Distributed Collaborative Execution on the Edges and Its Application to AMBER Alerts

Qingyang Zhang, *Student member, IEEE*, Quan Zhang, Weisong Shi, *Fellow, IEEE* and Hong Zhong

***Abstract*—In the Internet of Everything (IoE) era, billions of geographically distributed things will connect to the Internet and generate hundreds of zettabytes of data per year. Pushing that data to the cloud requires tremendous network bandwidth cost and latency. This is too onerous for some latency-sensitive applications, such as vehicle tracking using city-wide cameras. One application currently limited by such obstacles is the AMBER Alert system–but edge computing could transform this system's capabilities. Edge computing is a new computing paradigm that greatly diminishes data transmission and response latency by processing data at the proximity of data sources. However, most vision-based analytics are compute-intensive, and an edge device might be overwhelmed given tens of frames each second for real-time analysis. Also, the system needs a customized and flexible interface to implement efficient tracking strategies. To meet these needs, here we extend a big data processing framework, called *Firework*, to support collaboration between multiple edge devices and customizable task-scheduling strategies. Based on this extended version of *Firework*, we implement the AMBER Alert Assistant (A3), which efficiently tracks and locates a vehicle by analyzing city cameras' data in real time. We also propose two kinds of customized task-scheduling algorithms for vehicle tracking in A3. Comprehensive evaluation results show that A3 achieves real-time video analytics by collaborating among multiple edge devices; and the proposed location-direction-related diffusion strategy effectively controls the searching area for vehicle tracking by smartly selecting candidate cameras.**

***Index Terms*—edge computing; AMBER alert; public safety; video analytics.**

I. Introduction

IN the last decade, researchers and practitioners have treated cloud computing [1] as the de facto large-scale data processing platform. Numerous cloud-centric data processing platforms [2]–[7] that leverage the MapReduce [8] programming framework, have been proposed for both batched and streaming data. In recent years, we have witnessed the onset of the Internet of Everything (IoE) era [9], where billions of geographically distributed sensors and actuators are connected and immersed in our daily life. As one of the most sophisticated IoE application, real-time video analytics promises to significantly improve public safety. *Video Analytics* leverage information and knowledge from video data content to address a particular applied information processing need; and as the public safety community massively adopts this technology, it provides near real-time situational awareness of citizens' safety and urban environments, including automating the laborious tasks of monitoring live video streams, streamlining video communications and storage, providing timely alerts, and making the task of searching enormous video archives tractable [10]. However, the conventional cloud-centric data processing model is inefficient to process all IoE data in data centers especially for the response latency, network bandwidth cost, and possible privacy concerns, given the zettabytes of data generated by edge devices [11], [12]. On the other hand, rarely are data shared among multiple stakeholders–a variety of concerns restrict video analytics' practical deployment, even though video data analytics could take advantage of many data sources to make a smart decision. Moreover, there is no efficient data processing framework for the community to program and deploy easily for public safety applications across geographically distributed data sources.

Edge computing could serve as a major boon to transform this technology, however. The emerging field of edge computing (also known as fog computing [13], mobile edge computing [14] or Cloudlet [15]–[20]) refers to "the enabling technologies allowing computation to be performed at the edge of the network, on downstream data on behalf of cloud services and upstream data on behalf of IoE services" [21]. It complements the existing cloud computing model and enables data processing at the proximity of data sources, which is promising for latency-sensitive applications that leverage computing resources at the close edge instead of in the remote cloud.

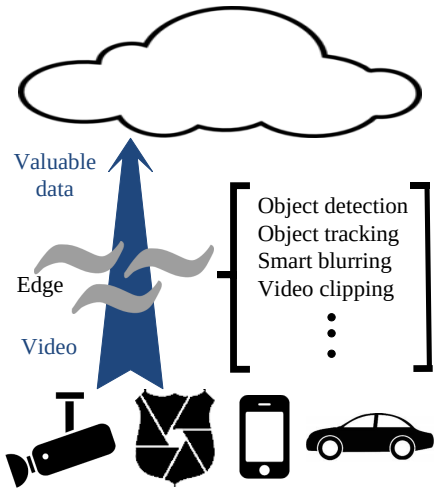


Fig. 1. Edge video analytics for public safety.

Qingyang Zhang and Hong Zhong are with the School of Computer Science and Technology, Anhui University, Hefei, China, 230601.
E-mail: qyzhang@wayne.edu, zhongh@ahu.edu.cn
Quan Zhang and Weisong Shi are with the Department of Computer Science, Wayne State University, Detroit, MI, U.S.A., 48202. Qingyang Zhang was a visitng student at Wayne State University while working on this project.
E-mail: quan.zhang, weisong@wayne.edu
Manuscript received December 1, 2017; accepted May 31, 2018.

As Fig. 1 shows, given the edge nodes deployed along the data propagation path between cameras and the cloud, we can process the partial (for example, license plate extraction, feature extraction, smart blurring, and video clipping) or entire workload as the video stream arrives, which significantly reduces the video data transmission's size, and consequently the network bandwidth cost. Then we send the semi- or fully-processed data to the cloud for aggregation-based analytics. Moreover, we can share the services deployed at the edge nodes and intermediate data generated by the edge nodes with other stakeholders, and then compose a customized video analytics application by leveraging these existing intermediate data/services.

