Future Generation Computer Systems 93 (2019) 130–142

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Efficient data request answering in vehicular Ad-hoc networks based on fog nodes and filters☆

Yongxuan Lai [a,b], Hailin Lin [a], Fan Yang [c,b,*], Tian Wang [d]

[a] School of Software, Xiamen University, Xiamen 360000, China
[b] Shenzhen Research Institute, Xiamen University, Shenzhen 518000, China
[c] Department of Automation, Xiamen University, Xiamen 360000, China
[d] College of Computer Science and Technology, Huaqiao University, Xiamen 361021, China

ARTICLE INFO

Article history:
Received 7 July 2018
Received in revised form 14 September 2018
Accepted 28 September 2018
Available online xxxx

Keywords:
Push/pull
Data gathering
Filter cube
Fog nodes
VANET

ABSTRACT

Vehicles in urban city are equipped with more and more sensing units, and sensed data are continuously generated in large amount. These sensed data could be filtered and preprocessed before being shared or uploaded to the road side units and the cloud for efficiency. In this paper we propose a filter-based framework called FERA (Filter-based Efficient Request Answering), which combines the concept of fog computing and vehicular sensing, and adopts the pull/push strategies to adaptively and efficiently gather the requested data in vehicular ad hoc networks. Filters are defined based on the ratio of cost between the push and the pull methods to control the passage or blockage of the data readings. Moreover, filter cubes are defined to manage large number of filters, where efficient algorithms are developed to construct, update and store the filter cubes so that the matched data readings are pushed upward and unmatched data readings are blocked effectively. Extended simulated experiments demonstrate the proposed scheme has a much higher success ratio of request answering than existing schemes, e.g. REED (Abadiet al., 2005) and GeoVanet (Delotet al., 2011). Up to 94 percent of the requests could be successfully processed, while at the same time maintaining a relatively low query cost.

1. Introduction

Vehicular nodes are equipped with more and more sensing units, and large amount of sensing data such as GPS locations, speeds, video clips, and so on are generated [1,2]. These data are shared or uploaded as inputs for applications that aim at more intelligent transportations, emergency responses, and reduced pollutions and fuel consumptions. So cooperative urban sensing [3,4] is at the heart of the intelligent and green city traffic management. The key components of the platform will be a combination of pervasive vehicular ad hoc network and a central control and analyzing system. This has led to the emergence of a new kind of system, i.e. the Vehicular Ad-hoc Sensing System [5,6], where vehicles travel along roads and exchange information with encountered vehicles or nodes through V2V(vehicle to vehicle) or V2I (vehicle to infrastructure) communications. Data can be disseminated and reach a far distance by using moving vehicles as intermediates, following multi-hop routing protocols. Recently, IEEE 802 committee defined wireless communication standard IEEE 802.11p [7] that serves specifically for V2I communication. The Federal Communications Commission has allocated 75 MHz of bandwidth, which operates on 5.9 GHz channel for short range communications.

One key and challenging issue in VANET is the vehicular data gathering [2,8–10]. First, vehicular nodes are limited to road topology while moving, and under various road conditions and high moving speeds the network usually suffers rapid topology and density changes. The communications are usually fragmented and intermittent-connected. Second, the vehicular sensed data is in large amount and characterized as continuous generation. The sensed data should be filtered and preprocessed before being shared or uploaded. Data filtering technologies tailored to the VANET environment are highly needed. Generally speaking, there are two strategies to gather data: the *push*-based and the *pull*-based models, which are similar to those strategies used in the field of distributed and mobile databases. In a push-based model, each vehicle senses the data and proactively to upload data to a central server through V2V or V2I communications [11,12]. So when a node receives information from its neighbors, it has to decide whether that information is relevant or not. The system

☆ This research is supported by the Natural Science Foundation of China (No. 61672441), the Shenzhen Basic Research Program (No. JCYJ20170818141325209), the National Key Technology Support Program (No. 2015BAH16FF01), the State Scholarship Fund of China Scholarship Council (No. 201706315020), Educational Research Projects for Young and Middle-aged Teachers in Fujian (No. JA15365).

* Corresponding author at: Department of Automation, Xiamen University, Xiamen 360000, China.
E-mail address: yang@xmu.edu.cn (F. Yang).

https://doi.org/10.1016/j.future.2018.09.065
0167-739X/

incurs overheads when duplicate messages or irrelevant data are pushed. In a pull-based model, a query is issued from a node or the cloud [13,14]. Vehicles are able to understand, route, and process those queries, and finally route back the query results to the query requester. The pull-based model provides more flexibility in terms of the types of queries [15], which could in principle be diffused far away to retrieve remote data. There are three steps in the query processing: (1) query requester diffuses the request to different data sources, either directly or by using multi-hop relaying techniques, (2) each node that receives the request computes a partial result based on its local data, and (3) the nodes deliver the result to the source node of the query. However, most of existing pull-based schemes assume no fixed data server available in VANET, and they only consider the resource of the in-network vehicular nodes [14,16,15,9]. Inevitably, those approaches incur relatively large delays, especially in the VANET environments. Delays would result in failures of query result deliveries because vehicular nodes would move to other locations during the delay intervals.

This paper proposes an adaptive and efficient data gathering scheme based on the pull/push request answering model. The main idea is to devise filters that capture the data pattern to adaptively push results up along the network layers to reduce the processing delay of queries/requests. Due to the movement of vehicular nodes, it is clear that reducing the delay of queries would benefit the query processing and increase the success ratio of query processing in VANET. Also, the concept of fog/edge computing and vehicular sensing are adopted for the system design. Fog computing extends the traditional cloud computing paradigm to the edge of networks [6,17,18]. Fog nodes are new kinds of nodes that are capable of carrying out a substantial amount of storage (rather than storing primarily in cloud data centers), communication (rather than routing over the internet backbone), control, configuration, measurement and management. For example, the Intel's Next Unit of Computing [19] is a small-form-factor computer, whose motherboard measures 4×4 inches and could be integrated to the road site units deployed at the edge of networks. Fog nodes, also called *edge nodes*, are able to gather and maintain metadata about the network, requests, and vehicles. These gathered metadata are then used to generate filters that adaptively control the passages of data and requests, pruning unnecessary data transmissions. The main contributions of this paper are as follows:

1. We propose a filter-based framework called FERA (Filter-based Efficient Request Answering) that combines the pull/push strategies to adaptively and efficiently process the requests in VANET. Data readings that could pass through filters are forwarded to higher layers, and those blocked are stored at the current layer. Requests are forwarded up to edge nodes and the cloud to extract matched data. If a request is satisfied, its results are forwarded back to the source of the request. If a request is unsatisfied, i.e. no matched data are found, it is directed and forwarded down to edge nodes and ordinary nodes to further search the requested data.
2. We design efficient data structures and procedures that captures the pattern of data and requests to set and update the filters. Cost ratio is defined to calculate the states of filters, based on which the review operations are performed on filters. To update and set filters efficiently, we define the concept of filter cube and propose efficient algorithms to construct, update and store a filter cube that is effective to push matched data readings and block unmatched data readings.
3. We conduct extensive experiments to demonstrate the effectiveness of the proposed scheme in vehicular sensing applications. The proposed scheme achieves a higher success ratio of request answering than existing schemes, e.g. REED [20] and GeoVanet [15]. Up to 94 percent of the requests could be successfully processed, while at the same time maintaining a relatively low query cost.

The rest of the paper is structured as follows. Section 2 describes the related work; Section 3 introduces some preliminaries and defines the filters and the network model; Section 4 presents the overall procedures and cost analysis of the request answering framework; Section 5 describes the details of updating filters; Section 6 presents the procedures of filter cube construction and update; finally, Section 7 describes the environmental setup and analyzes the simulation results, and Section 8 concludes the paper.

2. Related work

Vehicles could be viewed as powerful mobile sensors, and numerous recent research works in vehicular networks have addressed the problem of data gathering or request answering. The solutions of queries and request answering could roughly be categorized into three types: the push-based, the pull-based, and the pull/push-based. Here we review some related works to position our work in the research community.

2.1. Push-based model

Push-based model installs constraints within the network and trigger data transmission when these conditions are met. Lee et al. [11] proposed the MobEyes system for proactive urban monitoring. The system exploits the vehicle mobility to opportunistically diffuse concise summaries of the sensed data, harvests these summaries, and builds a low-cost distributed index of the stored data to support various applications. Palazzi et al. [12] proposed a delay-bounded vehicular data gathering approach, which exploits the time interval to harvest data from the region of interest satisfying specified time constraints, and properly alternates the data muling and multi-hop forwarding strategies. Muhammad et al. [21] proposed a proactive data dissemination scheme for pushing critical content to one-hop neighbors in VANET. It treated content categorically and allowed pushing of content when necessary.

2.2. Pull-based model

Pull-based model has the data requester or query requester to request particular data, e.g. the query processing belongs to this type. Mehul et al. [13] proposed the PeopleNet that relies on the existence of a fixed network infrastructure to send a query to an area that may contain relevant information, and extract the query results. Abadi et al. [20] proposed the REED framework in wireless sensor networks, which is based on the TinyDB to store filter conditions in tables, and then distribute those tables throughout the network to extract the query results. Lee et al. [14] proposed a mobility assisted query dissemination scheme called FleaNet, where the node that submitted the query periodically advertises it only to its one-hop neighbors, which will see if they can provide some answers from information stored on their local storage. Similar to FleaNet, Zhang et al. [16] proposed a content sharing scheme called Roadcast, where a vehicular queries other encountered vehicles on the way. The keyword-based queries are submitted by the users and the scheme tries to return the most popular content relevant to the query. The researches use a "delay tolerant" strategy to handle the pull-alike requests, they cannot meet the time requirements at streaming environment.

2.3. Push/pull-based model

The push/pull-based model strikes a balance between the two strategies to achieve better efficiency. It is first introduced in the

Table 1
Comparison of the three models.

