheduler when the requirements (discussed in Section IV) are satisfied. The *resource management* module controls the available resources that the *backends* can use. Note that CA tasks such as safety, monitoring, and emergency reporting are not offloaded and are processed locally with the highest priority. Apart from the resources occupied by CAs, the remaining resources are available to the *backends*. The *scheduling* module is the core of the manager. The *scheduling* module is responsible for communicating with other nodes, making assignment decisions, sending jobs and receiving results. In the proposed framework, a *Global Positioning System* (GPS) device is also required for the scheduling module to obtain current geographical position and velocity of the vehicle. This requirement is trivial since *GPS* devices are widely installed.

The network established in our framework uses *dedicated short-range communication* (DSRC) standards, such as *IEEE* 802.11p. This type of communication enables rapid link setup without the establishment of a basic service set. This reduces the overhead incurred for sending data to other vehicles.

C. Jobs

Each offloaded workload is called a *job*. Offloaded workloads may occur at various levels, such as functions, applications and virtual machines. To accord with this diversity, jobs are considered here as generalized entities. However, for the modeling of real-life workloads for scheduling, the following job characteristics are assumed in our context:

1) *Context-free:* This is the basic requirement for a job to be offloaded and executed on another machine. The resource files on which jobs rely are either distributed in the *backends* before offloading or regarded as part of the jobs' data and sent during the offloading period. This characteristic implies dependency of jobs should be handled within the *applications*, before offloading to *manager* module. The implementation that the *applications* use to separate the jobs is out of this paper's scope. But we suggest that: For jobs that depend on another job's completion, the application can choose to offload them one by one, each after the previous one's result is returned, or to pack all sequential jobs into one larger, independent job. Regardless of the approach selected, there will be no dependency among the cached jobs, and each job can be scheduled individually. For example, in the *CloneCloud* case, each job is a suspended thread, with all of its related memory data transferred to the processing host.
2) *Utility:* Different jobs generally have different impacts on the user experience that differ with their completion times. Here a *utility* factor is considered. Each job carries a utility function that maps its completion time to a cumulative real-number value called the utility. The utility also incorporates the importance of the job.
3) *Host-specified:* We assume that each node does not necessarily possess all functions necessary to able to process all jobs. The requirements for the processing host are also among the factors considered in the proposed framework.
4) *Brief:* As in a distributed environment, each vehicle must provide potential processors with some information about its requested jobs for processing time estimation and further scheduling. The transmission of an entire job in such a query is unrealistic in the bandwidth-limited vehicular environment. Instead, the specification of a small number of parameters, which we call a *brief*, is required from each application to describe a job's workload. The content and format of these parameters are determined by the applications and corresponding backends.

D. Neighbor Availability Index

To help vehicles make decisions in a distributed manner, a value called the *neighbor availability index* (NAI) is introduced. The NAI can be used to indicate how many potential assisting

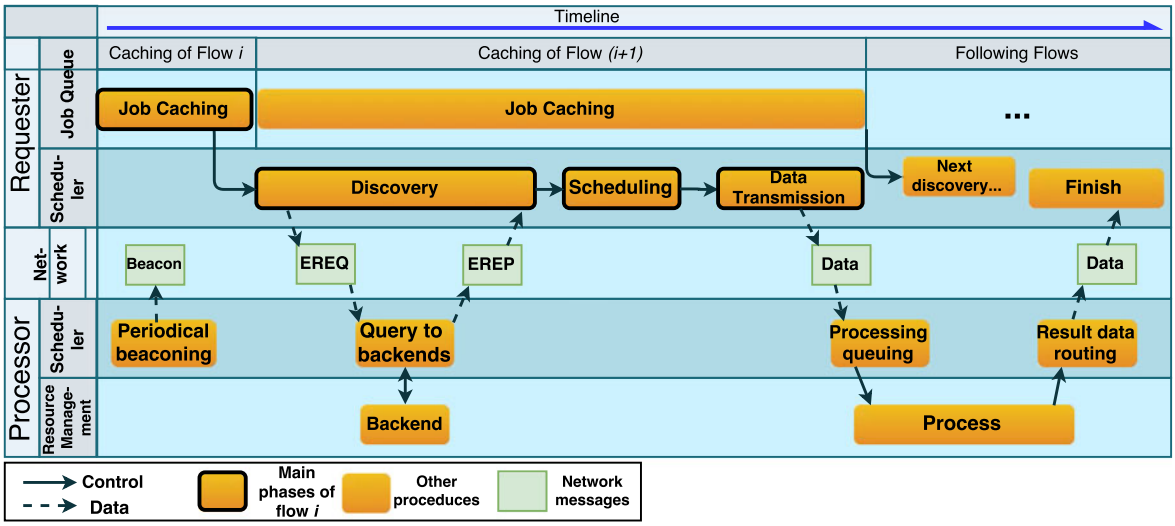

Fig. 3. Workflow of AVE. The orange boxes with borders correspond to the four main phases discussed in the following sections. Other minor procedures discussed in those sections are also shown.

vehicles are within the reachable area of a requester. Although their available resources may vary, this can still serve as a rough estimate of the nearby resources. Each vehicle calculates its NAI as

$$\text{NAI} = \sum_{k=1}^{2} (\text{number of } k\text{-hop idle neighbors}) \cdot \phi^k$$

Here, "idle" means that the neighbor's *resource manager* is not processing any job using the available resources that it manages. k specifies the number of hops to a neighbor. Because communication via a relay of more than 2 hops is unstable according to existing research [22], nodes at more than 2 hops are ignored in both the NAI calculation and further scheduling. ϕ is the factor that expresses the fading of resource usability with an increasing number of hops. "Fading" here refers to the fact that as the number of hops increases, the transmission overhead and the probability of losing the connection increases, which results in less gain from offloading. As tested in our simulation scenarios, $\phi = 0.8$ is appropriate, but the most suitable value might differ among different practical cases.

For calculation of the NAI when needed, AVE establishes an NAI table on each vehicle to store the required information. The entries in the table are in the format $\langle$*vehicle ID, idle state, number of hops, expire time*$\rangle$, and an entry is automatically removed when its *expire time* is reached. The procedure for updating this table is further discussed in Section IV.

IV. Workflow

The workflow of AVE framework consists of two components: active periodic beaconing and proactive *flows*. A *flow* represents the procedures for handling a job from its arrival to the return of its results. In this framework, we divide a *flow* into four main phases: *job caching*, *discovery*, *scheduling* and *data transmission*, as shown in Fig. 3. The details of beaconing and phases of *flow* are explained in the remainder of this section.

A. Beaconing

The AVE framework employs beacon messages to help vehicles acquire information about the resources available nearby. Each vehicle individually broadcasts a short beacon message at configurable intervals (the default considered in this paper is 10 seconds). These beacon messages are not relayed. An AVE beacon message carries the following information:

1) The source vehicle's ID,
2) its velocity vector,
3) its idle state, and
4) a list of the IDs of the source vehicle's idle 1-hop neighbors.

Note that if the list in 4) is too long, then only a few randomly selected entries are included and the rest are ignored. We choose 50 as the maximum number of selected entries in our implementation, which is sufficient in most cases.

Any receiver of this beacon message can extract the information it carries. Upon receiving a beacon, the receiver calculates the speed difference between itself and the sender as $|v - v_0|$, where v is the received velocity and v_0 is the receiver's velocity. If this speed difference is larger than a pre-assigned value, which is set to 15 m/sec (54 km/hr, approximately the speed difference when two vehicles are driving in orthogonal directions on urban streets) in the default design, then the receiver considers its link with the source node to be unstable and ignores this beacon message. Otherwise, the receiver records every vehicle mentioned in the message to its NAI table using the following rules:

1) If the vehicle is not in the table, an entry is created for it. *Number of hops* is set to 1 for the source vehicle of this beacon or to 2 for a vehicle in the source vehicle's neighbor list. *Expire time* is updated to $t_{\text{now}} + D$, where t_{now} is the current time and D is a configurable duration (where the default is the beacon interval).
2) If the vehicle is in the table and the source vehicle of the beacon is recorded as a 2-hop neighbor in the table, then

the hop number is changed to 1 and *expire time* is updated accordingly; otherwise, only *expire time* is updated.

3) Otherwise, no modification is made to the table.

B. Job Caching

Rather than scheduling and uploading each job immediately upon its arrival, the manager module runs a queue to cache jobs, where each caching duration is called a *caching window*. The main reason for caching is to avoid multiple simultaneous discovery processes and data transmissions, which not only would cause jobs to compete with each other for limited bandwidth but also might make returned messages invalid. This is because the available resources may change as a result of offloading earlier jobs. Moreover, caching allows better assignment decisions and reduces the discovery frequency. When more jobs are collected, we can more effectively find the best processor for each job by considering the job priorities and node utilization statuses.