Beyond processing data in real time, this distributed and collaborative application in an edge-cloud environment also offers increased reliability, leading to a quicker response. Take, for example, the America's Missing Broadcast Emergency Response (AMBER) alert system. Right now, tracking a suspect's vehicle heavily relies on the reports of witnesses. But what if we leveraged edge computing instead, so a license plate-recognition (LPR) application running city-wide security cameras could significantly improve the efficiency of suspect vehicle tracking? Then, the local license plate-recognition process for an image could respond in milliseconds without data transmission, instead of waiting for seconds to communicate with a remote data center, which is far quicker than the time required to send the image to the data center, process the data, and retrieve the results. The potential consequence is stark; this could mean the difference between identifying the missing child immediately versus losing the child in sight. Moreover, the network traffic caused by sending a deluge of data to a data center significantly impacts network performance, which further intensifies the response latency and data transmission cost.

To realize the vision of edge computing and real-time video analytics for public safety, we must tackle several barriers systematically: First, there is no existing programming tool and framework that allows programmers to build cost-effective real-time applications among various geographically distributed data sources. Second, most video analytics algorithms undoubtedly are computationally intensive, of which hundreds of milliseconds might be taken to process one video frame on edge nodes [22], so that the edge device is overwhelmed given the tens of frames in a second. Thus, the collaboration among multiple edge nodes would ensure high resource usage and provide opportunities to balance the workload. However, realizing efficient resource management and task scheduling are difficult in an edge computing environment, where the edge nodes primarily are heterogeneous, with various types of network connectivity. Third, efficient task scheduling might require user intervention, in which domain knowledge could be applied to boost the performance. However, rarely is user intervention considered in task allocation, especially when the application-defined network topology differs from the physical network topology.

To tackle the aforementioned issues, we extend our previous work on *Firework* [23], [24].Specifically, we implement a collaborative mechanism among multiple edge nodes for real-time data processing and a programming interface to construct a customized task-scheduling strategy, depending on the application-defined network topology. Based on *Firework*'s extensions, we implement the *AMBER Alert Assistant* (A3) system, which aims to improve target tracking efficiency (that is, tracking suspect's vehicles) using the *AMBER Alert* system. By implementing multiple customized task-scheduling strategies, we evaluate various target-tracking strategies relying on the status of real-time video analysis results.

The rest of the paper is organized as follows. We describe our motivation in Section II and present the *Firework*'s extensions in Section III. We discuss the proposed A3 system in Section IV. Section V focuses on experiments and results. We review related work in Section VI. Finally, we conclude in Section VII.

II. Motivation

A. AMBER Alert

The *AMBER Alert* is a system that alters the public to child abductions that is implemented with different names in different countries [25]. In the United States, when a kidnapping occurs, an alert is sent to citizens' smart phone immediately if they are near the event location. The alert usually includes descriptive information about this event, such as time, location, the kidnapper's vehicle license plate number, and a description of the child or kidnapper. Then, witnesses can provide pertinent information to the police department if they spot the kidnapper's vehicle on the road. But, tracking a suspect's vehicle heavily relies on the reports of witnesses, which is inefficient, because many people miss the alert or might not recognize the suspect's vehicle.

Nowadays, video surveillance is common in cities, including security cameras, traffic cameras, and even smartphone/on-dash cameras. For example, participants in Project Green Light [26] provide their video data for public safety, which includes more than a hundred cameras in Detroit, Michigan. Using an automatic license plate-recognition (ALPR) technique, video surveillance greatly improves the tracking of kidnappers' vehicles. Coped with the privacy issues, other cameras (i.e., the cameras in the Project Green Light, on-dash car cameras, smart phone cameras) might provide video data by the owners.

Usually, the network bandwidth requirements of a 720P, 1080P, and 4K live video are 3.8 Mbps, 5.8 Mbps, and 19 Mbps, respectively. Thus, pushing all video data to the cloud leads to huge data transmission costs and high latency. Considering the amount of data generated by many cameras simultaneously, cloud-based solutions are no longer suitable for real-time video analytics, due to high data transmission costs, bandwidth requirements, and onerous latency. Edge computing processes data locally, which significantly reduces the data transmission cost and lowers network bandwidth requirements. But it still has two barriers that prevent real-time vehicle tracking in the *AMBER Alert* system:

1) limitations of edge devices: Most computer vision algorithms are computationally intensive, such as object detection, face recognition, and optical character recognition (OCR).

TABLE I
PROCESSING TIME FOR PLATE RECOGNITION IN THE VIDEO.

Edge Device	**Video Decoding** (ms)	**Motion Detection** (ms)	**Plate Detection** (ms)	**Plate Recognition** (ms)
Amazon EC2 t2 node	4.83	53.35	136.31	16.92
Dell Inspiron 5559	3.76	42.07	121.40	13.92
Dell Wyse	14.35	158.75	601.32	53.80
Dell OptiPlex	3.51	38.75	95.67	12.47

Thus, the edge node might lack the computation resources needed to process video in real time. For vehicle tracking, an open source ALPR system is built, called OpenALPR [27], which usually has two stages: license plate detection and license recognition. The latter usually employs the OCR technique. Because of its video streaming features, it is worthwhile to consider video decoding; we also implemented motion detection using OpenCV [28], which detects the different areas between two frames and avoids needless license plate detection and recognition. Here, we measure OpenALPR's [27] performance on different devices, including the Amazon Elastic Compute Cloud t2 node (Amazon EC2 t2: an Amazon virtual machine with Intel Xeon CPU at 2.4 GHz), Dell Inspiron 5559 (a laptop with Intel i7-6500U at 2.5 GHz, going up to 3.1 GHz), Dell Wyse (a home gateway with Intel N2807 at 1.58 GHz) and Dell OptiPlex (an Intel i5-4590 at 3.3 GHz, going up to 3.7 GHz).