Models	References	Advantages and drawbacks
Push-based	[11,12,21]	Easy to implement, short delay, not message efficient
Pull-based	[20,13,14,16]	Message efficient, larger delay
Push/pull-based	[15,22,23]	Adaptive, tradeoff between push and pull, more complicated

area of wireless sensor network. Adam [24] presented an overview of sensor network query processing and characterized it in the context of push versus pull techniques for data extraction. Lai et al. [22] proposed a partition-based algorithm for the external join processing in sensor networks. It organizes the sensory data of the network through an optimized "value-to-storage" mapping/filter, through which tuples can choose their joining point that incurs the least communication cost. Delot et al. [15] proposed the GeoVanet scheme, where data readings are pushed to a DHT-based fixed geographical locations that allow the user to retrieve his/her results in a bounded time.

The push-based models are easy to implement, have short query delay, but they are not message efficient as all the data readings that satisfied the constraints are pushed to the server. The pull-based models are message efficient, yet they have larger delay as requests should be first forwarded to distributed nodes before collecting the results. The push/pull-based models stride a tradeoff between the push and pull-based models. They are adaptive yet more complicated to implement. Table 1 summarizes the advantages and drawbacks of these models.

Also, there are some other data sharing and delivery schemes in VANET. Zhang et al. [25] proposed a P2P content sharing scheme called Roadcast. It relaxes user's query requirement a little bit so that each user can have more chances to get the requested content quickly. And it ensures popular data is more likely to be shared with other vehicles so that the performance of overall query delay can be improved. Zhao et al. [26] proposed an approach called 3GDD for 3G-assisted data delivery in a VANET. It constructs a utility function to explore the tradeoff between delivery ratio and delivery delay, where the 3G-assisted data delivery is formulated as an optimization problem in which the objective is to maximize the overall utility under the 3G budget constraint.

2.4. Fog/edge computing

Recently, there is also a research trend to integrate the cloud and vehicular networks [27–29]. The concept of VANET cloud, however, is highly related to "fog/edge computing" [17,18,30], which extends traditional cloud computing paradigm to the edge. Fog nodes are able to provide computation, storage, and networking services between the end nodes and traditional clouds. Fog nodes reduce service latency, and improve QoS, resulting in superior user-experience [31,32]. Within the concept of fog/edge computing, more and more fog nodes are deployed at the edge of networks for various applications. The proposed scheme in this paper is a step further integrating the cloud and VANET, where resources at the cloud and the RSUs are taken advantage for the request answering. Data are cooperatively stored and indexed, and requests are processed and forwarded to specific RSUs based on the filters.

Most of the above-mentioned push or pull schemes work in a two tier static networks. Their main focuses are on the routing and message forwarding mechanisms. And filters are usually assumed to be static and not adaptive, which downgrade their performances. The proposed scheme has three layers in VANET, and adaptively adjusts the states of filters according to the cost ratio to achieve better performance. The most related work is the CEB architecture (Cloud, Edge and Beneath) proposed by Yi et al. [23]. It adopts the concept of optimal push/pull envelope to dynamically adjust the basic push and pull rates for each sensor. However, CEB solely adjusts the push/pull based on the setting of data rates that are assumed prerequisite knowledge, and the nodes are assumed to be static. On the contrary, in this paper we mainly focus on the design and setting of filters that capture and reflect the pattern of the match between requests and data readings in the dynamic environment, i.e. the VANET.

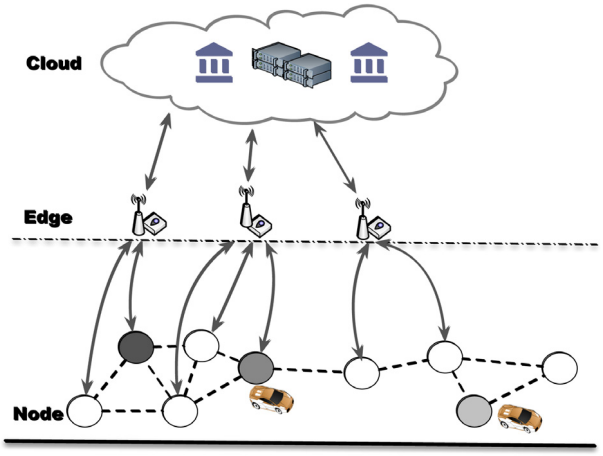


Fig. 1. Illustration of a VANET network.

3. Preliminaries

3.1. Requests and data

We assume a three layered VANET consisting of ordinary nodes, the *edge* nodes and the *cloud* as illustrated in Fig. 1. Each vehicle, v_i, monitors the road condition and surrounding environment through periodical sensing. Edge nodes provide storage and networking services between the vehicular nodes and the cloud. Data readings at ordinary nodes are denoted as $data\,(s, t, d)$, where s is the source node, t is the timestamp of the data, $d = \langle d_1, \ldots, d_K \rangle$ is a K-dimensional data reading. Data requests are submitted by users to get desired results. Without loss of generality, we assume requests are only issued from the vehicular nodes, and a request is denoted as $req(s, t, f, \Gamma)$, where s is the source node that generates the request, t is the time when request is issued, f is a filter describing the requested data, and Γ is the time interval of the requested data. Vehicular nodes would push their readings to the edge nodes, and the edge nodes would further push some of the readings to the cloud to answer requests quickly and efficiently. Also, the requests are forwarded to edge nodes and the cloud, and then forwarded down to the edge nodes or ordinary nodes to find the matched data.

3.2. Filter

Filters are assumed to be metadata that describe the ranges of data dimensions. A basic filter is denoted by $f(a_1, a_2, \ldots, a_K)$, where a_i is the range of value or set of elements at the ith dimension. a_i is either a value range when dimension d_i is continuous, or a set of elements when dimension d_i is categorical. A reading

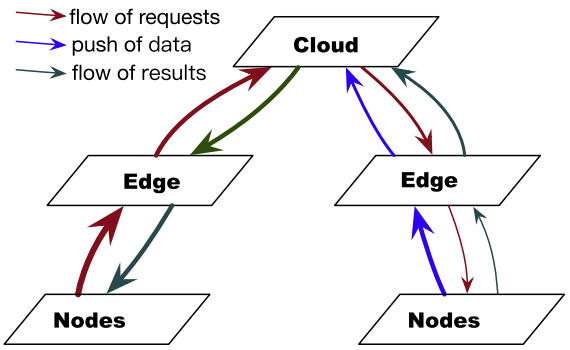


Fig. 2. Flow of requests, data, and results in request answering, where the thickness of lines indicates the amount of requests or data.

$data(s, t, \langle d_1, d_2, \ldots, d_K\rangle)$ is *compatible* to filter $f(a_1, a_2, \ldots, a_K)$ if the following conditions are satisfied:

$$d_i \in a_i, \quad i = 1, 2, \ldots, K \tag{1}$$

denoted as $f(data) = true$. Data readings are routed to their compatible filters, and the states of filters determine whether these readings could pass through or not. A basic filter has two states: "*open*" and "*close*". If the filter is at "*open*" state, the data compatible to this filter would pass through the filter, else the data would be blocked. If $f(req.f, data) = true$, $data(s, t, d)$ is said to be able to answer request $req(s, t, f, \Gamma)$, denoted by $match(req, data) = true$.

3.3. Sliding window

Requests and data arrive sequentially in a streaming environment. So we maintain a sliding window to process the data requests within a time interval Γ. Window W is denoted by $\langle t_1, t_2, \ldots, t_m\rangle$, where t_i is the ith time slot. The set of requests and data within W are denoted by R and D respectively.

Requests and data readings are matched within W to extract the requested data, where $R \ltimes D$ denotes the set of matched requests, and $D \ltimes R$ denotes the set of matched data:

$$R \ltimes D = \{r | r \in R, \exists d \in D \, s.t. \, match(r, d) = true\} \tag{2}$$

$$D \ltimes R = \{d | d \in D, \exists r \in R \, s.t. \, match(r, d) = true\} \tag{3}$$

4. Filter-based efficient request answering

There are three layers in VANET: the ordinary nodes, the edge nodes, and the cloud. Filters are installed on ordinary nodes and the edge nodes to suppress unnecessary push of data readings.

Fig. 2 depicts the request answering procedures based on the pull/push strategy. Vehicular nodes push their readings to the edge nodes. Those that could pass through the filters are forwarded to higher layer, and those that are blocked are stored at the current layer. Also, requests are forwarded to edge nodes and the cloud to extract the matched data. If a request is satisfied, i.e. finding its matched data, it is stopped at the layer and the requested results are forwarded back to the source node of the request. If a request does not find its matched data, it is forwarded down to the edge nodes and ordinary nodes to further query the requested data. The thickness of lines in Fig. 2 indicates the amount of requests or data.

In this section we present the overall description and cost analysis of the request answering framework, and in the next sections we will discuss the update mechanisms of filters and filter cubes.

Algorithm 1: Messages handling in the procedure of data push.

```
1  for all d generated at node s do
2      store d at s;
3      f = map(d); updateX(d, f);
4      if f.state == "open" then
5          forward d to Edge;
6  for all d received at Edge e do
7      store d at e;
8      f = map(d); updateX(d, f);
9      if f.state == "open" then
10         forward d to Cloud;
11 for all d received at Cloud do
12     store d at Cloud;
```

Algorithm 2: Messages handling in the procedure of data pull.

```
1  for all ⟨r, x⟩ at Node s do
2      f = map(r); updateY(r, f);
3      if d ∈ s.D matches r then
4          route ⟨r, d⟩ to source of r;
5      else if x==UP then
6          forward ⟨r, x⟩ to Edge;
7  for all ⟨r, x⟩ received at Edge e do
8      f = map(r); updateY(r, f);
9      if d ∈ e.D matches r then
10         route ⟨r, d⟩ to source of r;
11     else if x==UP then
12         forward msg⟨r, x⟩ to Cloud;
13     else if x==DOWN then
14         broadcast msg⟨r, x⟩ to nodes;
15 for all ⟨r, x⟩ received at Cloud do
16     if d ∈ Cloud.D matches r then
17         route ⟨r, x⟩ to source of r;
18     else
19         for all e in Edge Nodes do
20             if match(r, e.f) == true then
21                 forward ⟨r, DOWN⟩ to Edge e;
```

4.1. Overall description

Algorithm 1 depicts the procedures of data push strategy. Data readings are generated and stored at ordinary nodes (line 2). The *map* function returns compatible filter for data d, and function $updateX(d, f)$ updates the statistics of data about the filter (line 3). If the filter is at "*open*" state, data could pass through the filter and be forwarded to the edge node that currently covers the vehicular node (lines 1–5). Similarly, when the edge node receives data from ordinary nodes, it stores them, updates the filter statistics and forwards the data to the cloud if the filter is at "*open*" state (lines 6–10). When the cloud receives data readings from edge nodes, it just stores them (line 12).