When the first job arrives in the current window at t_{first}, an end time of $t_{\text{end}} = t_{\text{first}} + Q/(\text{NAI}+1)$ is set. Here, Q is the estimated time for this requester to process all jobs in its current processing queue, which can also be taken as the "queuing time" if a new job is to be processed locally at the current time. In particular, $Q = 0$ if this requester is idle.

The caching duration ends when both of the following conditions are satisfied:

1) the previous flow has finished data transmission, and
2) either the number of cached jobs is greater than $(\text{NAI}+1)$ or the time exceeds t_{end}.

The general idea is as follows: if there are sufficient available vehicles nearby, then we want to cache as many jobs as the current *edge* (formed by nearby nodes) can handle, so that the frequency of discovery can be reduced and a better assignment can be found. However, we do not want jobs to wait too long if resources are abundant (i.e., if NAI is high).

To eliminate the overhead incurred by useless caching in sparse areas, caching is skipped if the current *NAI* is 0. When this occurs, the *flow* is terminated. All jobs are scheduled to be processed locally, in a first-come-first-served manner.

C. Discovery

When the caching time expires, the current job queue is locked and the discovery phase begins. During the discovery phase, the requester first broadcasts an *edge* request message (EREQ) to discover potential nearby processors. This EREQ contains the requester's ID, its velocity vector, and *briefs* of the jobs cached during the last *caching window*.

Each receiver checks the velocity vector contained in the EREQ and calculates the speed difference using the same rule described for beaconing. If the speed difference is too large, this EREQ is ignored. Otherwise all 1-hop receivers relay the EREQ to nearby vehicles; i.e., they become the *forwarders* for this requester. Moreover, for each job with which the receiver is compatible, a query with the job's *brief* is sent to the *resource manager*. The *resource manager* then asks the corresponding backend to generate a *bid*. The *bid* is an estimate of the time required to finish the job, which is determined based on the necessary computation (specified in the brief provided in the job entry), the processing capacity of the vehicle, and the available computing resources that it is willing to share. We let B_{ij} denote a bid from node i for job j. After all queries are complete, if there are any *bids*, then the receiver sends back an *edge* response message (EREP). The EREP includes the responding vehicle's ID and the aforementioned *bids* for each job. If the responding vehicle is the requester's neighbor, then the EREP is sent back directly. Otherwise, the EREP will be sent to the requester via the *forwarders*. The same route is used in this framework (i.e., a node sends its EREP to the node that relayed the corresponding EREQ to it) since it is reasonable to assume that the network topology will remain the same over such a short duration. To avoid conflicts of requests from two different requesters, each receiver reserves the promised resources for a short duration (the default is 50 ms), during which it pauses in responding to requests. However, relaying is not restricted by this rule.

The requester waits for a short period of time to collect acknowledgments. The wait time is set to the maximum round-trip delay for 2-hop transmission. If this round-trip delay cannot be estimated, a manually configured time (the default is 50 ms) will be set. After the wait time expires, the requester starts job scheduling based on the collected information.

D. Scheduling

The scheduling problem scenario can be described as follows: After caching and discovery, we have a finite set of jobs and a finite set of nodes. Each job is to be transmitted to one of the nodes to be processed there. First, transmission must be completed before a job can be processed. To reduce the average transmission time, each job's transmission begins only after the previous job's transmission is finished. Because of the limited bandwidth available in a vehicular environment, the transmission time is not negligible. When the assigned processor has received all data for a job, it immediately places the job into processing queue. Queued jobs are processed one by one, in first-come-first-served order. At any given instant of time, there is only one job being executed to allow it to utilize all available resources. When the processing of a job is finished, the result of the job is transmitted back to the requester. When the result is received, the job is assumed to be finished and the corresponding utility is gained.

This scheduling approach attempts to maximize the total utility by determining *the order of jobs to be transmitted* and *the destination node of each job* (i.e., for each transmitted job, which node is selected to assist in processing). The problem can be treated as a 2-stage hybrid flow-shop problem [23], which is *NP-hard* in a strong sense [24]. The formulation of and proposed solution for the scheduling problem are described in Section V. After scheduling is finished, the requester determines the sequence of jobs to be transmitted and the destination node of each job. Then, data transmission begins immediately.

After assignment is complete, a short notification message is broadcast using the same rule as for EREQs. This notification contains the assignment result obtained during scheduling. This

is to notify the assigned nodes to stop sending EREPs and wait for the jobs that have been assigned to them to arrive.

E. Data Transmission

The offloading process begins after the scheduling phase finishes. Because the running time of the assignment algorithm is short, it is reasonable to assume that the network structure will not change compared with the network structure during the job discovery phase. Therefore, the data can be transmitted along the same paths used by the EREQs. The order of transmission is determined during the scheduling process, as discussed in Section IV-D.

Upon receiving all data for one job, a processor adds that job to the end of its processing queue. Queued jobs are processed one by one. For each job, the *resource management* module brings the corresponding *backend* into its managed VM for execution and then waits until the *backend* finishes the current job. During execution, the *backend* can utilize the promised resources of the managed VM.

After the job process is complete, the results will be sent back to the requester. Since we do not focus on ad hoc routing in this paper, we use the traditional routing protocol known as *ad hoc on-demand distance vector (AODV) routing* [25] for this transmission in our implementation. Please note that AODV is not the only choice, other unicast protocols, e.g., Predictable Broadcast Reckon (PBR) [26] are also feasible here. Moreover, if the routing path between the two vehicles is available (e.g., established by other vehicular network applications), we can take advantage of this path directly.

V. Scheduling Algorithm

In this section, the ACO-based scheduling algorithm for each node that requires the help of other nodes is explained. AVE is a decentralized platform, and each node individually makes the scheduling decisions for its own jobs based on the proposed scheduling algorithm.

A. Problem Formulation

The symbols used in the formulation are listed in Table II. The problem can be stated as follows: Two finite sets, J and P, are given, where J is the job set and P is the set of the processors. For each job $j \in J$, a utility function $\omega_j(t)$ is defined, where t is the completion time of the job (when the result is returned to the requesting application). We do not place any constraints on $\omega_j(t)$ except that it must be a non-increasing function with positive values. Thus, we have the following equation:

$$\omega_j(t_1) \geq \omega_j(t_2) > 0, \text{ for all } t_1 < t_2.$$

This assumption is based on the fact that earlier completion is better for all jobs. For different jobs (say j_1 and j_2) and the same time t, $\omega_{j_1}(t)$ may not equal $\omega_{j_2}(t)$ because of the priority differences among jobs. The corresponding application provides the ω function for each job, which accounts for the importance of the job and its real-time requirements.

TABLE II
Symbols Used in Formulating the Problem

Symbol	Explanation
J	set of jobs
P	set of nodes
$c(i,j)$	compatibility of job j and node i
S	a scheduling solution
$\omega_j(\cdot)$	utility function of job j
$\Omega(S)$	utility gained with solution S
Q_i	queuing time on node i
B_{ij}	required processing time for job j on node i
TS_{ij}	estimated time period for sending job j to node i
TR_{ij}	estimated time period for sending back job j's result from node i to the requester
p_j	time when job j arrives at its processor
q_j	time when job j's processor finishes jobs queued before j
o_j	time when job j finishes processing
t_j	completion time of job j

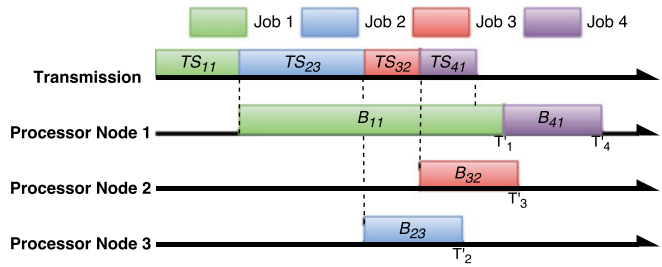

Fig. 4. An example of the time constraints of the assignment problem.

As previously mentioned, not all vehicles will necessarily possess all functions needed to be capable of processing all jobs; $c(i,j)$ is used to denote this situation. For each node i and each job j, $c(i,j) = 1$ if and only if j can be processed on i; otherwise, $c(i,j) = 0$. Note that we assume here that for any job j, there is at least one node that can process it (otherwise, the job is declared to have failed and is discarded before assignment); *i.e.*, $\sum_{i \in P} c(i,j) \geq 1$.