Fig. 2 shows that an ALPR system without motion detection will cost much more than the one with motion detection executing on an Amazon EC2 t2 node. This is because motion detection marks the moving area between two frames, which significantly reduces the workload of the latter steps by reducing the recognition area. Table I shows the time costs on different steps of video analytics, in which the license plate detection contributes to the majority of processing time, and none of these devices can analyze video in real time if only one thread is used. Note that the average LPR time without motion detection is 191.05 ms, and the time with motion detection is 131.18 ms on an Amazon EC2 t2 node. The average ALPR's complete processing times for two cases are 195.69 ms and 187.59 ms. The difference between two cases is small, which is why we use video from a peak period. We also use another set, where the time is less than 100 ms for an instance with motion detection. In this paper, we always use peak-period video data, because it is important to consider the worst case to avoid overload. Thus, video analytics must be multithreaded and have motion detection; both are included in our system design. Also, because LPR takes much less time to process than license plate detection, here we simply use LPR rather than the full steps of license plate detection and recognition.

2) Control of the vehicle tracking area: As the suspect's vehicle moves, the vehicle tracking area should enlarged; and it also should shrink (or close in on) the area once the vehicle is found. A simple method to track a vehicle is to enlarge the radius of the area as time goes by. However, the speed of different routes varies. For example, the highway allows twice the speed of city streets. Thus, it is unreasonable to set the same rate for different types of cameras. Furthermore, mobile cameras deployed on taxis and/or garnered by crowdsourcing

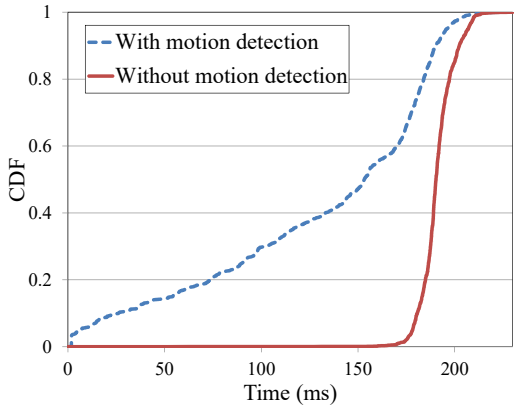


Fig. 2. The time cumulative distributive function (CDF) of license plate detection and recognition on an Amazon EC2 t2 node.

private vehicles could be integrated into the system to provide more video sources. It is exceedingly difficult, though, to provide a customized and dynamic control strategy for tracking the suspect's vehicle using both static and dynamic data provided by moving cameras.

B. Firework

To contend with these issues, we extend the *Firework* framework. *Firework* helps build an enhanced *AMBER Alert* system on edge nodes and the cloud. However, previous versions of *Firework* did not work well in the instances noted (see [24] for an overview of the previous version). Thus, we extend the framework in section III, and introduce the previous version's architecture while also detailing what we extend.

Firework is a framework for big data processing and sharing among multiple stakeholders in a collaborative edge-cloud environment. *Firework* provides an easy-to-use programming interface to develop and deploy applications to edge nodes. It also allows splitting the whole service into several subservices. For example, an ALPR service includes a video decoding subservice, motion detection subservice, and license plate number recognition subservice.

We first introduce the terminologies that describe abstract concepts in *Firework*. Based on existing definitions from prior work [24], we extend and enrich their meanings and summarize as follows:

- *Firework*. This is an operational instance of the *Firework* paradigm. A *Firework* instance might include multiple *Firework.Node*s and *Firework.Manager*s, depending on the topology. Fig. 3 shows an example of *Firework*

instance consisting of five *Firework.Node*s and one *Firework.Manager* employing heterogeneous computing platforms. We do not distinguish the host of *Firework.Node*s and *Firework.Manager*. If all firework nodes host on edge devices, it will be a complete edge computing platform, and if they host on cloud virtual machines, it will be a complete cloud platform. Also, some firework nodes are able to host on edge devices and some on cloud virtual machines, thereby forming an edge-cloud collaborative platform.

- *Firework.View*. The *Firework.View* is defined as a combination of a *dataset* and *function*s, which is inspired by object-oriented programming. The *dataset* describes the data generated by edge devices and historical data stored in the edge or cloud. The *function*s define applicable operations upon the dataset. A *Firework.View* could be adopted by multiple data owners who implement the same functions on the same type of dataset.
- *Firework.Node*. A *Firework.Node* is a device that implements *Firework.View*s, and it allocates computation resources including the CPU, memory, and network to run (sub)services. Note that the (sub)service is the worker running on a *Firework.Node*. A *Firework.Node* might have only data producers, such as sensors and mobile devices, which publish sensed data without analyzing. While it inherits *Firework.View*s, it is a data consumer. In this case, it uses others' functions as data sources. Here, a *Firework.Node* is a data producer and a data consumer simultaneously–so it senses the data and provides the processed data.
- *Firework.Manager*. This first provides centralized service management to oversee registered *Firework.View*s. All *Firework.View*s implemented by *Firework.Node*s should register to the *Firework.Manager*. It also manages the deployed services built on top of these views. Second, it serves as the job tracker that dispatches tasks to *Firework.Node*s.[1] It will monitor *Firework.Node*s' computation resources and optimize the running (sub)services via dynamic workload balancing among multiple *Firework.Node*s, depending on resource usage and user intervention. Third, it exposes available services to users so that they can leverage existing services to compose their own applications.

C. Distributed Collaborative Execution on the Edge

As mentioned previously, there are two barriers to combining edge computing with the *AMBER Alert* system: the limitations of edge devices, and control of the vehicle tracking area. These can be abstracted to two types of collaborations–the collaboration of data processing, and the collaboration of task diffusion. This leads to the requirements for a distributed collaborative execution. Similar requirements exist in other edge computing applications. For example, consider activity detection in a jail. By analyzing captured videos, the threatening event can be detected using several features–such as the same people crowding in several different videos. This scenario involves synthetically analyzing multiple video data with different times to find a suspected event, which also needs to combine multiple edge nodes and the task control on multiple edge nodes. However, no platform or framework currently exists that can cope with both barriers. The *Firework* framework might be approximated for this case, and it provides a uniform programming interface to develop applications in the collaborative edge-cloud environment.