Algorithm 2 depicts the procedures of the data pull strategy. A request message is represented by $\langle r, x\rangle$, where r is the request and $x \in \{UP, DOWN\}$ denotes the direction of the request diffusion. When an ordinary node receives a request $\langle r, x\rangle$, it first gets the compatible filter and updates the request statistic of the filter (line 2). If there are data readings in local storage that could answer the

Algorithm 3: Procedure of sliding the window.

```
1 for each time slot do
2     slide forward window W;
3     for all filter f at nodes and edges do
4         calculate cost ratio of f according to Eq. (7);
5         update state of f;
6     for all e at edges do
7         route update of filter to the cloud;
```

request, the data are routed to the source of the request, and the matching statistics about the filter are updated (lines 4). If there are not matched data and the direction of message is UP, the request is forwarded to the edge node that covers the vehicular node (line 6). Similarly, when an edge node receives a request, it checks its local storage, and data readings in the storage that could answer the request are extracted and routed to the source of the request, and the matching statistics about the filter are updated (line 8–10). If there are no matched data for the request, the request is handled according to the direction parameter. If the request is diffused up the network layer, it is forwarded to the cloud (line 12). If it is diffused down the layer, the request is broadcasted within the nodes covered by the edge node to search the matched data (line 14). When the cloud receives a request, it first searches its local storage for the match. If there are data readings that could answer the request, the data is routed back to the source of the request (line 17); otherwise, the request is forwarded down to edge nodes to search for the results (lines 19–21). Note that FERA maintains a copy of filters of all edge nodes at the cloud, so it could calculate a set of edge nodes whose filters could match the request, and the request is forwarded to these edge nodes.

The push and pull of data depend on the setting of filters installed in ordinary nodes and edge nodes. FERA adopts a sliding window to maintain statistics and states of the filters. Algorithm 3 is the pseudocode of the window sliding. At each time slot window W is slid forward with two operations. First, the *cost ratio* of the filters at the node and edge nodes is calculated and the states of the filters are updated (lines 3–5). The cost ratio is calculated distributively among ordinary nodes and edge nodes based on the statistics of the data and requests. The calculation is performed according to Eq. (7) at Section 5, where the update operations are also discussed. Second, the update of the filter at each edge node is sent to the cloud (line 7), so the cloud has the knowledge of the latest distributions of data in the edge nodes.

4.2. Overall cost analysis

The cost of the request answering actually consists of several parts:

- P1: pushing data to edge nodes or to the cloud;
- P2: forwarding requests up to edge nodes or to the cloud;
- P3: forwarding requests down to edge nodes or to the ordinary nodes;
- P4: forwarding matched results to the source nodes of requests;
- P5: exchanging auxiliary data (e.g. filters) among layers of the network.

Cost P1, P3, P4 relate to each other. The more data are pushed to edge nodes or the cloud (larger P1), the larger possibility a request could be matched. Hence fewer requests have to be forwarded down to the edges or ordinary nodes (smaller P3), which leads to smaller number of hops for the results to be forwarded back to the request source (smaller P4).

Suppose we have *oracle* knowledge about all matches between the requests and the data, then only the part of matched data need to be pushed to the cloud, where the data match requests and are routed back to the sources of the requests. We call the cost of data gathering under oracle knowledge assumption the *minimal cost*, which is denoted by $Cost(opt)$:

$$\sum_{r \in R} cost(r, r.s, cloud) + \sum_{d \in D \ltimes R} cost(d, d.s, cloud) \tag{4}$$

where $cost(a, s, t)$ is the cost forwarding a from s to t, $r.s$ is the source of request r, $d.s$ is the source of data reading d, and $D \ltimes R$ denotes a join operation that extracts the set of matched data.

In real applications it is unknown whether the requests and data readings would match or not beforehand, yet we could estimate the matches through filters. Filters determine the passage or blocking of data readings at each layer of the network, so it strikes a balance between the push and pull of the data readings. The mechanism of filter design and update plays an important role for the performance of the request answering. In the following sections we present the detailed mechanisms for the update of filters.

5. Update a single filter

When a filter is at the "*open*" state, its compatible data readings pass through; else the data is blocked at the node where the filter is deployed. When the pattern of requested data changes with time, the filter is updated accordingly through a *review* operation, which checks and calculates the state of a filter to determine whether a change of the state is needed. There are three issues that need to be concerned about in the review operation: (1) Criteria: what is the criteria to change the state of a filter? (2) Frequency: how often does the review operation should be done? (3) Efficiency: how to update and set filters efficiently, especially when there are large number of filters?

These issues relate to each other and their solutions are presented in this research. We will discuss request answering through a single filter in this section, and present the details of request answering through a filter cube in Section 6.

5.1. Cost of a filter

Data readings would pass through a filter if the data are compatible with the filter and the filter is at the "*open*" state. Some of these data answer requests, and some might not match any requests. However, when the filter is at the "*close*" state, all its compatible data are blocked and not forwarded to the upper layer. So requests that are not matched at current layer have to be routed down to the lower level to extract the requested data.

We denote the set of data and the set of requests that are compatible with filter f within the time window as D_f and R_f respectively. Then the cost of state for filter f is calculated as follows:

$$\begin{cases} cost(f, \text{“}open\text{”}) = w_0 * |D_f| \\ cost(f, \text{“}close\text{”}) = w_0 * |D_f \ltimes R_f| + w_1 * |R_f| \end{cases} \tag{5}$$

where w_0 is the factor for one-time data transmission, and w_1 is the factor for one-time request transmission and request broadcasting, $D_f \ltimes R_f$ denotes the set of data that are matched with the requests in R_f.

5.2. Criteria of state change

The state of a filter is set and updated according to a cost metric. If $cost(f, \text{“}close\text{”}) > cost(f, \text{“}open\text{”})$, it is more efficient for f be in

the "*open*" state, else it is better for f to be in the "*close*" state. In other words, if the *cost ratio* meets the following condition:

$$cost\_ratio(f) > 1 \tag{6}$$

that is,

$$\begin{aligned} cost\_ratio(f) &= \frac{cost(f, close)}{cost(f, open)} \\ &= \frac{w_0 * |D_f \ltimes R_f| + w_1 * |R_f|}{w_0 * |D_f|} \\ &= \frac{|D_f \ltimes R_f|}{|D_f|} + \frac{w_1}{w_0} * \frac{|R_f|}{|D_f|} > 1 \end{aligned} \tag{7}$$

then f is set "*open*" within a period of time. Here we define the *data match ratio* $\varphi(D_f, R_f)$ and the *request-to-data ratio* $\rho(D_f, R_f)$ as follows:

$$\varphi(D_f, R_f) = \frac{|D_f \ltimes R_f|}{|D_f|}, \quad \rho(D_f, R_f) = \frac{|R_f|}{|D_f|} \tag{8}$$

where $D_f \ltimes R_f$ denotes the set of data that are matched with the requests in R_f, and $\varphi(D_f, R_f)$ is calculated as dividing the number of matched data by the number of the whole dataset. $\rho(D_f, R_f)$ is calculated as dividing the number of requests by the number of data readings.

Formula (7) could be rewritten as:

$$cost\_ratio(f) = \varphi(D_f, R_f) + \frac{w_1}{w_0} * \rho(D_f, R_f) > 1 \tag{9}$$

If Formula (9) holds, filter f is set to "*open*" state, else it is set to "*close*" state. The main idea of our approach is to always choose a "cheaper" cost by adaptively setting the status of filters. The *cost_ratio* is calculated based on a sliding window, where the numbers of data and requests compatible with filter f are recorded. In formula (8), the set D_f and R_f is assumed not empty. Yet when $|D_f| = 0$ or $|R_f| = 0$, the state of f is simply set to "*close*" without further calculation.

5.3. Review operation

The numbers of data and requests compatible with filter f are recorded at each time slot of a window, e.g. W. Sequences that indicate the amount of data and requests are denoted by $X_f = [x_1, x_2, \ldots, x_k]$ and $Y_f = [y_1, y_2, \ldots, y_k]$ respectively, where k is the size of the window, x_i and y_i are the number of data readings and requests at the ith time slot respectively. When W moves a time slot forward, the latest numbers are added as x_k, y_k. The oldest element x_1, y_1 are removed, and other elements are updated accordingly: $x_i = x_{i-1}, y_i = y_{i-1}$. These update operations are denoted by the functions $updateX(d, f)$, $updateY(r, f)$, which are illustrated in Algorithm 1 and 2.

Given a time window W, the *data match ratio* and *request-to-data ratio* defined at Formula (8) are calculated as follows:

$$\varphi(D_f, R_f) = \frac{\sum_1^k min(x_i, x_i * y_i)}{\sum_1^k x_i}, \quad \rho(D_f, R_f) = \frac{\sum_1^k y_i}{\sum_1^k x_i} \tag{10}$$

Here when there is not matched request at the ith time slot, i.e. $y_i = 0$, $min(x_i, x_i * y_i)$ would return zero. Hence the unmatched data readings are pruned when calculating $\varphi(D_f, R_f)$. For each time slot, the algorithm recalculates the cost ratio defined at Formula (9) and determines whether to reset the filter state.