As shown in Fig. 4, the transmission of each job begins only after the previous job has finished. For two jobs assigned to the same node (Job 1 and Job 4 in the figure), the one that is received later must wait for the completion of the earlier job before its processing can start. The queuing time before the current scheduling on node i is represented by Q_i. In this framework, because only currently idle nodes will respond, Q_i is 0 for all nodes except the requester itself, which bears the responsibility for all of its own jobs even if it is currently busy. For a node–job pair (i,j), $TS_{ij} \geq 0$ represents the length of time required to send the data for job j to node i, $TR_{ij} \geq 0$ represents the length of time required for the processor to return the processing result, and $B_{ij} > 0$ represents the length of time required to process the job. B_{ij}, as previously discussed, is a *bid* generated by node i during the discovery phase. TS_{ij} and TR_{ij} are roughly estimated from the current network conditions (including the channel bandwidth and so forth), the number of hops to i and the amount of data to be transmitted for job j. In

particular, we have $TS_{ij} = TR_{ij} = 0$ if i itself is the *requester* of j.

A sequence $[(i_1, j_1), (i_2, j_2), \ldots, (i_n, j_n)]$, where $i_k \in P$ $(k = 1, 2, \ldots, n)$ and $\bigcup_{k=1}^{n} \{j_k\} = J$, must be found to

$$\text{maximize} \sum_{j \in J} \omega_j(t_j) \tag{1}$$

$$\text{s.t.} \quad j_k = j_h, \text{ i.f.f. } k = h; \tag{C1}$$

$$c(i_k, j_k) = 1, \ \forall k = 1, 2, \ldots, n; \tag{C2}$$

$$t_{j_k} = o_{j_k} + TR_{i_k, j_k}, \ \forall k = 1, 2, \ldots, n; \tag{C3}$$

$$o_{j_k} = \max(p_{j_k}, q_{j_k}) + B_{i_k, j_k}, \ \forall k = 1, 2, \ldots, n; \tag{C4}$$

$$p_{j_0} = 0; \tag{C5}$$

$$p_{j_k} = p_{j_{k-1}} + TS_{i_k, j_k}, \ \forall k = 1, 2, \ldots, n; \tag{C6}$$

$$q_{j_k} = \max(\{o_{j_h} | i_h = i_k, h < k\} \cup \{Q_{i_k}\}), \quad \forall k = 1, 2, \ldots, n. \tag{C7}$$

Objective function (1) represents the total utility gained from all assigned jobs, which is our target of optimization. Condition (C1) means that any given job cannot be assigned twice. Condition (C2) indicates that each processing node must be compatible with each job assigned to it. Conditions (C3)–(C7) are the aforementioned transmission and processing time constraints. p_{j_k}, q_{j_k}, and o_{j_k} all specify time instants: p_{j_k} is the receiving time, q_{j_k} is the wait time for previous jobs on the assigned node to finish, and o_{j_k} is the time when job j_k finishes processing on the node. We let $\Omega(S)$ denote the value of $\sum_{j \in J} \omega_j(t_j)$ obtained as the outcome of a solution S.

Such a sequence $[(i_1, j_1), (i_2, j_2), \ldots, (i_k, j_k)]$, denoted by S, is called a *solution* to this problem throughout the remainder of this paper. If $k = |J|$, this solution is a *complete solution*; otherwise, it is called an *intermediate solution*. Specifically, if $k = 0$, *i.e.,* for the sequence $[\varnothing]$, we call it a *null solution*. In a *solution*, each (i, j) pair represents an assignment of job j to node i. The order of the assignments in a solution is the order in which these jobs are to be sent to the processors.

The problem formulated here is a version of the 2-stage hybrid flow-shop problem with more inputs and an objective function that can take any form (a linear function or a reciprocal function, among others). The original 2-stage hybrid flow-shop problem has been proven to be *NP-hard* [24] even with only 2 nodes in the second stage. For the problem considered here, which is more difficult, it is unlikely that a complete optimizing algorithm can be found with a polynomial solution time.

B. ACO-Based Algorithm

Lacking a complete algorithm to solve the problem, we turn to heuristic algorithms, which can yield near-optimal solutions within a reasonable running time. In this paper, we use ACO-based job scheduling.

1) Ant Colony Optimization: ACO is a methodology inspired by the behavior of ants searching for paths to food. In the natural world, upon finding food, an ant lays down a pheromone trail as it returns to the colony, which then attracts other ants to follow its path. Because pheromones evaporate over time, a longer trail will tend to lose more pheromone as the ant travels back, which makes it less attractive. Conversely, a shorter path will retain more pheromone and attract more ants to follow. These additional attracted ants will also deposit more pheromone, and this positive feedback will ultimately, through iteration, attract all ants to use the shortest path.

When this concept is applied to a mathematical problem, the term "trails" often refers to state transitions. A series of consecutive transitions will lead from the starting state (null solution) to the ending state (complete solution), thereby providing a feasible solution to the problem. Typically, an ACO algorithm has four steps:

1) *Initialization:* Initialize all trails' pheromone levels.
2) *Solution construction:* Set the state to the starting state (from an empty solution). Then, repeat the selection of trails to the next state, with probabilities given by their pheromone levels, until the ending state is reached.
3) *Updating:* Update the trails' pheromone levels:
   a) reduce all pheromone levels according to the evaporation rate, and
   b) increase the pheromone levels of all trails chosen by ants.
4) *Termination:* If the termination condition is met, return the best solution. Otherwise, return to Step 2.

The ACO concept has previously been adopted to solve assignment problems [27] and scheduling problems [28] [29] with satisfactory results. Next, we will show how we solve the job scheduling problem in AVE using ACO.

2) Definitions of Trails and Pheromone Levels: To utilize the ACO algorithm for our problem, we must first define "trails" and "pheromone" in the AVE framework. Trails should represent components of the final solution, each of which can be optimized locally. $\omega_j(t_j)$ is not a good choice for local optimization because jobs are in competition for resources; when resource allocation is optimized for one job, it will always worsen for other jobs, thereby reducing the chance of improving the global performance. Therefore, we wish to find a better parameter that can represent local optimality while not being bound to specific jobs. Using Condition (C1) in the problem formulation, we can transform the objective function (1) as follows:

$$\sum_{j \in J} \omega_j(t_j) = \sum_{i \in P} \sum_{i_k = i} \omega_{j_k}(t_{j_k}) \tag{2}$$

since $\sum_{i_k = i} \omega_{j_k}(t_{j_k})$ is the total utility of jobs completed on node i. Now, we let

$$a_i(t) = \omega_{\bar{j}}(t_{\bar{j}})/(o_{\bar{j}} - q_{\bar{j}}), \text{where } x_{i\bar{j}} = 1 \text{ and } o_{\bar{j}} > t \geq q_{\bar{j}}. \tag{3}$$

Note that if there is no such $\bar{j}$ that satisfies the conditions, then $a_i(t) = 0$. Additionally, there will be at most one $\bar{j}$ that satisfies the conditions. The reason is that for each processor, there will be at most one job (say j') being processed at one time, which starts at $q_{j'}$ and ends at $o_{j'}$. The real-world meaning of $a_i(t)$ can be understood to be the "amortized utility". The idea is to spread the utility gained from finishing a job throughout the processor time it requires, which clearly includes the processing time and

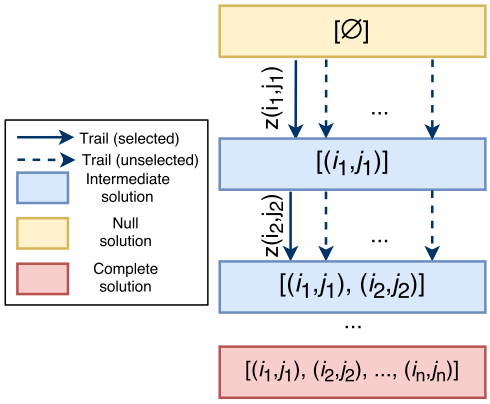

Fig. 5. Example of trails and solution construction.

also includes the wait time if its data have not arrived before the completion of the previous job.

We can see that $\int_{q_j^-}^{o_j} a_i(t)dt = \omega_j^-(t_j^-)$. Thus, it is easy to obtain

$$\sum_{i_k=i} \omega_{j_k}(t_{j_k}) = \int_0^{+\infty} a_i(t)dt \tag{4}$$

Note that although the integration here is performed on $[0, +\infty)$, this does not mean that we are scheduling up through a time of $+\infty$. This is because the jobs that we are considering right now, *i.e.,* the set J, are finite. With (4) and (2), the objective function (1) can be rewritten as

$$\text{maximize} \sum_{i\in P} \int_0^{+\infty} a_i(t)dt \tag{5}$$

Now, the goal of optimization is transformed into the maximization of the value of $a_i(t)$. $a_i(t)$ is a better parameter for measuring "trails" in ACO because it not only can be locally optimized but also contributes to global optimization. Thus, we define a trail as follows:

Definition 1: A *trail* $z(a, b)$ is a state transition that augments a null or intermediate solution $S = [(i_1, j_1), (i_2, j_2), \ldots, (i_k, j_k)]$ to another solution $S' = [(i_1, j_1), (i_2, j_2), \ldots, (i_k, j_k), (i_{k+1}, j_{k+1})]$, where $i_{k+1} = a$ and $j_{k+1} = b$.