[1]A service is from the users' view, and one service can be split into several subservices. The job and task are from the view of scheduling. A job implements a specific function of *Firework.View*, and provides that service for users. A job might consist of several tasks that only implement a subservice.

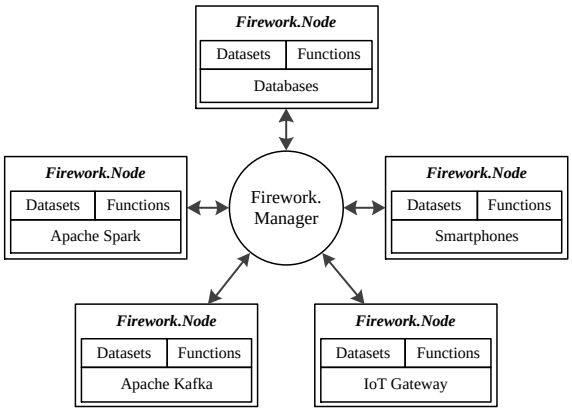


Fig. 3. A high-level overview of *Firework*.

III. Extended *Firework* System Design

Now that we have explored the motivations for this work, we present our design. We primarily focus on introducing *Firework*'s novel extensions. Then, we overview the prototype implementation and introduce several important features.

A. System Architecture

To support the aforementioned features, the previous *Firework* version is extended. As with the previous version of *Firework*, a three-layer architecture is designed (see Fig. 4), which consists of *Service Management*, *Job Management*, and *Executor Management*. The *Service Management* layer performs service discovery and deployment, and a new module called *Access Control* is added for providing a uniform access point to other nodes in *Firework*. The *Job Management* layer manages services running on a computing node. The functions of the three original modules are merged into a new *Task Dispatch*, and two new modules called *Task Monitor* and *Job Schedule* are added. The *Executor Management* layer manages computing resources.

Because the steps deploying services have been introduced in previous work, we only discuss them briefly here. Then we introduce each layer in this new, extended version of *Firework*.

To deploy a service on *Firework*, a user must implement at least one *Firework.View*, which defines the shared data and functions, and a deployment plan, which describes how computing nodes are connected and how services are assigned to the computing nodes. Upon registering a service (i.e., *Firework.View*), *Firework.Manager* creates a service stub for

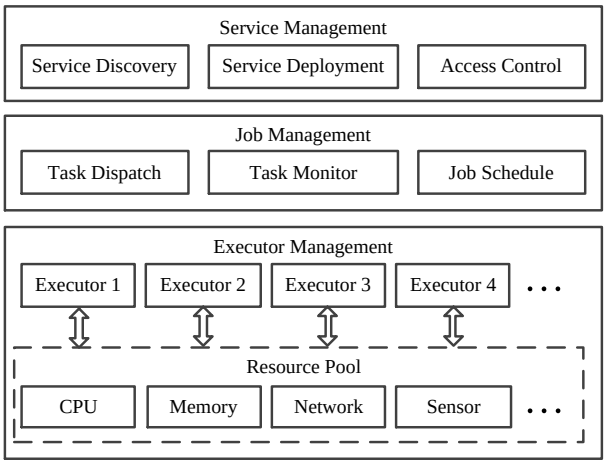


Fig. 4. *Firework*'s new architecture, including its extensions.

that service (note that the same service registered by multiple nodes shares the same service stub entry), which contains the metadata to access the service, such as the network address, functions' entries, and input parameters. To take advantage of existing services, a user retrieves the list of available services by querying *Firework.Manager*. Then, by using a service's metadata, any *Firework.Node* can access the service. Based on combining different services, an application can be built.

1) Service Management: The *Service Management* layer consists of three components: *Service Discovery*, *Service Deployment*, and *Access Control*. The *Service Discovery* module stores and publishes the metadata of registered (sub)services, and it also discovers other *Firework.Node*s' published services. It is similar to the *Discovery* of the *Service Management* layer in the previous version of *Firework*. The difference between the two versions of *Firework* is that the new, extended *Firework* supports a distributed service discovery. That means that each *Firework.Node* plays a *Firework.Manager* role, and all *Firework.Node*s in a *Firework* platform are able to discover one another. The reason we do that is to make our extended *Firework* a completed distributed structure. The *Service Deployment* distributes an application to edge nodes according to the application-defined network topology. Note that the application-defined deployment topology might be different from the underlying network topology. The reason *Firework* provides a customizable deployment plan is to avoid redundant data processing and facilitate application-defined data aggregation.

The new module *Access Control* manages the connections with other *Firework.Node*s and determines the accessibility of registered services on a *Firework.Node*. As mentioned previously, the job might be computationally intensive, making it difficult for a *Firework.Node* to process in real time. Assuming that a service is split into two subservices, part of the output stream of the first subservice will be transferred to the input stream of the second subservice hosted on the other *Firework.Node*. In this case, the data will be transmitted via the *Access Control* module. This module also provides authentications and ensures security between different *Firework.Node*s.