Table 2 is an example that illustrates moving the window forward when conducting the review operation, where $\hat{z}$ in X_f and Y_f means there are z readings or requests at current time slot. Given $w_0 = 1$ and $w_1 = 2$, the ratios of filter f are calculated according to Formula (10), and the *cost ratio* at t, $t+1$, $t+2$ are calculated as: $16/13 = 8/13 + (2/1) * (4/13)$, $16/21 = 8/21 + (2/1) * (4/21)$, $30/22 = 14/22 + (2/1) * (8/22)$ in time *t*, *t+1*, *t+2*. So according to Formula (9), the state of f is set as "*open*", "*close*", "*open*" during the time period.

Table 2
Example of sliding a window forward and conducting the review operations ($w_0 = 1$, $w_1 = 2$). $\hat{z}$ in X_f and Y_f means there are data readings or requests of size z at current time slot.

Time	t	$t+1$	$t+2$
X_f	$[05\hat{8}]$	$0[58\hat{8}]$	$05[88\hat{6}]$
Y_f	$[00\hat{4}]$	$0[04\hat{0}]$	$00[40\hat{4}]$
$\varphi(D_f, R_f)$	8/13	8/21	14/22
$\rho(D_f, R_f)$	4/13	4/21	8/22
cost_ratio	16/13	16/21	30/22
state	"*open*"	"*close*"	"*open*"

Table 3
Cost of states of a filter.

Cases	State	Data match	Push data	Pull data	Cost
$cost_1$	"open"	Yes	d	ϕ	$w_0 * \|d\|$
$cost_2$	"close"	No	ϕ	ϕ	0
$cost_3$	"open"	No	d	ϕ	$w_0 * \|d\|$
$cost_4$	"close"	Yes	ϕ	$d + r$	$w_0 * \|d\| + w_1 * \|r\|$

5.4. Cost analysis of models

Suppose the ratio of matched data belong to f is m, i.e. $\varphi(D_f, R_f) = m \in [0, 1]$, it is clear that the cost of push-based and pull-based strategies are calculated as follows:

$$cost_{push} = w_0 * |D_f| \tag{11}$$

$$cost_{pull} = w_0 * |D_f| * m + w_1 * |r| \tag{12}$$

Term $w_0 * |D_f| * m$ is the minimal cost $Cost(opt)$ defined at Eq. (4). So both $cost_{push}$ and $cost_{pull}$ have extra costs compared to the minimal cost. FERA aims to capture the pattern of request-data matching so as to gain on the minimal cost through setting the filter states. Given a set of data readings d, there are four cases when setting the states of filters (see Table 3):

Suppose the accuracy of setting the state of a filter is $\xi \in [0, 1]$, i.e. the probability of case 1, 2 is ξ, and the probability of case 3, 4 is $1 - \xi$. Then the overall cost of state setting for a filter in FERA is:

$$\begin{aligned} cost_{FERA} =& \xi * (cost_1 + cost_2) + (1 - \xi) * (cost_3 + cost_4) \\ =& \xi * (w_0 * |d| * m + 0) + (1 - \xi) * \\ & (w_0 * |d| * (1 - m) + (w_0 * |d| + w_1 * |r|) * m) \end{aligned} \tag{13}$$

$$= \xi * w_0 * |d| * m + (1 - \xi) * (w_0 * |d| + w_1 * |r| * m) \tag{14}$$

The cost of FERA depends on the predicting accuracy ξ. If the predicting accuracy is high, e.g. $\xi \to 1.0$, then in Eq. (14) the first term approximates the $Cost(opt)$ and the second term approximates 0, so the cost of FERA gains on the minimal cost. There has been numerals research that indicates the spatial–temporal patterns on the trajectory or vehicular applications [33], and other schemes, e.g. [34], that achieve high accuracy of request or query predication could be integrated into the proposed scheme.

6. Filter cube construction and update

A filter is updated through *review* operator, which is performed according to the cost ratios within the time window. Hereby we have handled the criteria and frequency issues about the update of filters. In this section we address the third issue, i.e. the efficiency of updating filters. We present the details of constructing, updating, and storing a large number of filters.

6.1. Filter cube

A *filter cube* is denoted by $F([D_1, s_1], [D_2, s_2], \ldots, [D_K, s_K])$, where K is the number of dimensions, $[D_i, s_i]$ denotes the range of dimension D_i is split into s_i segments. $c(sg_1, sg_2, \ldots, sg_K)$ is called a *cell*,

where sg_i is a segment in the ith dimension. For the continuous dimension, it is evenly split into segments on its value range; for the categorical dimension, it is split according to the number of elements contained in segments. Each cell has a filter attached to it, and the state of a cell refers to the state of the attached filter. There are $\|F\| = \prod_1^K s_i$ cells in F. All these cells form a cube, i.e. *filter cube*.

$F_b = F([D_1, ms_1], [D_2, ms_2], \ldots, [D_K, ms_K])$ is called the *basic filter cube*, where ms_i is the maximal number of splits on dimension D_i, $i = 1, 2, \ldots, K$. Basic filter cube has the dimensions that are split at the lowest level of granularities, and has the largest number of cells. Also, the filter that is attached to the cell in a basic filter cube is called a *basic filter*.

6.2. Cube construction

The basic filter cube F_b is constructed at the very beginning of request answering. F_b contains basic filters that have the maximal splits on all dimensions. The state of each basic filter within the cube is set based on the cost ratio, and each basic filter answers requests and prunes out unmatched data. After collecting some statistics about the data and requests, a new filter cube F that derives from F_b and has fewer cells and filters would be generated in FERA.

Two attributes are defined for a filter cube: the *size* and the *uniformity*. A filter cube is preferable when it has smaller size and higher uniformity. Smaller size means smaller number of filters, so fewer review operations are needed. Higher uniformity means that within a cell there are more basic filters that have the same state, so there are fewer mismatches between the basic filters and cells. We define a metric called *split factor* (χ) to capture these two attributes, namely:

$$\chi(F) = \alpha * \frac{\|F\|}{\|F_b\|} + (1 - \alpha) * \sum_{c \in F} \sum_{f \in \Omega_c} mis(f, c) \tag{15}$$

where $\alpha \in [0, 1]$ is a balance factor between the two attributes, $\|\circ\|$ denotes the number of cells in cube $\circ$, Ω_c denotes the set of basic filters within cell $c \in F$, and $mis(f, c)$ is a function that indicates the difference of states between basic filter f and cell c. $mis(f, c)$ returns 0 if f and c has the same state, else it returns 1. The state of a cell refers to the state of its attached filter, which is set according to the *cost ratio* defined at Formula (7).

The aim of cube construction is to find a split F^* that minimizes $\chi(F^*)$. One cube building approach works by searching all the possible partitions. It chooses a dimension D_i, iterates the dimensional splits from 1 to the maximal number of splits ms_i, and for every partition it calculates the split factor $\chi(F)$. Given K dimensions, for dimension d_i there are ms_i ways of splitting, so there are $\prod_1^K ms_i$ ways partitioning the cube. For each possible partition, there are $\|F\|$ cells and a total of $\|F\|$ operations are needed to compute the split factors $\chi(F)$. So the overall complexity of the algorithm is:

$$\Theta(\prod_1^K ms_i * \|F\|) \tag{16}$$

given that $mis(f, c)$ is a unit operation. Suppose n is the maximal number of splits of all dimensions, the overall time complexity of Formula (16) could be rewritten as:

$$O(n^{2K}) \tag{17}$$

as $\prod_1^K ms_i < n^K$, $\|F\| < n^K$. The cube building approach in this way is exponential and works only when K is small, e.g. less than 10.

In this research we adopt a greedy approach for the cube construction that is based on the *dimensional split factor*, which is defined as follows:

$$\chi(F, D_i, x) = \alpha * \frac{x}{ms_i} + (1 - \alpha) * \sum_{s \in D_i} \sum_{f \in \Omega_s} mis(f, s) \tag{18}$$

Algorithm 4: Build a Filter Cube.

```
1  split=new array(K), mf=new array(K);
2  for i ∈ [1, K] do
3      mf[i] = ∞;
4      for j ∈ [1, ms_i] do
5          calculate χ(F, D_i, j) according to formula 18;
6          if χ(F, D_i, j) < mf[i] then
7              mf[i] = χ(F, D_i, j);
8              split[i] = j;
9      split ith dimension of F into split[i] segments;
10 return F;
```

where α is the same as defined in Formula (15), x is the number of splits on dimension D_i, and Ω_s is the set of cells in F_b that belong to segment s. Function $mis(f, s)$ returns 0 if f and s has the same state, otherwise it returns 1. The number of splits of D_i is calculated as follows:

$$|D_i| = \operatorname*{argmin}_x \{\chi(F, D_i, x) : x \in [1, ms_i]\} \tag{19}$$

Algorithm 4 is the pseudocode of the greedy cube construction algorithm. It first creates two arrays to store the number of splits (*split*) and the minimized split factor (*mf*) (line 1). For each dimension, the split factor is initialized to ∞ (line 3) and the algorithm loops from 1 to ms_i to search for the splits of the dimension to calculate the minimal dimensional split factor (lines 4–8). Then that dimension is split accordingly and finally filter cube F is returned (lines 9–10). Note that in the initial phase of the request answering the mismatch $mis(f, s)$ in Formula (18) could be efficiently calculated because the requests and readings are recorded at the basic cube F_b. Also, as $\|F\| \ll \|F_b\|$, the cost of update on F is much smaller than that on F_b.

6.3. Review operation and states update

Once a filter cube is built, newly arrived data or requests are forwarded to the compatible filters in the cube. A filter is able to accumulate statistics on its compatible data and requests, and calculates its *cost ratio* to determine its state. This is called a *review* operation of a filter. The *review* operation of a filter cube is done through reviewing all filters within a cube.

For a data reading that is routed to a filter within a cube, four cases might occur: (1) the data passes through the filter and is pushed to the upper level, it then matches to a request; (2) the data passes through the filter and is pushed to the upper level, yet it is not matched by any requests; (3) the data is blocked by the filter, and it matches no requests in the higher level; (4) the data is blocked by the filter, but could match some requests from the higher level. Case 1 and 3 are called "correct" match of the filter, yet case 2 and 4 are called "incorrect" match. We defined a metric called *mismatch factor* $\epsilon(D_f, R_f)$ to reflect the cost of the incorrect match by combining case 2 and 4:

$$\epsilon(D_f, R_f) = \frac{|D_f^+| + |D_f^-|}{|D_f|} \tag{20}$$

where D_f is the set of data readings that are directed to f, D_f^+ is the set of data readings that pass through f to the higher level yet match no requests, D_f^- is the set of data readings that are blocked by f yet are pulled by requests from higher level.