The trail concept is illustrated in Fig. 5. The initial *null* solution is set to $S^* = [\varnothing]$. A complete solution $S = [(i_1, j_1), (i_2, j_2), \ldots, (i_n, j_n)]$, where $n = |J|$, is found by performing n consecutive transitions.

The desirability of selecting transition $z(i, j)$ in state S is given by $a_i(q_j)$. Here, the starting point q_j is chosen to represent the corresponding value in $[q_j, o_j)$. The current level of deposited "pheromone" for (i, t) (where $i \in P$ and $t \in [0, +\infty)$) is denoted by $\tau_i(t)$. This value is used as follows: if the desirability a of a trail is lower than the corresponding τ, it is multiplied by a/τ; otherwise, the desirability remains the same (a). The rationale behind this calculation is that we attempt to reduce the possibility to select a lower a so that the search can be guided toward solutions with higher objective values. Although forms other than a/τ might be feasible, we choose this factor because it is simple and is not affected by the scale of the utility values

Algorithm 1: Solution Construction.

Data: "Pheromone" level τ_i
Result: Complete solution S
Initialize current solution $S = [\varnothing]$;
Initialize set of unassigned jobs $J' = J$;
while $J' \neq \varnothing$ **do**
  for $j \in J'$ **do**
    for $i \in P$ **do**
      if $c(i, j) = 1$ **then**
        Calculate the value of $a_i(q_j)$ when transition $z(i, j)$ is applied to S;
        if $a_i(q_j) < \tau_i(q_j)$ **then**
          $\eta_{ij} = (a_i(q_j))^2/\tau_i(q_j)$;
        else
          $\eta_{ij} = a_i(q_j)$;
        end
      else
        $\eta_{ij} = 0$;
      end
    end
  end
  Select a pair (i, j) by sampling from the probability distribution $\rho_{ij} = \eta_{ij}/(\sum_{i'\in P}\sum_{j'\in J'}\eta_{i'j'})$;
  Augment S by applying $z(i, j)$ to it;
  Remove j from J';
end

(e.g., if the value of ω were to be enlarged by 10 times for all jobs, a form like $a - \tau$ would yield probability differences that were 10 times higher).

The "pheromone" level τ starts at 0. For each successfully found solution, if it improves upon the best known result, then "pheromone" with an initial amount equal to $a_i(t)$ will be deposited at the end of the current iteration. Additionally, in each iteration, the deposited "pheromone" will evaporate at a constant evaporation rate, thereby reducing the value of τ.

Therefore, we can summarize the overall procedure as follows: we first *initialize the parameters* and then repeat the process of *constructing solutions* and *updating the parameters* until the termination condition is met, i.e. when the best solution is returned. Further details of each step are described in the following section.

C. Detailed Algorithm

1) Initialization of Parameters: The first step of the ACO algorithm is to set the initial parameters. In this step, the best solution, denoted by S^*, is set to null ($[\varnothing]$), and all levels of deposited "pheromone", which are represented by $\tau_i(t)$, are set to 0 for all $i \in P$ and $t \in [0, +\infty)$.

2) Solution Construction: Then, we construct a complete solution via local optimization using Algorithm 1.

The algorithm iterates through all compatible combinations (i, j) (according to Condition (C2)) of nodes and jobs to obtain the values of the desirability η_{ij}. As discussed in the definitions, we use the amortized utility $a_i(t)$ for this purpose. The calculation of $a_i(q_j)$ is very inexpensive. Essentially, the

Algorithm 2: Parameter Update.

Data: "Pheromone" level τ_i, best solution S^*, last solution S, evaporation rate λ
Result: Updated τ_i and S^*
for $i \in P$ **do**
  Update $\tau_i(t)$ to $(1-\lambda)\tau_i(t)$ for all $t \in [0, +\infty)$;
end
if $\Omega(S) \geqslant \Omega(S^*)$ **then**
  Set the new best solution S^* to S;
  for $i \in P$ **do**
    Calculate $a_i(t)$ using solution S for $t \in [0, +\infty)$. For any t such that $a_i(t) > \tau_i(t)$, set $\tau_i(t)$ to $a_i(t)$;
  end
end

implementation can choose to keep track of the values of q (Condition (C6)) for the last job assigned during this iteration and q (Condition (C7)) for the last job assigned to node i during this iteration. Then, using (3), we obtain $a_i(q_j)$ with $O(1)$ time complexity.

η is the final desirability, or the weight, for selecting a transition. We can see that in the first iteration, η will be directly set equal to the corresponding a, which provides this iteration with a good starting point for faster convergence. Then, we sample from the probability ρ_{ij} to select j as the next job to send and i as the processor to which to assign it. In the final step, j is removed from J' such that Condition (C1) is satisfied.

3) Parameter Update: A complete solution S is found in the previous step. Then, the parameters are updated to guide the next iteration to a possibly better solution. The updating algorithm is presented as Algorithm 2.

The first loop of this algorithm simulates the evaporation process. All thresholds lose some of their strength, thereby increasing the probability that alternative solutions will be sought. The parameter affecting this process is the *evaporation rate*, denoted by λ. It is chosen to be 0.4 by default. As tested on our arbitrarily generated cases, the algorithm achieves a higher utility on average when this value is used. The second loop will be called when the last constructed solution is the new best result. This new best solution is used to update the thresholds to be at least equal to the amortized utilities. With this update, we can guide the subsequent iteration toward a better solution.

4) Termination: After an iteration is completed, the termination condition is evaluated. A typical strategy is to end the optimization after a sufficient number of iterations or when the improvement of $\Omega(S^*)$ over the last several iterations is less than a desired value. If the termination condition is not met, we return to Step 2 (Algorithm 1) with the updated τ_i and S^* value for an additional iteration.

D. Algorithm Analysis

Suppose that the number of jobs is n and that the number of nodes is m. In each iteration, one solution is constructed. Each solution consists of n assignments (trails). When selecting a trail, a maximum of $n \cdot m$ pairs are examined, each with $O(1)$ time complexity. Therefore, the time complexity of one iteration is $O(n^2m)$. Additionally, the updating of the parameters can be performed in $O(n)$ time. Thus, for t iterations, the total time complexity of the proposed algorithm is $O(tn^2m)$.

However, in practice, the convergence rate is a greater concern than the per-iteration complexity. For a preliminary evaluation of the convergence rate, we

1) arbitrarily generated jobs and nodes with randomly given parameters,
2) solved the scheduling problem using both the proposed ACO algorithm and a brute-force method, and then
3) calculated the ratio of the resulting total utility to the optimal one.

Because a brute-force search has a time complexity of $O(n!m^n)$ (n permutations of jobs and m^n assignment alternatives), the numbers of generated nodes and jobs could not be too large. We selected 5 jobs and 5 nodes for this evaluation. We generated 50 cases with random parameters, and each case was run 50 times to obtain the average outcome. We also employed a random assigning algorithm, which forms a legal solution randomly in each iteration, as a competing algorithm.

The results are shown in Fig. 6(a). This figure shows that the proposed algorithm achieves, on average, 95% of the utility of the optimal solution within fewer than 10 iterations, which demonstrates the fast coverage and good performance of the proposed scheduling algorithm. Moreover, from the results of random assignment, we can observe that 100 random iterations in the search space can achieve only 87% optimality. In addition, as tested on a desktop computer with a 3.0 GHz CPU and 8 GB of RAM, the algorithm requires approximately 40 ms to run 100 iterations, with the same input as above, whereas a brute-force search requires more than 15 seconds. Note that this is only a preliminary test; we will show how the algorithm works in practice in Section VI.

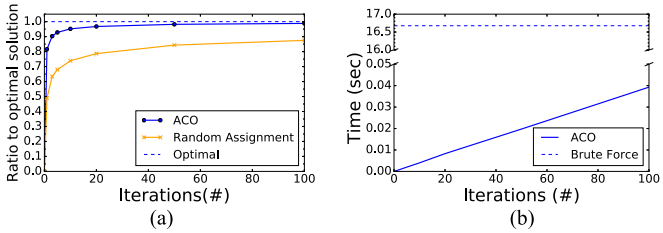

Fig. 6. Preliminary algorithm evaluation. (a) Convergence rate. (b) Running time.