2) Job Management: In the *Job Management* layer, we introduce three major components, including the *Task Monitor*, *Task Dispatch*, and *Job Schedule* to manage tasks on a *Firework.Node*. The input/output data of a subservice are streamed as message queues. Thus, a service's different subservices will be in series as a message queue, which is the upriver subservice's output message queue and the downriver subservice's input message queue. In our extended *Firework*, we reuse this structure and extend it. A subservice launches several instances to implement a multithread subservice and these instances subscribe with the same message queue. Besides, the collaboration of different edge nodes is achieved by connecting message queues on different *Firework.Node*s. For example, one *Firework.Node*, which receives video data and launches the video analysis service, serves as the master node, and the other collaborative *Firework.Node*s for the same service will serve as work nodes. In this case, each work node will create a local message queue and subscribe message (task) from the master node's message queue. Thus, the master node can offload task to collaborative work nodes. The *Task Monitor* collects task-related performance metrics (such as message queue length, CPU usage, memory usage, network bandwidth, and response latency). Based on those performance indicators, a *Firework.Node* automatically scales instances of one subservice or migrates tasks to other edge nodes when the *Firework.Node*'s related message queue is too long to finish all of them without violating the response latency requirement. The task migration is carried out by the *Task Dispatch*, which chooses target edge nodes depending on the cost of task migration (for example, resource usage on the target edge nodes, the cost to transfer data, and the cost of retrieving results). The *Task Dispatch* also issues service requests to certain edge nodes when a local service relies on other services hosted by these edge nodes. The *Job Schedule* provides a customizable job-scheduling interface for users, so that they can intervene when a job is scheduled to execute. A customizable scheduling interface enhances *Firework*'s flexibility in providing job/workload offloading. A user can determine the job/workload offload/migration according to different metrics or optimization goals. For example, in our application A3, it assigns a `timeout` value (also defined in Listing 1), and will offload the jod to neighbouring *Firework.Node*s to search the suspect's vehicle after the local job is overtime. An application can also dynamically determine which new jobs to invoke, when to invoke these jobs, and where to run these jobs, depending on the application's current status. A user can send *Firework.Manager* a request to start a job. Upon receiving the invocation, a local job is created by the *Job Management* layer, which initializes the task locally. Next, *Firework.Manager* forwards the request to involve *Firework.Node*s, which implements the *Firework.View* of the requested task. Finally, the task is added to task queue waiting for execution. When the user terminates a task, the *Job Management* layer stops executors and releases that job's dedicated port.

3) Executor Management: A service in *Firework* runs on an executor that has dedicated CPU, memory, and network resources. *Firework.Node*s leverage heterogeneous computing platforms and consequently adopt different resource manage-

ment approaches. Therefore, the *Executor Management* layer serves as an adapter that allocates computing resources to a task. Specifically, some *Firework* nodes such as smartphones or smart access points (AP) may adopt a Java virtual machine (JVM) or Docker [29], while some nodes (such as commodity servers) may employ OpenStack [30] or VMWare [31] to host an executor.

B. Prototype Implementation

Based on the code from the previous version of *Firework*, we implemented a prototype of the extended *Firework* using Java. In the *Service Management* layer, *etcd* [32] is used to perform the service registration, which is a distributed, consistent key-value store for shared configuration and service discovery. In the future, we will aim for a more lightweight version of this function. We use a registered service's metadata to access the service stored in *etcd*, so users can access the list of available services via *etcd*'s RESTful interface. As we mentioned, an application-defined topology is provided to customize the application-deployment plan. Fig. 5 shows an example where the underlying network topology differs from the application-defined topology. To describe an application-deployment plan, we use a JavaScript Object Notation (JSON) structure [33], which is a lightweight data-interchange format and is completely language-independent, to represent the deployment plan's features. Listing 1 shows an example of the application-deployment plan (including the service configuration and application-defined topology) for the laptop in Fig. 5. In the JSON structure in Listing 1, an application defines different topologies with their metadata for the video searching service, motion detection service, and plate recognition service. According to the deployment plan, the motion detection service on the laptop in Fig. 5 receives real-time video data from camera 1 via real-time transport protocol (RTP) and detect motion area. Then, the plate recognition service on the laptop will dispatches part of tasks to the smart AP and tablet. Moreover, the laptop also migrates the video searching task to the desktop after a time threshold thus the later can recognize plate numbers in the video of camera 2, which are connected by a blue line. The application-deployment plan is used by the *Task Dispatch* and *Job Schedule* to assign or migrate tasks, depending on scheduling strategy. Note that the *timeout* value is also used for the application A3, and we will introduce in Section IV-D.

Listing 1. An example of an application-defined topology.

```
{ "id": 6,
  "topology": [
    { "service_name": "video_searching",
      "service_id": 1,
      "node": [
        { "id": 4,
          "ip-port": "192.168.1.40:10001",
          "timeout": 30 } ]
    },
    { "service_name": "motion_detection",
      "service_id": 2,
      "node": [
        { "id": 2,
          "ip-port": "192.168.1.20:10001",
          "protocol": "rtp" } ]
    },
    { "service_name": "plate_recognition",
      "service_id": 3,
      "node": [
        { "id": 1,
          "ip-port": "192.168.1.1:10001"
        },
        { "id": 5,
          "ip-port": "192.168.1.50:10001" } ]
    } ]
}
```

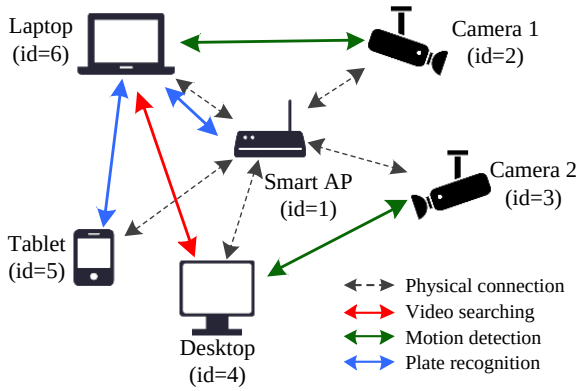


Fig. 5. Two different topologies defined by an application.