Mismatch factor $\epsilon(D_f, R_f)$ reflects the error of predicted match between data readings and requests in a realtime streaming environment. Two factors would lead to a larger error $\epsilon(D_f, R_f)$ for a

Algorithm 5: Review a Filter Cube.

1 **for** *all* $f \in F$ **do**
2 calculate $\epsilon(D_f, R_f)$ by formula 20;
3 **if** $\epsilon(D_f, R_f) < \tau$ **then**
4 calculate *cost ratio* of f by formula 7;
5 update state of f;
6 **else**
7 generate basic filter cube f_b on f;
8 $f.state = basic$;
9 set timer $Timer(f, T, \epsilon(D_f, R_f))$;
10 **if** $Timer(f, T, \epsilon(D_f, R_f))$ *fires* **then**
11 $avg = \frac{\sum_{a \in f_b} mis(a,f)}{||f_b||}$, $er = \frac{avg}{\epsilon(D_f, R_f)}$;
12 **if** $er \geq \eta$ **then**
13 calculate *cost ratio* of f by formula 7;
14 update state of f;
15 **else**
16 $(I, k) = argmin_{(i,x)}\{\chi(f, D_i, x) : i \in [1, K], x \in [1, ms_i]\}$;
17 split f into k filters by splitting dimension D_I;
18 add split filters $\{f_1, ..., f_k\}$ into F;
19 $f.state$ = "*more*";

filter. The first one is *predictability error*. For example, when data and requests arrive in a random way, filters are not able to capture the patterns and hence cannot predict whether a data reading could contribute to the final matched results. In this study we assume there are some spatial–temporal patterns for the matches between the requests and data readings, and assume the predictability error is a constant given the distributions of data and requests. The second one is *granularity error*. It occurs when a filter is in relatively large granularity and could not represent the detailed characteristics of the basic filters contained in the filter. For this case, FERA would further split and transform the filter into smaller filters so that each smaller filter could have its own "*open*" or "*close*" states to control the passage of data readings, which leads to smaller mismatch factor.

Algorithm 5 presents the details of reviewing a filter cube. For every filter f in the cube, it calculates the mismatch factor $\epsilon(D_f, R_f)$. If the value is less than a predefined threshold τ, the algorithm updates the cost ratio and state (lines 2–5); else a basic filter based on f is built and a timer $Timer(f, T)$ is set for possible splits on f, and the state of f is set to *basic* (lines 7–9). Here T is a time period for observing the statistics of the filter, and during that period the incoming data compatible with f are processed by the basic filters in f_b (line 8). When $Timer(f, T)$ is fired, the average mismatch factor avg on all basic filters on f and the ratio er that divides avg by $\epsilon(D_f, R_f)$ are calculated. er is the ratio of granularity error in the mismatch factor (line 11). If er is greater or equal than threshold η (predictability error dominates), filter f is set according to the cost ratio defined by Formula (7). If er is less than η (granularity error dominates), filter f is split into k filters on dimension d_I and added to the original filter cube (lines 19–18). In line 16 the dimension and the number of splits on f are determined by the dimensional split factor defined in Formula (18). The state of filter f is also set to "*more*", meaning further filters would control the passage of data readings, i.e. data or requests compatible to f are furthered mapped to the split filters and processed there.

6.4. Storage of filter cube

The generation and update of a filter cube, which we have discussed in previous sections, also needs an efficient storage structure. We adopt a lazy storage strategy that uses hash tables [35] to store filters within a cube.

Filter cube F is split to seg_i segments on dimension $d_i, i = 1..., K$. So each cell is represented by an entry of a hash table $\mathcal{H}$, i.e. $\langle key : [h_1, h_2, \ldots, h_K], value : \{state, X_f, Y_f, S\}\rangle$, where h_i is the index for filter f on dimension d_i in cube F, X_f, Y_f are data sequence and request sequence defined at Section 5.3, and S is data structure that stores how f is split when f is at "*more*" state. A filter is only created when there are some data to be added to X_f or Y_f. For filters or cells that do not have compatible data readings or requests, no entry is needed at the hash table, so large number of storage space could be saved. It is worth noting that other data structures that handle sparse data are also feasible for the storage of filter cubes, yet the detailed description of the data structure is out of the scope of this paper.

Filters are stored in a cube with equal cells. A data reading could mapped to dimensional indexes that are used to access the filter quickly. When a data reading or request arrives to a filter cube, it needs $O(1)$ to locate the compatible filter. Moreover, a filter cube achieves two aspects of efficiency compared to the set of individual basic filters when doing the request answering. First, A filter cube F is built based on the basic cube F_b. The number of filters that it maintains is reduced by a ratio of $\frac{\|F\|}{\|F_b\|} = \prod_1^K \frac{|D_i|}{ms_i}$, where the splits at dimension D_i is smaller than or equal to the maximal split ms_i. In the cube building procedure we could also see that $\|F\| \ll \|F_b\|$ holds. Second, a lazy and spare storage strategy is adopted for the cube storage. A large proportion of filters within a filter cube are "empty" filters because no data readings or requests are compatible to them. So they do not need any storage structure to maintain their statistics of the readings or requests.

7. Experimental study

7.1. Environment setup

We conduct experiments on the ONE platform [36] with real-world road network to verify the performance of the proposed algorithm. The ONE is a popular simulation environment that is capable of generating node movement using different movement models and routing messages between nodes with various routing algorithms.

7.1.1. Trajectory dataset and network setting

The Xiamen Taxi Dataset[1] is used for the simulation. The dataset contains trajectories of about 5000 taxis in Xiamen city, China during July 2014. The region is limited to $[118.066E, 118.197E] \times [24.424N, 24.561N]$, and maps provided from OpenStreetMap is used to build a road network. In the simulation, the most active 300 taxis are selected to act as vehicular nodes. Each vehicle moves along the historic trajectory. The moving speed ranges from 0 to 72 KM/h, which differs according to road segments and time periods. The simulation runs on a 64 bit desktop with 8G memory and Intel CoreTM i7 3.60 GHz CPU.

There are 81 edge nodes (RSUs) evenly deployed in the map, and each edge node periodically updates its filters in the cloud every 60 s. The communication range of I2I or I2V used by the vehicles to exchange data is set to 200 m. The total simulation time is 6 h within a day, from 8:00 to 14:00. The size of sliding window is 5 min, and each time slot is 30 s. The ratio of unit cost $\frac{w_1}{w_0}$ defined in Formula (9) is 1/2 by default. As the proposed scheme

[1] http://mocom.xmu.edu.cn/xmdata.

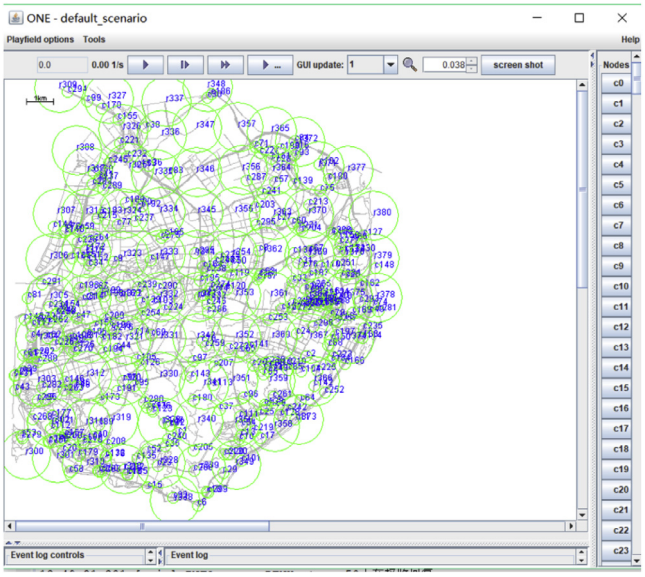


Fig. 3. Snapshot of the simulation field in Xiamen Island. The blue texts denote vehicular nodes, and green circles denote the coverage areas of RSUs.

belongs to the application layer of the network protocol stack, we assume ideal links when two nodes encounter and establish a connection. The size of a message is set 1024 Byte and the metadata are wrapped in one message per request. The bandwidth of the V2V or V2I channel is 500 Kbps/250 Kbps for the down/up links.

7.1.2. Data and request generator

Data readings are in the form of $(s, t, lat, lon, type, size)$, where s is the ID of the node that generates the data, t is the time, lat, lon are the latitude and longitude of the location, $type$ is the type of the generated data, and $size$ is the detailed sampled data whose size corresponds the type of the data. In the experiment there are five types of data whose sizes are {16 K, 64 K, 512 K, 1024 K, 18 M}. Each vehicle periodically reports one data reading every 150 s, and we construct a request data *generator* to synthesize the data readings.

Requests are in the form of $req(s, t, f, \Gamma)$, where s is the source node that generates the request, t is the time when request is issued, f is a filter describing the requested data, and Γ is the time interval of the requested data. Here, the filter is defined as $\langle type, lat, lon\rangle$, where $type$ denotes the type of the requested data, lat, lon are the latitude and longitude of the position to which the requested data belong. Also, the requests are generated from two ways: (1) each origin–destination (OD) pair is mapped to a request that is generated from the origin and targets the data readings from the destination, so the real-world origin–destination dataset is integrated into the request answering. Existing research [33] has disclosed that there are some spatial–temporal patterns in the OD pairs, which matches the request answering scenarios in this paper; (2) requests are generated in the form of Zipf's law [37], where a skewness parameter determines where target locations of the requests. We construct a request *generator* to control the generation process of requests, and the query rate, skewness, and deadlines are defined as parameters of the generator.

For both the data and requests, the domain of the latitude and longitude dimensions are [118.066 E, 118.197 E], [24.424 N, 24.561 N] respectively, as showed in Fig. 3. The deadline of query is five minutes, and the queries are generated according to uniform distribution from 2 to 5 s by default.