VI. Evaluation

A. Simulation Setup

The simulation setup is introduced in terms of the traffic, network, application settings and competing schemes considered for comparison.

1) Simulation Tools and Traffic Simulation: To better emulate real traffic, we used *Simulation of Urban MObility* (SUMO)[30] as the traffic simulator. The *Luxembourg SUMO Traffic scenario* (LuST)[31] was used to generate the maps and

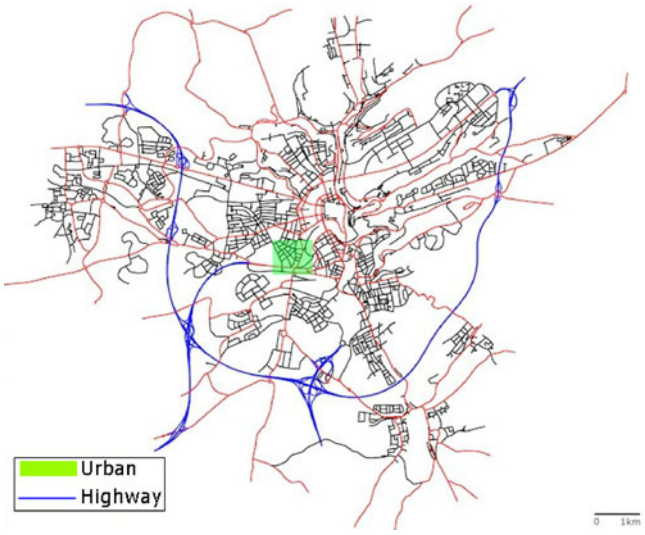

Fig. 7. Map of Luxembourg used as the traffic map in the evaluation. The blue lines delineate the highway region, and the area marked with a green block in the center was used for the urban scenario. *Figure source: LuST* [31] (modified).

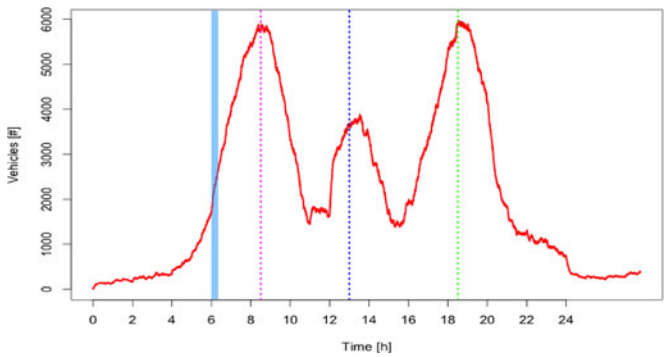

Fig. 8. Traffic demand in the *LuST* scenario. The time window selected for the simulation (6:00 *am* to 6:15 *am*) is marked with a blue bar. *Figure source: LuST* [31] (modified).

the traffic. The geographic map is a map of Luxembourg City, Luxembourg. The scenario covers an area of 155.95 km^2 and more than 24 hours of traffic. We chose two representative regions of this large map. The geographical areas of the two selected regions are shown in Fig. 7 by the blue lines and the green block. As we can see from this map, these two regions represents *highways* and *urban roads*, respectively. On highways, vehicles move faster, and the inter-vehicle distance is longer, which results in a lower density. However, there are also considerably fewer obstacles (e.g., buildings) in a highway scenario. We tested the performance of the proposed framework in these two representative scenarios. We chose a time period of 15 minutes, from 6:00 *am* to 6:15 *am*, during which the traffic demand is neither too high nor too low, as shown by the blue bar in Fig. 8. Please note that the proposal still works in higher or lower traffic demand. But some modification to the parameters (e.g., beacon interval) might be needed for optimizing performance.

2) Network: The well-known IEEE 802.11p DSRC standard was used here for the connections between vehicles. We simulated IEEE 802.11p DSRC using the *Omnet++* [32] simulator, together with *Veins* [33], a network framework for 802.11p. The carrier frequency was set to the 5.9 GHz band. The default transmission power of the vehicles was set to 20 mW, for which the maximal transmission radius in our simulation is approximately 500 meters. A simple shadowing model [34] was used for the channels. In this simulation, one random service channel was selected, and all service channels had an equal probability of being selected.

3) Application Settings: To provide universality for applications, the following scenario was chosen for consideration: Each vehicle is moving toward its destination. During driving, the driver or the passengers periodically activate applications and generate jobs; e.g., the passengers activate an AR application to look for interesting nearby objects and shut down this application once the request has been satisfied. Such applications are selected based on personal preference; hence, they are used independently by different users. When an application is activated, requests are generated. Each request is partitioned into several context-free jobs [20] (so that it can be offloaded) to be run in the background, and these jobs are submitted to the framework. To reflect their impacts on the user experience, the application assigns weight factors to the jobs. Then it issues a request to the framework to submit.

The job generation model is as follows (note that this framework also works for other models; we select this one as an example). Each application has two states: *idle* and *busy*. Jobs are generated only in the busy state. A node switches from the *idle* state to the *busy* state following a *Poisson process* with a mean arrival rate of $1/I_S$, where I_S is the expected value of the interval. In the busy state, R requests are made, each containing n jobs. Request arrival in the busy state also follows a *Poisson process*, with an arrival rate of $1/I_R$. After all requests have been made, the application transitions into the *idle* state. Each job j has attributes s_j and s'_j, which denote the sizes of the data required to be transmitted for the job to be offloaded to other vehicles and for the results to be downloaded from the other vehicles, respectively. l_j is the workload. Each vehicle has a *processing rate* r_i; thus, the processing time B_{ij} can easily be estimated as l_j/r_i. The utility function of job j is set to $\omega_j(t) = u_j\omega(t)$, where u_j is its weight factor and $\omega(t)$ is a unified function, which is $l_j/(t + l_j)$ by default. The t used in this function is the time elapsed after the generation of the job. The compatibility $c(i, j)$ is randomly set as follows: if i is not j's requester, then $Pr(c(i, j) = 1) = 0.75$; otherwise, $c(i, j) = 1$. This means that all requesters are capable of processing jobs locally.

The parameters of this simulation are listed in Table III.

Note that these values were arbitrarily set because they vary among different applications. For essential parameters such as the workload and utility function, various values were investigated. The influence of other parameters that directly or indirectly affect the workload can be deducted from the results of these investigations.

4) Competing Schemes: Two competing schemes were selected to aid in evaluating the performance of the proposed scheme. The first one is referred to as the *local* scheme, in which only local processing is adopted. In *local*, there is no

TABLE III
APPLICATION PARAMETER SETTINGS IN THE SIMULATION

Parameters	Default values (if not explicitly set)
I_S	50 sec
R	$R \sim unif(1,6)$
n	3
I_R	3 sec
u_j	$Pr(u_j = 1) = 0.6$, $Pr(u_j = 2) = 0.3$, $Pr(u_j = 3) = 0.1$
s_j, s'_j	100 kByte
l_j	Exponential distribution with a mean of 4 sec
r_i	$r_i \sim unif(1,2)$
$\omega(t)$	$l_j/(t+l_j)$

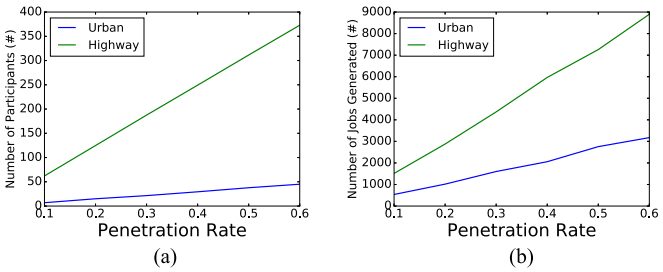

Fig. 9. Parameters of the evaluation. (a) Number of participating vehicles. (b) Number of generated jobs.

framework that allows vehicles to help each other. All jobs must be processed on the nodes that generate them. The other scheme we chose uses the proposed AVE framework but not the ACO-based scheduling algorithm. Instead, this scheme uses a simpler scheduling algorithm, which greedily assigns the jobs with the highest utility to the fastest nodes (the nodes with the shortest $TS_{ij} + B_{ij} + TR_{ij}$). This scheme is named *AVE+Naive* and has a lower computational complexity. The proposed scheme is labeled *AVE+ACO*. For the ACO algorithm used here, the termination condition is defined as "the number of iterations reaches 10".

The results shown in each figure are the average results from 10 runs. In each run, all schemes are applied to the same jobs generated at the same time points. We also show 95% confidence intervals in the related figures.

B. Simulation Results

The penetration rate is often used to describe the ratio of active wireless devices within an area [35]. Here, we use this term to refer to the ratio of vehicles participating in the framework; e.g., a penetration rate of 0.1 means that 10% of the vehicles (randomly chosen) on the road are participating. In these two scenarios, various penetration rates are considered to test the framework's performance.