The *Access Control* module manages the network communication for services between *Firework.Node*s. A whitelist-based solution is implemented in the access control model, where a *Firework.Node* only accepts a connection from the nodes listed in the application defined topology and the remaining connections are rejected. A *Firework.Node* could accept any connection or connections from authenticated edge nodes based on secure authentication mechanisms. For future work, we will implement complex (but secure) access control mechanisms, such as attribute-based encryption-based mechanisms [34], [35].

The *Job Management* layer manages all the tasks running on an edge node and dispatches tasks/workloads to the collaborative edge nodes, which are defined in the application topology. When receiving a task, the *Task Dispatch* module analyzes the received job to determine the task's dependent subservices. If dependent subservices are not running, the *Task Dispatch* module will launch the subservices. After all dependent subservices are launched, the job will be added to the job queue. If a dependent subservice is requested by multiple jobs, *Firework* reuses the dependent subservice by sharing the data input/output streams among multiple jobs. The *Task Monitor* module collects resource utilization of a *Firework.Node*. Specifically, it uses a message queue's length to trigger task offload via *Task Dispatch* when the length exceeds a threshold predefined by the application. The *Job Schedule* by default uses a first-come-first-serve policy for scheduling user requests. As we mentioned, a user can customize the scheduling policy of his/her own application without affecting the fairness among multiple users. A customized job scheduling strategy migrates a waiting job to other edge nodes depending on the configurations (e.g., the job queue's length, or the timeout threshold) of the *Service Deployment*.

Taking the video analytics case as an example, we will describe the usage of message queues in our prototype. As

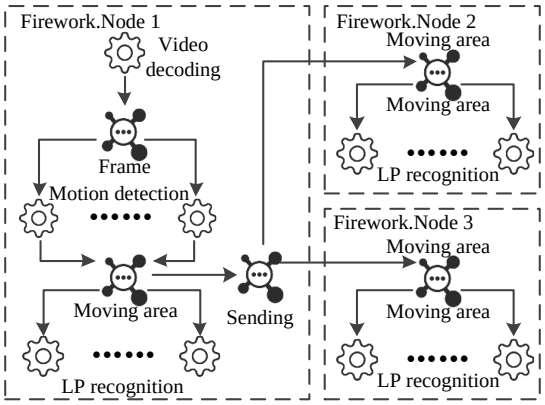


Fig. 6. An example for using message queues in the *Firework* prototype.

Fig. 6 shows, three *Firework.Node*s exist, and several message queues connect these *Firework.Node*s and the workers. According to the results of Table I, video decoding costs several milliseconds, and motion detection costs 38 ms at least. Thus, when using only one worker, video is easy to decode in real time, but motion detection poses difficulty in real time. Several worker instances are launched, then, to detect motion areas empirically. Each frame will be saved into a message queue and wait for free motion-detection worker processing. The output streams (referring here to moving area images) of motion-detection workers are connected with the input streams of license plate-recognition workers by a message queue. Hence, the instances of license plate-recognition worker auto scale, and a physical device always has an upper limitation. Thus, a sending queue–managed by *Access Control*–is connected with other *Firework.Node*s' moving area image queues (e.g., *Firework.Node*s 2 and 3 in Fig. 6). These queue will send the image to other nodes for collaborative processing between different *Firework.Node*s.

To implement the *Executor Management* layer, we use JVM as an executor and message queue to manage input/output data. Once a task is scheduled to execute, one or more JVM are allocated for the task. Note that other resource virtualization tools (such as OpenStack [30] or Docker [29]) can also serve as an executor manager by adding a corresponding adapter in *Firework*.

In addition, all modules in *Firework* are componentized, and they use message queues to communicate with each other, which is a inter-thread communication provided by message queue. Besides, the message queue system also enables inter-process communication, which allows message passing between different processes that host within the same device or different devices. Thus, we design such a message queue -based architecture, which is much easier for extending *Firework* modules in the future. Besides, the *Task Dispatch* could simply offload to other edge nodes by manipulating the data in message queues (e.g., forwarding input data directly to the input data queue of other edge nodes without any data manipulation), and it auto-scales easily. Moreover, it allows heterogeneous edge nodes to adapt the *Firework* framework by employing any message queue, which is more suitable considering the hardware and software constraints. For example, Message Queuing Telemetry Transport (MQTT) [36] may be more efficient than Apache Kafka [37] for certain Internet of Things (IoT) devices with limited hardware resources. Furthermore, the developers can focus on the implementation of user-defined functions while the data sharing can be easily achieved by a message queue.

In our prototype, we use Apache Kafka as an external message queue system, which connects not only different *Firework.Node*s, but also different modules within the same *Firework.Node*. The reason we use Apache Kafka is that most of the devices mentioned in Project Green Light can handle the cost of an Apache Kafka system, and our system can also use a more suitable message queue system for special scenario, easily. For example, in the scenario that the cloud is needed to store and analyze historical IoT data, a better solution is using an applicable message queue system for each subsystem. In a water quality monitoring project, all sensors, as *Firework.Node*s, upload the data to the nearest edge server connecting via an MQTT-based message queue system. And the analysis results can be shared by edge servers via an Apache Kafka message queue system. In the future, considering various scenarios, we plan to implement an elastic message queue module, which will launch different message queue models dynamically. For example, it will launch an MQTT-based or other lightweight message queue system for resource-limited device, such as IoT devices and smartphones. And it also will launch an inner message queue connecting different modules within the same *Firework.Node*, which would be more lightweight without requiring extra processes.