7.1.3. Evaluation metrics

We use the success ratio and the transmission cost as the main metrics for performance evaluation. The success ratio is defined as:

$$success\_ratio = \frac{n\_s}{n\_req} \tag{21}$$

where n_req denotes the number of requests submitted by all the vehicular nodes, n_s denotes the number of requests whose query results are successfully received by the requested nodes. The transmission cost is represented by the number of messages that include the requests forwarding, filters and data pushing, and query results forwarding.

7.2. Performance analysis

We compare the proposed FERA scheme with other schemes. Yet to the best of our knowledge, there are few research directly related to the data request answering schemes in VANET, so for the performance comparison, we implement other four request schemes as follows:

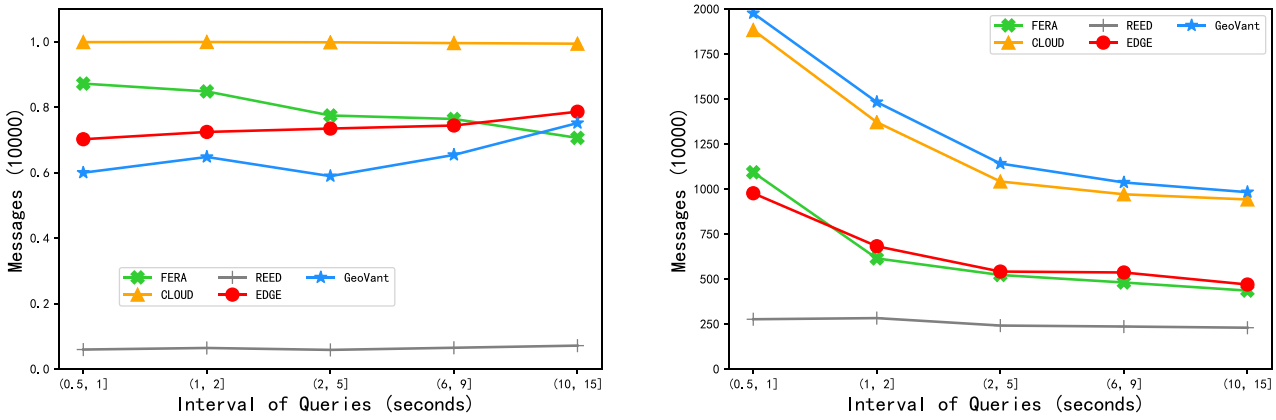
- CLOUD: all sensed data are uploaded to a centralized cloud server. Requests are processed at the cloud and results are routed back to the requested node;
- REED [20]: a pull based method where the sensed data are stored locally, and all requests are forwarded to RSUs and broadcasted to search the requested data readings;
- EDGE: all the sensed data are stored in the edge nodes (RSUs), and requests are forwarded to all edge nodes to search for the requested data readings;
- GeoVanet [15]: data readings are first pushed to a DHT-based fixed geographical locations, and requests are forwarded to this location to extract the matched data within a bounded time interval.

We vary the parameters to study their impacts on these schemes.

7.2.1. Interval of requests

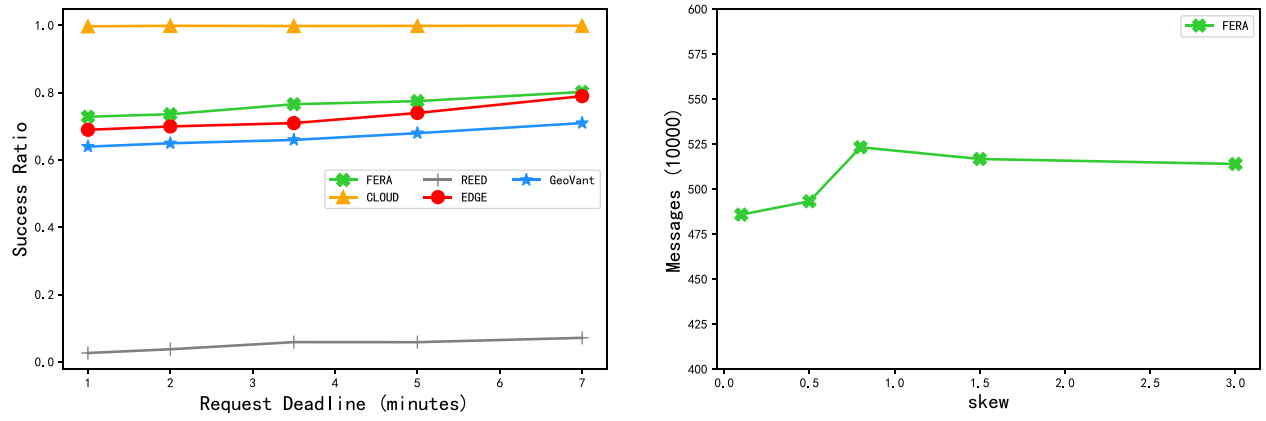
Data requests are generated periodically by a request generator, where a smaller interval means a larger number of requests. Fig. 4 depicts the impact of the request interval. From 4(a) we could see that the CLOUD approach achieves the best success ratio, as high as 0.98, and the REED approach has the lowest success ratio that is around 0.07. The success ratios of FERA, EDGE and GeoVant are in the middle. Most of the failed requests are due to the fact that the vehicular node that issues the request will move to other places, and when the query results are returned, they cannot find the requester. Also, when the request is outdated, the request would be failed. Yet for the CLOUD approach, because all the data are uploaded to the centralized server, the query processing time is relatively small so the results could be routed back to the requester just before vehicular nodes move out the coverage areas of RSUs. However, the CLOUD approach incurs large number of message transmissions, as depicted in Fig. 4(b). The success ratio of FERA is about 0.84 when the interval is (0.5,1], yet it decreases as the requests interval increases. This is because when there are more requests, the pattern would be captured by the filters, which would adaptively adjust the states of filters and push more data readings to upper layers of the networks. In this way, FERA has fewer request processing time and higher success ratio.

Fig. 4(b) depicts the number of the messages. From the figure we could see that the amount of messages decreases as the request interval increases. This is easy to explain as there are fewer requests when the request interval is larger. The GeoVant and the CLOUD have largest amount of message transmissions. The



(a) Success Ratio (b) Messages

Fig. 4. Impact of queries intervals.



(a) Success Ratio (b) Messages

Fig. 5. Impact of request deadlines.

message transmissions in the FERA and EDGE schemes are about 48–52 percent of those in the CLOUD and GeoVant schemes. This is because in FERA and EDGE only part of the data readings are pushed to upper layer of the network, yet in CLOUD and GeoVant all the data readings are routed to the cloud or hashed point, which incur the largest transmission cost.

7.2.2. Deadline of request

Fig. 5 depicts the impact of the request deadlines. The success ratio increases as the request deadline increases for all the approaches except CLOUD. This is understandable as when the deadline of requester are larger, fewer requests would be outdated and failed. The success ratio of FERA increase from 0.728 to 0.802. The request deadline has a relatively small impact on the number of message transmissions.

7.2.3. Skewness of requests

Request generator has a skewness parameter that determines where the target locations of requests are. In the experiment the map is split into 81 grids, and requests are generated in the form of Zipf's law [37]. The *skew* parameter determines the skewness of requests targeting the grids. From Fig. 6, we could see that the success ratio of FERA increases from 0.72 to 0.86 when the skewness parameter increases from 0.1 to 3. Higher skewness means more requests are routed to the same grids, hence it facilitates filters to control their states effectively. The message transmissions first

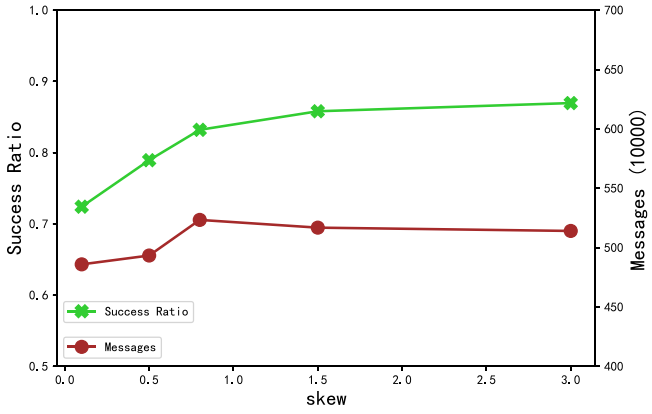


Fig. 6. Impact of query skewness.

increase with the skewness, and then decrease with the skewness factor. When skewness is low, the requests are distributed among grids. Each RSU would receive smaller amount of requests, and the filters would be more likely to be at the "close" state. So there are fewer message transmissions compared to those when with larger skewness.

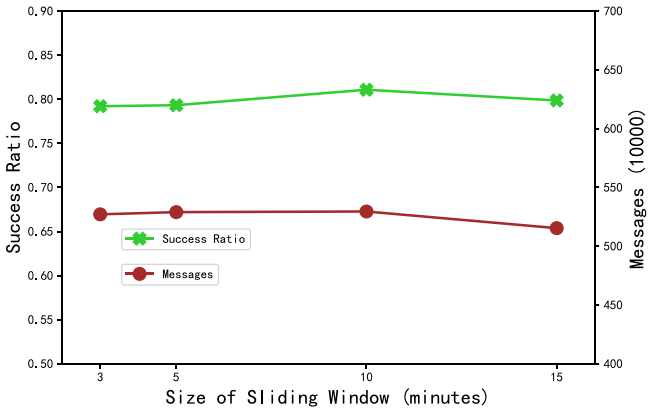


Fig. 7. Impact of sliding window.