To provide a general picture of the evaluation, in Fig. 9(a) and (b) we show the average number of vehicles participating in the framework and the total number of jobs generated, respectively. It can be seen that the number of jobs increases proportionally with the penetration rate, because more participants also mean more requesters.

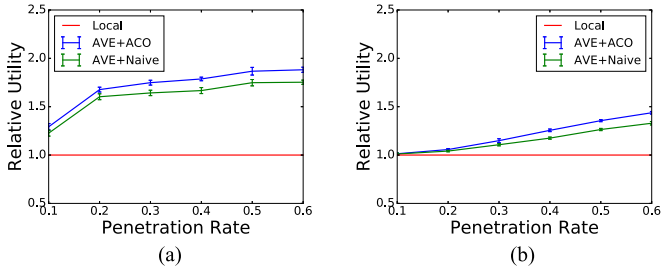

Fig. 10. Relative utility, normalized by dividing the utility gained by that in the purely local scheme. (a) Urban. (b) Highway.

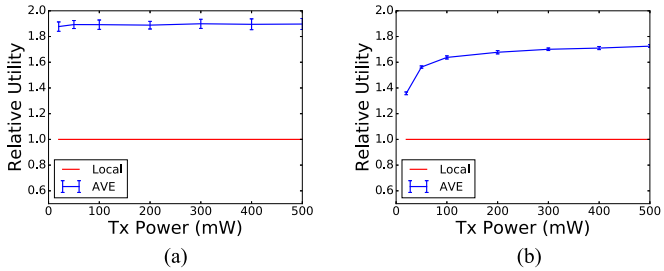

Fig. 11. Relative utility under different transmission power settings. The penetration rate is set to 0.5. (a) Urban. (b) Highway.

First, we evaluated the total utility gained in these two scenarios under various penetration rates, and the results are shown in Fig. 10(a) and (b). Given that a higher penetration rate (more participants) also means that more jobs are generated, the results have been normalized to avoid interference from the total number of jobs to facilitate interpretation. This normalization was performed by dividing the results by the corresponding results for the *local* scheme. Thus, in the following discussion of these two figures, the base *local* scheme is associated with values of 1.0. The resulting ratio is called the "relative utility" in the following discussion and is represented on the y axis in these figures. Fig. 10(a) shows the results for the urban scenario. As shown in this figure, when more participants are involved, AVE performs better. Additionally, with more participants, greater flexibility is gained in scheduling; hence, the ACO algorithm achieves a higher utility gain over the competing schemes. Fig. 10(b) shows the simulation results for the highway scenario. The trend is similar to that observed in the urban scenario. However, because of the higher mobility and longer inter-vehicle distances, the neighboring vehicles are fewer in number and the connections between these vehicles are more unstable in the time domain; hence, the increment is smaller.

We also investigated the effect of the transmission power in both scenarios. The results are shown in Fig. 11(a) and (b). As can be seen, a higher transmission power is helpful in the highway scenario because the vehicles can reach farther and find more resources. By contrast, in the urban scenario, the main restriction on communications is the blockage from buildings; therefore, a higher transmission power does not help in this case.

Then, we examined how the workload length (l_j) affects performance; the results are shown in Fig. 12(a) and (b). We can observe that for extremely short jobs, the advantages of

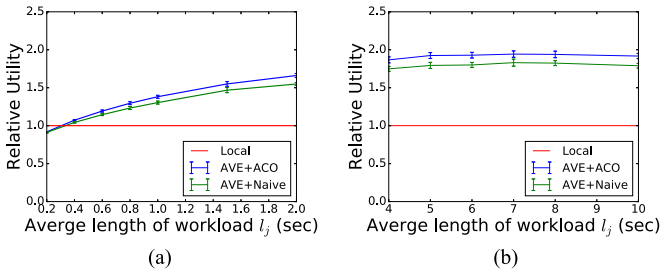

Fig. 12. Relative utility under different settings in the urban scenario. (a) Short workload. (b) Long workload.

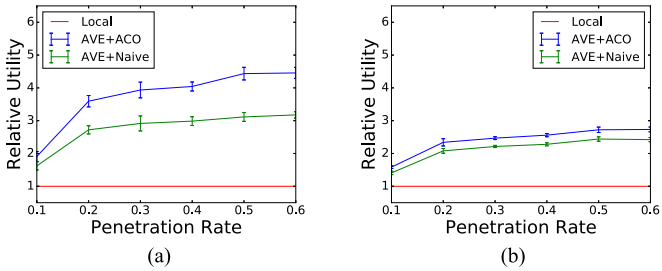

Fig. 13. Relative utility for alternative utility functions in the urban scenario. (a) $\omega_j(t) = max(u_j(1 - t/l_j), 0.01)$. (b) $\omega_j(t) = u_j e^{-t/l_j}$.

offloading and scheduling cannot compensate for the overhead incurred for discovery and transmission. At the 0.2 workload setting, in which a job requires 0.1 to 0.2 seconds for a vehicle to complete, the average outcome with the AVE framework is even slightly worse than that achieved without it. However, for longer jobs, AVE still provides a significant improvement, as observed from the figures. This is reasonable since for micro-jobs, the overhead for job discovery, data transmission, and so forth is relatively larger compared with their computational requirements than it is for macro-jobs. Here, the term micro-jobs refer to jobs with short workloads, such as function codes that run for hundreds of milliseconds, whereas macro-jobs are those with long workloads, such as data processing tasks that require seconds to run.

To demonstrate how the results are influenced by the utility function, we used a linear function (Fig. 13(a), with a minimum utility value of 0.01 to satisfy $\omega_j(t) > 0$) and an exponential function (Fig. 13(b)) as the utility function in the urban scenario. The outcome values are slightly different, but the *AVE+ACO* scheme still outperforms the other two.

For the evaluations discussed below, we set the jobs to have the same utility function (equal to u_j for all j) to allow them to be scheduled without bias.

Fig. 14(a) and (b) show the proportion of jobs offloaded to other nodes for completion. Because the job generation is the same in all cases, these results can be used to evaluate the extent to which the cloud resources are utilized. As shown in the figure, in the urban scenario with AVE and the ACO algorithm, the ratio rapidly increases with an increasing penetration rate. The ratio reaches its maximum value (0.71) when the penetration rate is 0.6. In this case, 71% jobs are successfully offloaded, whereas the percentage is 63% for *AVE+Naive*. In the highway scenario,

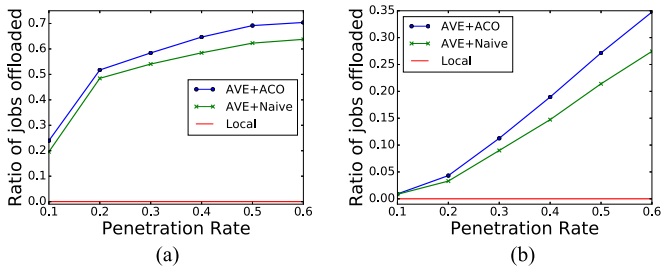

Fig. 14. Proportion of offloaded jobs. (a) Urban. (b) Highway.

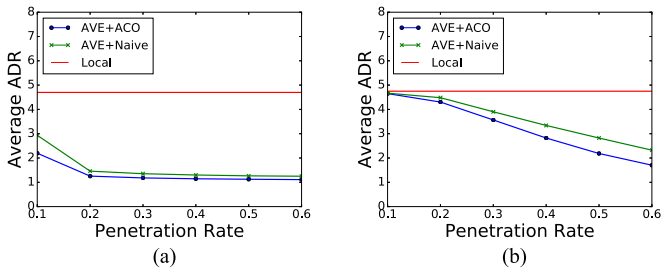

Fig. 15. ADR under different penetration rates. (a) Urban. (b) Highway.

these values are smaller, but the ACO algorithm still achieves a higher offloading ratio than *AVE+Naive* does.

To compare the improvement in terms of reducing the response times for applications, we define the *Application Delay Ratio* (ADR) [36] as

$$ADR = \frac{\text{total time taken to finish a job}}{\text{minimal time required to finish the job locally}}.$$

Here, the *minimal time required to finish the job locally* is the time required for the job to be processed by its own requester with all resources available and no queuing delay. Therefore, *ADR* can be used as a metric to evaluate the extent to which the framework reduces the response time, without any influence from the different lengths of different jobs. For a cooperative system such as AVE, if all nodes have identical processing speeds and no job has any transmission overhead or queuing delay, then the expectation value of *ADR* is 1. In Fig. 15(a), we show the average ADR values of the three schemes for different penetration rates. As shown, both AVE-based schemes quickly converge to a value close to 1 (not exactly 1 because of transmission overhead). However, in the highway scenario, as shown in Fig. 15(b), such convergence is not reached because of the lower vehicle density. Moreover, it can be observed that the difference between *AVE+Naive* and *AVE+ACO* is not always the same. The reason is that when the density of active vehicles (density of vehicles multiplied by the penetration rate) is extremely low, such as in the leftmost region of Fig. 15(b), the performance of both algorithms is limited by the lack of neighbors providing assistance, whereas when the density is very high, each processor will be processing only one or two jobs; therefore, even a simple greedy algorithm can achieve an *ADR* of near 1.0.