IV. AMBER Alert Assistant

After extending the previous version of *Firework*, we implement an application called *AMBER Alert Assistant* (A3). After receiving an AMBER alert from police, A3 automatically tracks the suspect's vehicle by analyzing the video data of city-wide cameras and it also automatically controls the tracking area. In this section, first we consider A3's potential use and application design. Then, we implement the application and discuss two task-scheduling strategies, which are used to control the vehicle tracking area. Note that here, a *Task* refers to a vehicle-tracking *Job*, and when a *Task Receiver* receives a *Task*, it creates a *Job* and launches related subservices.

A. Application Scenario

In A3, we consider an edge computing network model (see Fig. 7) inspired by Project Green Light [26], in which cameras are located at gas stations or shops and most of them connect to several edge devices, such as desktops. We also consider mobile cameras in A3, such as car's dash cameras, where the number of mobile cameras will increase quickly with the rise of autonomous driving. Each road camera and its edge nodes connect to a router via wire or wireless links, where the router is used to connect with a wide area network (WAN), and mobile cameras take part in A3 by connecting to wireless cellular network. Once the edge node receives an alert from police, it will automatically pull the video from a connected camera and then collaboratively analyze video

data with other local edge nodes in real time. As time passes, the edge node will extend the searching areas using a task-scheduling strategy. When the targeted vehicle is found, the edge node will automatically shrink the searching area to minimize resources, including energy.

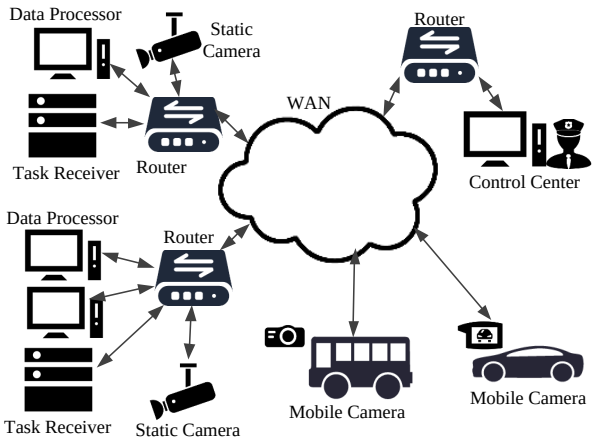


Fig. 7. The network settings of an A3 application scenario.

B. Application Design

Based on A3's network, we define three types of devices in accordance with their functions in A3: the *Control Center*, *Task Receiver*, and *Data Processor*. These three types of devices are implemented based on our extended version of *Firework*, and we implement different *Firework.View*s for different functions.

1) Control Center: A *Control Center* is used by police to publish AMBER alerts to *Task Receiver*s and collect the reports from *Task Receiver*s. It configures a customized task scheduling by defining an application-defined topology. After starting a vehicle tracking task, it will receive regular status reports from a working node. Thus, it knows the current searching area, updated periodically. Once the targeted vehicle is found, it will receive the report from the related *Task Receiver*. After arresting the kidnapper, it clears the alert for all *Task Receiver*s.

2) Task Receiver: The video data captured by a traffic camera will be pulled and analyzed by a local *Task Receiver*. As an initial operation, it will get the application-defined topology from the *Control Center*, and store it in the *Service Deployment* module using JSON format. As Listing 1 shows, for a video searching service, it has a field called *timeout*, which it uses to control the opportunity for transferring the task to an appointed edge node. It also defines a topology for the *Data Processor*, to collaboratively analyze video in real time. According to the values watched by the *Task Monitor* (e.g., the message queue's length), a *Task Receiver* dispatches part of the video analytics workloads (such as plate recognition) to the *Data Processor*s defined in the application-defined topology. Once the *Task Receiver* get a report from others or a stop signal from the *Control Center*, it will stop analyzing the video and clear the message queue waiting for processing.

Listing 2. Example of the *Control Center*'s interface.

```
/* ControlCenter: the interface should be implemented on
     the edge node located at the police department */
public interface ControlCenter_I{
  /* Send Video Searching Task to edge nodes */
  public void PublishAMBERAlert(List<int> node_list,
       JsonString vehicle_info);
  /* Stop Video Searching Task */
  public void ClearAMBERAlert(int task_id);
  /* Report object location */
  public void TaskReport(String task_report);
  /* Set the application defined topology */
  public void SetTopology(JsonString whole_topology);
  /* Get the application defined topology */
  public JsonString GetTopology(int node_id);
  /* Report status from Task Receiver */
  public void ReportStatus(JsonString status);
  /* Login for mobile cameras */
  public Byte[] MobileCameraLogin(Byte[] data);
  /* Logout for mobile cameras */
  public Byte[] MobileCameraLogout(Byte[] data);
}
```

Specifically, it will forward the report to the *Task Receiver* that originally transferred this task.

3) Data Processor: The *Data Processor* in A3 only provides services to analyze the video data, but it is necessary for real-time video analytics. This type of edge node includes any *Firework.Node*-hosted local device, such as a desktop, laptop, and even smartphone. After accessing the same network with the *Task Receiver*, it gets the sub video analytics task from the connected *Task Receiver*.

Note that the connections between edge nodes are defined in the *Service Deployment* module, and controlled by the *Access Control* module. This means that the connection from an unknown edge node (that is, a node not defined in an application-defined topology) will be rejected. In Fig. 7, the police's desktop is a *Control Center* node, and other devices are either a *Task Receiver* or a *Data Processor*, where only one *Task Receiver* exists.

C. Implementation Details

As we mentioned, we implemented three types of *Firework.Node*s: *Control Center*, *Task Receiver*, and *Data Processor*. The *Task Receiver* and *Data Processor* provide (sub)services about video analytics. In this section, we will use the Java interface to describe A3's different services. Note that we use the word "interface" instead of "(sub)service".