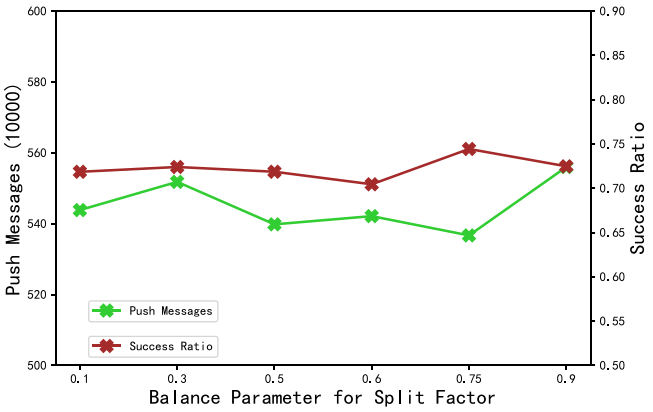


Fig. 9. Impact of balance parameter for the split factor.

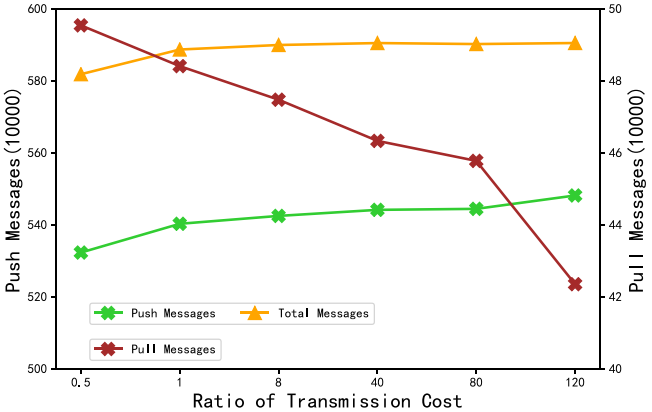
7.3. Impact of parameters of FERA

7.3.1. Size of sliding window

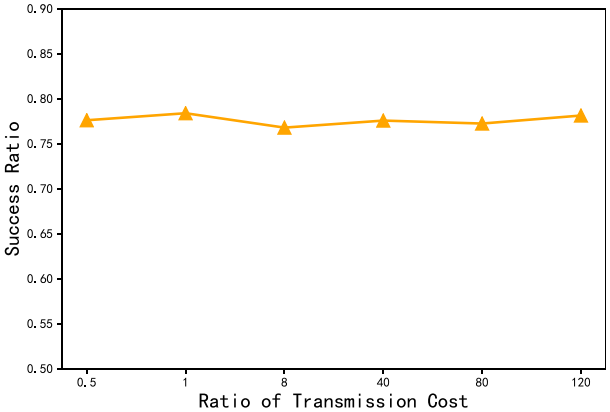
FERA uses a sliding window to control the states of the filters. So in the experiment we vary the window size to study its impact on the performance. As showed in Fig. 7, the success ratio achieves the best performance at 0.86 when the size is 10 min. A window size either too smaller (e.g. 2 min) or too larger (e.g. 15 min) does no good for utilizing the pattern of requests, which harms the success ratio. The size of sliding window has a relatively smaller impact on the message transmissions. The number of messages decreases a little when the window size is large. This is because a larger window makes it harder to change the states of filters, and hence the data readings would be blocked by the "closed" filters.

7.3.2. Ratio of transmission cost

We use $\frac{w_1}{w_0}$ to denote the ratio of transmission cost for a one-time request transmission versus a data transmission. When the cost of request transmission is larger than that of data transmission, i.e. a larger $\frac{w_1}{w_0}$, FERA prefers pushing the data from lower layers; otherwise, FERA prefers pulling the data from the upper layers. This is verified by the experiments. From Fig. 8(a) we could see that the number of push messages increases with the cost ratio, and the number of pull messages decreases with cost ratio. But generally, the amount of push messages is much larger than that of the pull messages, and the overall success ratio is relatively stable, as depicted in Fig. 8(b).



(a) Messages



(b) Success Ratio

Fig. 8. Impact of ratio of transmission cost.

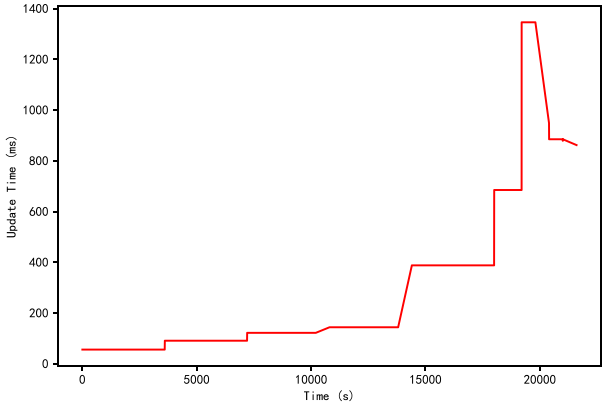
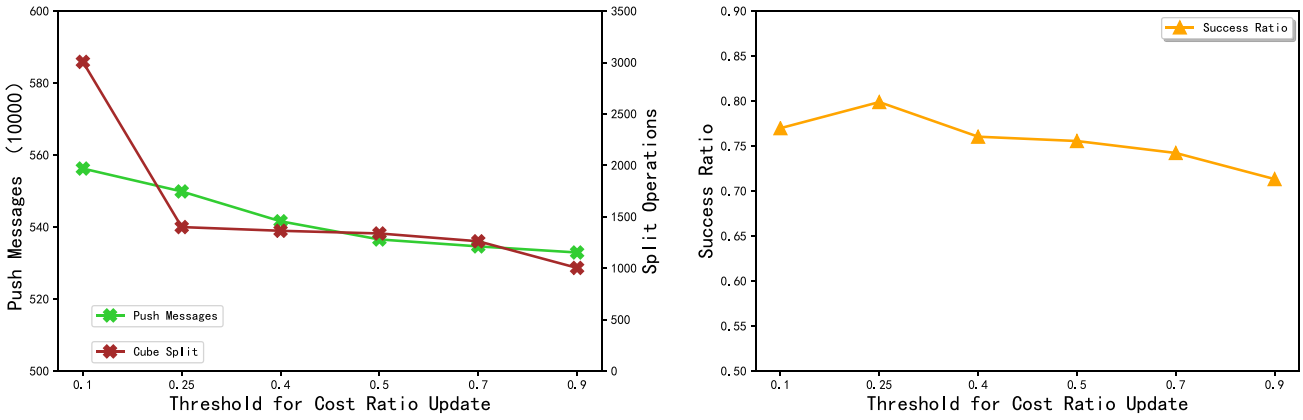


Fig. 10. Update Time of filter cube.

7.3.3. Balance parameter for the split factor

FERA uses a balance parameter α for the *dimensional split factor* in Formula (18). It balances the size and the uniformity of filters. From Fig. 9 we could see that when α is 0.75, the success ratio gets to the maximal, and at the same time the push messages is the lowest. So we choose 0.75 as the default value for α.

The average time of updating a filter cube is also depicted in Fig. 10 in our simulation. The cost time increases as the request answering progresses. This is because more filters cells are generated as data and requests are received by the nodes and RSUs, so



(a) Messages (b) Success Ratio

Fig. 11. Impact of threshold for cost ratio update.

the time for reviewing a filter cube increases. The time of update increases to the maximal of 1344 ms when the time is at about 19 203 s, and then it decreases to about 862 ms. The decrease is mainly because of restructuring filters in the cube.

7.3.4. Threshold for filter cube update

In the cube update operation in Algorithm 5, threshold τ is used to determine whether updating the cost ratio and state or building basic filter cube. Larger τ means it harder for filters to be split. Fig. 11(a) depicts the number of push messages and the number of split operations versus threshold τ. From the figure, we could see that as τ increases from 0.1 to 0.9, the number of split operations decreases from about 3000 to 1000 and the number of push messages decreases from 5.56 to 5.32 ($\times 10^6$). However, the success ratio increases to the maximal when τ is 0.25, and then decreases as τ grows larger, i.e. when it is harder to split the filters. This indicates the necessary to split the filter when the error $\epsilon(D_f, R_f)$ grows larger. So the threshold τ is set 0.25 by default in our experiment. Similarly, we vary the other threshold η that determines the proportion of granularity error or predictability error, and set the threshold as 0.6 by default.

7.4. Discussions and comments

FERA adopts the pull/push strategies to adaptively and efficiently gather the requested data in vehicular ad hoc networks. It has larger request success ratio and fewer message transmissions compared to other schemes that are solely push-based or pull-based. Although the CLOUD has higher success ratio, the CLOUD is actually infeasible in request answering scenarios. Firstly, the CLOUD scheme incurs about twice of the message transmissions of that in FERA, which makes it expensive. Secondly, in the CLOUD scheme each vehicle has to establish a connection with the cloud to upload its data readings. The in-network processing is not available in this scheme. Instead, it needs a scalable and much more expensive backend system to answer the requests when there are large number of vehicles.

The request deadline has a relatively small impact on the number of message transmissions, but the skewness of requests plays a role. Higher skewness means more requests are routed to the same grids, hence it facilitates filters in FERA to control their states effectively. For the sliding window, either smaller or larger size does no good for utilizing the pattern of requests, which harms the success ratio. Actually, the states of filters are determined by the ratio of transmission cost. When the cost of request transmission is larger than that of data transmission, FERA prefers pushing data from lower layers; otherwise, FERA prefers pulling the data from the upper layers.

The time of updating filter cube reflects the cost of reviewing large number of filters. As the request answering progresses, more filters cells are generated as data and requests are received by the nodes and RSUs, so the time for reviewing a filter cube increases. Also, the balanced factor and threshold for filter cube update are studied. The experiments indicate the necessary to split the filter when the mismatch error $\epsilon(D_f, R_f)$ grows larger.

8. Conclusions

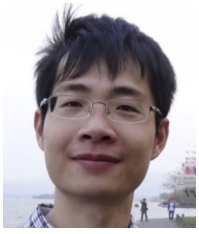
We have proposed a filter-based framework FERA that combines the pull/push strategies to adaptively and efficiently process data requests in VANET. Data readings that could pass through the filters are forwarded to higher layer, and those blocked are stored at the current layer. Requests are forwarded up to edge nodes and the cloud to extract matched data. Also, to update and set filters efficiently, the concept of filter cube is proposed and efficient algorithms are developed to construct, update and store a filter cube that is effective to push matched data readings and block unmatched data readings. Experiments based on simulations are conducted to demonstrate the effectiveness of the proposed scheme in vehicular sensing and request answering applications. The proposed scheme has much higher success ratio of request answering than existing schemes, while at the same time with a relatively low query cost.