We also studied how closely the proposed algorithm's result approaches the theoretical optimal solution. The optimal solution was calculated using the brute-force algorithm by searching all possible options. Because of the brute-force algorithm's high

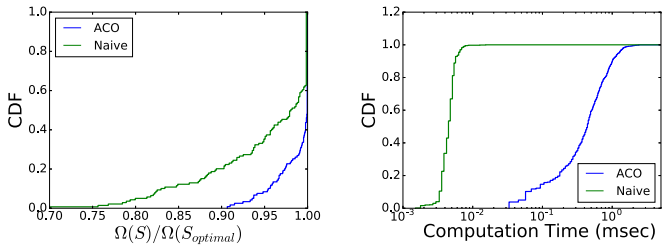

Fig. 16. Performance of the algorithms evaluated in the urban scenario with a 0.5 penetration rate. (a) CDF of $\Omega(S^*)/\Omega(S_{\text{optimal}})$. (b) Computation time of the algorithms).

time complexity (of $O(n!m^n)$), we were only able to obtain the theoretical optimal assignments at smaller scales. Therefore, problems with $n!m^n > 500000$ (where n is the number of jobs and m is the number of nodes) were ignored. We brute-force searched for the optimal solutions S_{optimal} for all remaining problems and then compared them with the solutions S^* found in the simulations. The distribution of $\Omega(S^*)/\Omega(S_{\text{optimal}})$ is shown in Fig. 16(a). As shown, in over 60% of the cases, the proposed algorithm achieves the optimal solution within 10 iterations, whereas the *naive algorithm* is less than 40% optimal. The computation time of the aforementioned algorithms is shown in Fig. 16(b). The proposed algorithm requires relative high computation time compared with the greedy algorithm. However, in most cases the computation time for ACO algorithm is less than 1 msec, which is much lower than either the transmission latency (5 to 1000 msec) or the processing time (200 to 10000 msec) here.

VII. Related Work

An increasing number of applications are being used on mobile devices, and this presents a new challenge for such devices. The reason is that because of their embedded hardware, mobile devices are subject to limitations in terms of their computational capabilities and battery life. MCC, the technology of augmenting mobile devices with cloud services or even creating a cloud among them, has proven to be an effective solution, as studied in [15]. Currently, there are several existing mobile cloud computing frameworks, such as *MAUI* [21], *MobiCloud* [37], and *CloneCloud* [20]. Additionally, *Yang* [36] has proposed an offloading scheme for applications that require low latency in MCC. The main concerns in these works are battery life, connection status and execution time.

Edge computing has been researched as a means of augmenting cloud computing. As studied by Shi *et al.* [4], this technology has the potential to shorten response times, increase processing efficiency and reduce network pressure. Ha *et al.* [38] have proposed an architecture for offloading Google Glass workloads to nearby *cloudlets*.

With regard to vehicular environments, several related studies exist concerning cloud computing systems, such as *Datacenter at the Airport* [6], *Pics-on-wheels* [7] and *Cloud Transportation System (CTS)* [8]. *Datacenter at the Airport* [6] is an attempt to utilize the resources of idle vehicles to provide cloud services in parking lots. It addresses a relatively static scenario (compared with that of vehicles on the road) and relies on access point devices for organization. Su *et al.* [39] also proposed a framework for utilizing parked vehicles as content caching nodes. Comparing with the traditional method of content-centric networking (CCN), which relies on RSUs, this proposal imposes less of a burden on RSUs, provides higher capacity and also allows vehicles to communicate via decentralized D2D communication. Similar research was done in [40], where detailed traffic and network models were built and a game-theory-based method was applied for utility optimization. *Pics-on-wheels* [7] is a proposal that vehicles are utilized as mobile cameras and organized by a centralized server to provide photo services, where vehicles serve as the sensors for acquiring pictures. The data are uploaded to cloud servers via a cellular network such that users can retrieve information from the cloud. Conversely, the *Cloud Transportation System (CTS)* [8] focuses on a crowd-sourcing scheme for collecting user data and processing them in the cloud for traffic model construction and congestion prediction. There are two main differences between AVE and the aforementioned works: the framework we propose is for generic computation offloading rather than specific application, and the main resources we utilize are located on moving vehicles.

Direct use of traditional cloud on the road is unrealistic for some applications, such as AR, for reasons such as high latency. However, edge computing is feasible in vehicular environments because communication between vehicles is constantly improving [5] and not all vehicles are running computationally intensive applications at all times. Hou *et al.* have reviewed the problem and possible solutions in [18].

VIII. Conclusion

In this paper, a framework named *Autonomous Vehicular Edge* (AVE) is proposed with the purpose of increasing the computational capabilities of vehicles. This framework specifies the workflow for vehicles and management of idle computing resources without infrastructures or centralized control. Arriving jobs are cached according to a job caching scheme, and an ACO-based scheduling algorithm is proposed to assign these jobs efficiently. Simulation environment was constructed using state-of-the-art simulation tools to evaluate the performance of the proposed scheme. The simulation results show that the proposed framework outperforms the existing schemes in terms of total utility and job delay.

The proposed framework can be further extended by introducing deployed infrastructures such as roadside units (RSUs). The decentralized scheme can be extended to solve the scheduling problem when such infrastructures are deployed. The inherent problem is how to perform discovery to efficiently utilize these infrastructures, which offer higher computation performance but short link durations because of the high relative speeds between stationary infrastructures and moving vehicles. When there is a dependency among jobs (e.g., one job's processing requires another job's output), the scheduling problem becomes much more difficult. To solve such problems is another potential direction for future research.

REFERENCES

[1] M. Armbrust *et al.*, "A view of cloud computing," *Commun. ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[2] H. S. Park, M. W. Park, K. H. Won, K.-H. Kim, and S. K. Jung, "In-vehicle AR-HUD system to provide driving-safety information," *ETRI J.*, vol. 35, no. 6, pp. 1038–1047, 2013.

[3] L. Che-Tsung, L. Yu-Chen, C. Long-Tai, and W. Yuan-Fang, "Enhancing vehicular safety in adverse weather using computer vision analysis," in *Proc. 2014 IEEE 80th VTC Fall*, 2014, pp. 1–7.

[4] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: vision and challenges," *IEEE Internet Things J.*, vol. 3, no. 5, pp. 637–646, Oct. 2016.

[5] Y. Ren, F. Liu, Z. Liu, C. Wang, and Y. Ji, "Power control in D2D-based vehicular communication networks," *IEEE Trans. Veh. Technol.*, vol. 64, no. 12, pp. 5547–5562, Dec. 2015.

[6] S. Arif, S. Olariu, W. Jin, Y. Gongjun, Y. Weiming, and I. Khalil, "Datacenter at the airport: Reasoning about time-dependent parking lot occupancy," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 11, pp. 2067–2080, Nov. 2012.

[7] M. Gerla, W. Jui-Ting, and G. Pau, "Pics-on-wheels: Photo surveillance in the vehicular cloud," in *Proc. 2013 Int. Conf. Comput., Netw. Commun.*, 2013, pp. 1123–1127.

[8] M. Meng, H. Yu, C. Chao-Hsien, and W. Ping, "User-driven cloud transportation system for smart driving," in *Proc. IEEE 4th Int. Conf. CloudCom*, 2012, pp. 658–665.

[9] M. Dorigo and L. Gambardella, "Ant colony system: A cooperative learning approach to the traveling salesman problem," *IEEE Trans. Evol. Comput.*, vol. 1, no. 1, pp. 53–66, Apr. 1997.

[10] B. Hull *et al.*, "CarTel: A distributed mobile sensor computing system," in *Proc. SenSys*, pp. 125–138, 2006.

[11] X. Feng, B. Richardson, S. Amman, and J. Glass, "On using heterogeneous data for vehicle-based speech recognition: A DNN-based approach," in *Proc. 2015 IEEE Int. Conf. Acoust., Speech Signal Process*. IEEE, Apr. 2015, pp. 4385–4389.

[12] J. Rivington and M. Swider, "Apple CarPlay: Everything you need to know about iOS in the car," 2015.