For the future work, we are going to further optimize the structure of filters, e.g. the dynamic data structures that could store and update the filters efficiently. As FERA adopts a best-effort approach for the request answering, we are also to investigate effective approaches and integrate it to our framework to guarantee the return of data results, i.e. the successful ratio is higher than some thresholds.Also, the data traffic scheduling techniques, e.g. priority queuing, first-in–first-out and weighted fair queuing as discussed in [38], could be integrated into the proposed framework.

References

[1] U. Lee, E. Magistretti, M. Gerla, P. Bellavista, A. Corradi, Dissemination and harvesting of urban data using vehicular sensing platforms, IEEE Trans. Veh. Technol. 58 (2) (2009) 882–901, http://dx.doi.org/10.1109/TVT.2008.928899.
[2] Y. Lai, F. Yang, L. Zhang, Z. Lin, Distributed public vehicle system based on fog nodes and vehicular sensing, IEEE Access 6 (2018) 22011–22024.
[3] G. Pau, R. Tse, Challenges and opportunities in immersive vehicular sensing: lessons from urban deployments, Signal Process., Image Commun. 27 (8) (2012) 900–908.

[4] Y. Lai, F. Yang, J. Su, Q. Zhou, T. Wang, L. Zhang, Y. Xu, Fog-based two-phase event monitoring and data gathering in vehicular sensor networks, Sensors 18 (1) (2017) 82.
[5] S. Al-Sultan, M.M. Al-Doori, A.H. Al-Bayatti, H. Zedan, A comprehensive survey on vehicular ad hoc network, J. Netw. Comput. Appl. 37 (2014) 380–392.
[6] A. Dua, N. Kumar, S. Bawa, A Systematic Review on Routing Protocols for Vehicular Ad Hoc Networks, Vol. 1, Elsevier, 2014, pp. 33–52.
[7] D. Jiang, L. Delgrossi, Ieee 802.11 p: towards an international standard for wireless access in vehicular environments, in: Vehicular Technology Conference, 2008. VTC Spring 2008. IEEE, IEEE, 2008, pp. 2036–2040.
[8] Y. Lai, J. Xie, Z. Lin, T. Wang, M. Liao, Adaptive data gathering in mobile sensor networks using speedy mobile elements, Sensors 15 (9) (2015) 23218–23248.
[9] T. Wang, J. Zeng, Y. Cai, H. Tian, Y. Chen, B. Wang, et al., Data collection from wsns to the cloud based on mobile fog elements, Future Gener. Comput. Syst. (2017).
[10] M. Dighriri, G.M. Lee, T. Baker, Measurement and classification of smart systems data traffic over 5g mobile networks, in: Technology for Smart Futures, Springer, 2018, pp. 195–217.
[11] U. Lee, B. Zhou, M. Gerla, E. Magistretti, P. Bellavista, A. Corradi, Mobeyes: smart mobs for urban monitoring with a vehicular sensor network, IEEE Wirel. Commun. 13 (5) (2006) 52–57.
[12] C.E. Palazzi, F. Pezzoni, P.M. Ruiz, Delay-bounded data gathering in urban vehicular sensor networks, Pervasive Mob. Comput. 8 (2) (2012) 180–193.
[13] M. Motani, V. Srinivasan, P.S. Nuggehalli, Peoplenet: engineering a wireless virtual social network, in: Proceedings of the 11th annual international conference on Mobile computing and networking, ACM, 2005, pp. 243–257.
[14] U. Lee, J. Lee, J.-S. Park, M. Gerla, FleaNet: a virtual market place on vehicular networks, IEEE Trans. Veh. Technol. 59 (1) (2010) 344–355.
[15] T. Delot, N. Mitton, S. Ilarri, T. Hien, GeoVanet: A routing protocol for query processing in vehicular networks, Mob. Inf. Syst. 7 (4) (2011) 329–359.
[16] Y. Zhang, J. Zhao, G. Cao, Roadcast: a popularity aware content sharing scheme in vanets, ACM SIGMOBILE Mob. Comput. Commun. Rev. 13 (4) (2010) 1–14.
[17] J. Zeng, T. Wang, Y. Lai, J. Liang, H. Chen, Data delivery from wsns to cloud based on a fog structure, Fourth IEEE Int. Conf. Adv. Cloud Big Data accepted (3) (2016) 959–973.
[18] T. Wang, J. Zhou, M. Huang, M.Z.A. Bhuiyan, A. Liu, W. Xu, M. Xie, Fog-based storage technology to fight with cyber threat, Future Gener. Comput. Syst. 83 (2018) 208–218.
[19] Intel, Next unit of computing, https://www.intel.com/content/www/us/en/products/boards-kits/nuc.html.
[20] D.J. Abadi, S. Madden, W. Lindner, Reed: robust, efficient filtering and event detection in sensor networks, in: Proceedings of the 31st international conference on Very large data bases, VLDB Endowment, 2005, pp. 769–780.
[21] M.F. Majeed, S.H. Ahmed, M.N. Dailey, Enabling push-based critical data forwarding in vehicular named data networks, IEEE Commun. Lett. 21 (4) (2017) 873–876.
[22] Y. Lai, Y. Chen, H. Chen, In-Network execution of external join for sensor networks, in: Web-Age Information Management, 2008. WAIM'08. The Ninth International Conference on, IEEE, 2008, pp. 78–85.
[23] Y. Xu, S. Helal, M. Scmalz, Optimizing push/pull envelopes for energy-efficient cloud-sensor systems, in: Proceedings of the 14th ACM international conference on Modeling, analysis and simulation of wireless and mobile systems, ACM, 2011, pp. 17–26.
[24] A. Silberstein, Push and pull in sensor network query processing, in: Southeast Workshop on Data and Information Management (SWDIM06), Raleigh, North Carolina, 2006.
[25] F. Bai, B. Krishnamachari, Exploiting the wisdom of the crowd: localized, distributed information-centric vanets, IEEE Commun. Mag. 48 (5) (2010).
[26] Q. Zhao, Y. Zhu, C. Chen, H. Zhu, B. Li, When 3g meets vanet: 3g-assisted data delivery in vanets, IEEE Sens. J. 13 (10) (2013) 3575–3584.
[27] M. Eltoweissy, S. Olariu, M. Younis, Towards autonomous vehicular clouds, in: International Conference on Ad Hoc Networks, Springer, 2010, pp. 1–16.
[28] K. Kai, W. Cong, L. Tao, Fog computing for vehicular ad-hoc networks: paradigms, scenarios, and issues, J. China Univ. Posts Telecommun. 23 (2) (2016) 56–96.
[29] Y. Lai, L. Zhang, T. Wang, F. Yang, Y. Xu, Data gathering framework based on fog computing paradigm in vanets, in: Asia-Pacific Web (APWeb) and Web-Age Information Management (WAIM) Joint Conference on Web and Big Data, Springer, 2017, pp. 227–236.
[30] Y.C. Hu, M. Patel, D. Sabella, N. Sprecher, V. Young, Mobile edge computinga key technology towards 5g, ETSI White Paper 11 (11) (2015) 1–16.
[31] F. Bonomi, R. Milito, J. Zhu, S. Addepalli, Fog computing and its role in the internet of things, in: Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing, ACM, 2012, pp. 13–16.
[32] Y. Lai, F. Yang, J. Su, Q. Zhou, T. Wang, L. Zhang, Y. Xu, Fog-based two-phase event monitoring and data gathering in vehicular sensor networks, Sensors 18 (1) (2018).
[33] D. Guo, X. Zhu, H. Jin, P. Gao, C. Andris, Discovering spatial patterns in origin-destination mobility data, Trans. GIS 16 (3) (2012) 411–429.
[34] H. Jeung, M.L. Yiu, X. Zhou, C.S. Jensen, Path prediction and predictive range querying in road network databases, VLDB J. 19 (4) (2010) 585–602.
[35] T.H. Cormen, Chapter 11: Hash Tables, Introduction to Algorithms, MIT press, 2009.
[36] A. Keränen, J. Ott, T. Kärkkäinen, The ONE Simulator for DTN Protocol Evaluation, in: SIMUTools '09: Proceedings of the 2nd International Conference on Simulation Tools and Techniques, ICST, New York, NY, USA, 2009.
[37] M.E. Newman, Power laws, pareto distributions and zipf's law, Contemp. Phys. 46 (5) (2005) 323–351.
[38] M. Dighriri, A.S.D. Alfoudi, G.M. Lee, T. Baker, R. Pereira, Comparison data traffic scheduling techniques for classifying qos over 5g mobile networks, in: Advanced Information Networking and Applications Workshops (WAINA), 2017 31st International Conference on, IEEE, 2017, pp. 492–497.



Yongxuan Lai received the PhD degree in computer science from Renmin University of China in 2009. He is currently an associate professor in Software School, Xiamen University, China. His research interests include network data management, vehicular ad-hoc networks, big data management and analysis. He is an academic visiting scholar during Sep. 2017–Sep. 2018 at Data and Knowledge Engineering (DKE) Group, University of Queensland, Australia.

Hailin Lin is an undergraduate student in the Department of Software Engineering at Xiamen University, China. His research interest includes vehicular networks and data mining using trajectories.

Fan Yang received the Ph.D. degree in control theory and control engineering from Xiamen University, Xiamen, China, in 2009. He is currently an associate professor in the Institute of Pattern Recognition & Intelligent Systems, Department of Automation at Xiamen University. His current research interests include mobile computing, pattern recognition, data mining and bioinformatics. He has published more than 50 journal articles and conference papers.

Tian Wang received the B.Sc. and M.Sc. degrees in computer science from Central South University in 2004 and 2007, respectively, and the Ph.D. degree from the City University of Hong Kong in 2011. He is currently a Professor with National Huaqiao University, China. His research interests include wireless sensor networks, fog computing, and mobile computing.