[13] G. Russo, E. Baccaglini, L. Boulard, D. Brevi, and R. Scopigno, "Video processing for V2V communications: A case study with traffic lights and plate recognition," *Proc. 2015 IEEE 1st Int. Forum Res. Technol. Soc. Ind. Leveraging Better Tomorrow*, 2015, pp. 144–148.

[14] J. W. Rittinghouse and J. F. Ransome, *Cloud Computing: Implementation, Management, and Security*. Boca Raton, FL, USA: CRC Press, 2010.

[15] L. Fangming *et al.*, "Gearing resource-poor mobile devices with powerful clouds: Architectures, challenges, and applications," *IEEE Wireless Commun.*, vol. 20, no. 3, pp. 14–22, Jun. 2013.

[16] Y. C. Hu, M. Patel, D. Sabella, N. Sprecher, and V. Young, "Mobile edge Computing—A key technology towards 5G," ETSI White Paper, vol. 11, pp. 1–16, 2015.

[17] S. Yu, Q. Liu, and X. Li, "Full velocity difference and acceleration model for a car-following theory," *Commun. Nonlinear Sci. Numer. Simul.*, vol. 18, no. 5, pp. 1229–1234, 2013.

[18] X. Hou, Y. Li, M. Chen, D. Wu, D. Jin, and S. Chen, "Vehicular fog Computing: A viewpoint of vehicles as the infrastructures," *IEEE Trans. Veh. Technol.*, vol. 9545, no. 2013, pp. 3860–3873, Jun. 2016.

[19] K. Kumar, J. Liu, Y.-H. Lu, and B. Bhargava, "A survey of computation offloading for mobile systems," *Mobile Netw. Appl.*, vol. 18, no. 1, pp. 129–140, Feb. 2013.

[20] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: Elastic execution between mobile device and cloud," in *Proc. Eur. Conf. Comput. Syst*. ACM Press, 2011, Paper 301.

[21] E. Cuervo *et al.*, "MAUI: Making smartphones last longer with code offload," *Proc. 8th Int. Conf. Mobile Syst., Appl. Serv.*, vol. 17, 2010, pp. 49–62.

[22] N. Sadagopan, F. Bai, B. Krishnamachari, and A. Helmy, "PATHS: Analysis of PATH duration statistics and their impact on reactive MANET routing protocols," in *Proc. 4th ACM Int. Symp. Mobile Ad Hoc Netw. Comput.*, Annapolis, MD, USA, 2003, pp. 245–256.

[23] P. Schuurman and G. J. Woeginger, "A polynomial time approximation scheme for the two-stage multiprocessor flow shop problem," *Theor. Comput. Sci.*, vol. 237, no. 1/2, pp. 105–122, Apr. 2000.

[24] J. A. Hoogeveen, J. K. Lenstra, and B. Veltman, "Preemptive scheduling in a two-stage multiprocessor flow shop is NP-hard," *Eur. J. Oper. Res.*, vol. 89, no. 1, pp. 172–175, 1996.

[25] C. E. Perkins and E. M. Royer, "Ad-hoc on-demand distance vector routing," in *Proc. 2nd IEEE Workshop Mobile Comput. Syst. Appl.*, 1999, pp. 90–100.

[26] V. Namboodiri and G. Lixin, "Prediction-based routing for vehicular ad hoc networks," *IEEE Trans. Veh. Technol.*, vol. 56, no. 4, pp. 2332–2345, Jul. 2007.

[27] Y.-C. Liang and A. Smith, "An ant colony optimization algorithm for the redundancy allocation problem (RAP)," *IEEE Trans. Rel.*, vol. 53, no. 3, pp. 417–423, Sep. 2004.

[28] D. Martens *et al.*, "Classification with ant colony optimization," *IEEE Trans. Evol. Comput.*, vol. 11, no. 5, pp. 651–665, Oct. 2007.

[29] C. Rajendran and H. Ziegler, "Ant-colony algorithms for permutation flowshop scheduling to minimize makespan/total flowtime of jobs," *Eur. J. Oper. Res.*, vol. 155, no. 2, pp. 426–438, 2004.

[30] D. Krajzewicz, J. Erdmann, M. Behrisch, and L. Bieker, "Recent development and applications of SUMO—Simulation of Urban MObility," *Int. J. Adv. Syst. Meas.*, vol. 5, no. 3/4, pp. 128–138, 2012.

[31] L. Codeca, R. Frank, and T. Engel, "Luxembourg SUMO Traffic (LuST) Scenario : 24 hours of mobility for vehicular networking research," in *Proc. 2015 IEEE Veh. Netw. Conf.*, 2015, pp. 1–8.

[32] V. Andras, "The OMNeT++ discrete event simulation system," in *Proc. Eur. Simul. Multiconf.*, 2001, pp. 1–7.

[33] C. Sommer, R. German, and F. Dressler, "Bidirectionally coupled network and road traffic simulation for improved IVC analysis," *IEEE Trans. Mobile Comput.*, vol. 10, no. 1, pp. 3–15, Jan. 2011.

[34] C. Sommer, D. Eckhoff, R. German, and F. Dressler, "A computationally inexpensive empirical model of IEEE 802.11p radio shadowing in urban environments," in *Proc. 2011 8th Int. Conf. Wireless On-Demand Netw. Syst. Serv.*, 2011, pp. 84–90.

[35] M. Sichitiu and M. Kihl, "Inter-vehicle communication systems: A survey," *IEEE Commun. Surveys Tut.*, vol. 10, no. 2, pp. 88–105, Apr.–Jun. 2008.

[36] L. Yang, J. Cao, H. Cheng, and Y. Ji, "Multi-user computation partitioning for latency sensitive mobile cloud applications," *IEEE Trans. Comput.*, vol. 64, no. 8, pp. 2253–2266, Aug. 2015.

[37] H. Dijiang, Z. Xinwen, K. Myong, and L. Jim, "MobiCloud: Building secure cloud framework for mobile computing and communication," in *Proc. 2010 5th IEEE Int. Symp. Serv. Oriented Syst. Eng.*, 2010, pp. 27–34.

[38] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, and M. Satyanarayanan, "Towards wearable cognitive assistance," in *Proc. 12th Annu. Int. Conf. Mobile Syst., Appl. Serv*. ACM, 2014, pp. 68–81.

[39] Z. Su, Y. Hui, and S. Guo, "D2D-based content delivery with parked vehicles in vehicular social networks," *IEEE Wireless Commun.*, vol. 23, no. 4, pp. 90–95, Aug. 2016.

[40] Z. Su, Q. Xu, Y. Hui, M. Wen, and S. Guo, "A game theoretic approach to parked vehicle assisted content delivery in vehicular ad hoc networks," in *IEEE Trans. Veh. Technol.*, vol. PP, no. 99, 2016, pp. 1–1, doi: 10.1109/TVT.2016.2630300.

Jingyun Feng (S'17) received the B.E. degree from the University of Science and Technology of China, Hefei, China, in 2012. He is currently a Ph.D. student in SOKENDAI (the Graduate University for Advanced Studies), Tokyo, Japan. His research interests include wireless network and mobile cloud computing. He is also a student member of IEICE.

Zhi Liu (M'14) received the B.E. degree from the University of Science and Technology of China, Hefei, China, in 2009 and the Ph.D. degree in SOKENDAI (the Graduate University for Advanced Studies), Tokyo, Japan. He is currently an Assistant Professor at Shizuoka University and an Adjunct Researcher at Waseda University, Shinjuku, Japan. He was a Junior Researcher (Assistant Professor) at Waseda University from December 2014 to March 2017. His research interest includes wireless networks, video/image processing and transmission, and he is a member of IEICE.

Celimuge Wu (M'10) received the M.E. degree from the Beijing Institute of Technology, Beijing, China, in 2006 and the Ph.D. degree from The University of Electro-Communications, Chofu, Japan, in 2010. He is currently an Associate Professor with the Graduate School of Informatics and Engineering, The University of Electro-Communications. His current research interests include vehicular networks, IoT, and mobile cloud computing. He has been track or workshops Co-Chair for a number of international conferences including IEEE PIMRC 2016 and CCNC 2017.

Yusheng Ji (M'94) received the B.E., M.E., and D.E. degrees in electrical engineering from the University of Tokyo, Tokyo, Japan. She joined the National Center for Science Information Systems, Japan (NACSIS) in 1990. She is currently a Professor at the National Institute of Informatics, Tokyo, Japan, and SOKENDAI (the Graduate University for Advanced Studies), Hayama, Japan. Her research interests include network architecture, resource management, and quality of service provisioning in wired and wireless communication networks. She is also a member of IEICE, IPSJ, and ACM.