Listing 2 illustrates the services we implemented in this type of *Firework.Node*. The `PublishAMBERAlert` interface and `ClearAMBERAlert` interface allow the police to publish or clear an alert to the related *Task Receiver*. Once an edge node finds the targeted vehicle, the edge node will report the vehicle's location using the `TaskReport` interface. As we mentioned in Section II, there are hundreds of cameras in the city, so we need a dynamic topology to optimize the task-transfer scheme. We define and implement the `SetTopology` and `GetTopology` interfaces to control the application-defined topology for *Task Receiver*s. We provide the `MobileCameraLogin` and `MobileCameraLogout` interfaces for mobile camera login and logout.

As Listing 3 shows, a *Task Receiver* provides several services. The *Control Center* will call the `FindObject`

Listing 3. Example of a *Task Receiver*'s interface.

```
/* TaskReceiver: the interface should be implemented on
     the edge node connected with control center */
public interface TaskReceiver_I{
  /* Receive object information for finding */
  public void FindObject(int task_id ,String object_info);
  /* Stop to search the object */
  public void StopSearching(int task_id);
  /* Receive object information for finding transferred by
       the other edge node */
  public void TaskTransfer(String object_info);
  /* Motion detection for a video */
  public Byte[] MotionDetection(Byte[] videoframe);
  /* Recongnize the license plate number from a license
       plate image */
  public Byte[] PlateRecognition(Byte[] image);
  /* Detect the license plate from motion of a frame */
  public Byte[] PlateDetection(Byte[] data);
}
```

interface to publish an AMBER alert. When a *Task Receiver* receives a task from the *Control Center*, it will set up a timer for task scheduling according to the value of timeout in the service deployment. And once conditions are met, a task will be transferred to other *Task Receiver*s by calling the `TaskTransfer` interface. If a *Task Receiver* receives a stop signal from the `StopSearching` interface, it will stop searching for this alert task, and also, it will forward this signal to the edge nodes that sent/received a transferred task to/from the former. Once the targeted vehicle is found, the *Task Receiver* will report the vehicle's location using the *Control Center*'s `TaskReport` interface and it will send a clear sign to neighboring *Task Receiver*s. Then, it will reset all task-scheduling timers for task scheduling. The `MotionDetection`, `PlateRecognition` and `PlateDetection` interfaces are used to implement video analytics. We implemented them based on the OpenALPR [27] library. As mentioned in Section III-B, those services use message queues as the input and output streams. This means that the `MotionDetection` interface will get the video frame from its input message queue, the data of which comes from a camera's live video. After processing, the `MotionDetection` interface will save the motion area image to its output message queue, which is also the input of the `PlateDetection` interface. As we mentioned, a subservice launches several worker instances for parallel execution. The `PlateDetection` interface supports this feature by declaring the deployment plan in a JSON string. The reason the `PlateRecognition` interface is not auto-scaling is because plate recognition is not as computationally intensive as plate detection, and some frames may not have a license plate.

The edge node of *Task Receiver* may not have enough computing power to process frames in real time. Therefore, we set up several *Data Processor*s in the local edge environment. Listing 4 shows the interface of a *Data Processor*. It just provides `PlateRecognition` and `PlateDetection` interfaces.

D. Task Scheduling

Here, we introduce strategies used in the *Job Schedule* module to control the vehicle-tracking area in our implementation.

Listing 4. Example of a *Data Processor*'s interface.

```
/* DataProcessing: the interface should be implemented on
     the data processing edge node */
public interface DataProcessing_I{
  /* Recongnize the license plate number from a license
       plate image */
  Byte[] PlateRecognition(Byte[] data);
  /* Detect the license plate from the motion of a frame */
  Byte[] PlateDetection(Byte[] data);
}
```

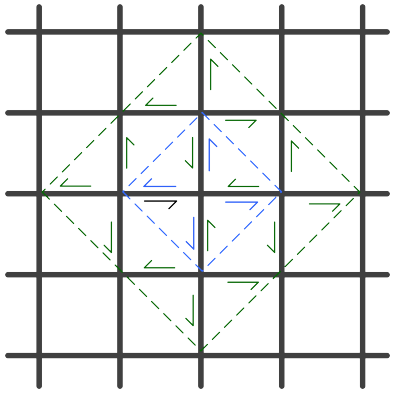


Fig. 8. A sample topology of road cameras.

Obviously, cameras have many limitations (for example, focal length, focus, and angle), and in the actual scenario it usually captures only one-way traffic flow. Thus, a two-way road needs two cameras located on each side of the road. In this case, we assume a simple and regular road topology where all roads are two-way and monitored by cameras (see Fig. 8). For a vehicle-tracking task, we present two diffusion models for task scheduling: distance-related diffusion (DD) and location-direction-related diffusion (LD). With both, we assume that a task will be sent to the edge node nearest to the kidnapping location with a fleeing direction. Thus, the number of initialization camera is $n_0 = 1$. For convenience, we choose the first crossing of the fleeing direction as the initial point for diffusing. Once the targeted vehicle is found, it will be reset so that only one edge node receives this task. For simplicity, we consider all cameras to have the same value of timeout for diffusing a task. The black arrow indicates the kidnapping location and driving direction.

1) Distance-Related Diffusion: A DD strategy, as the simplest strategy, diffuses the task according to the distance from where the kidnapping occurred. For example, the tracking area's radius will increase by a fixed number as time passes. For the topology in Fig. 8, the *Task Receiver* will send the task to seven other neighboring cameras based on its direction when diffusing a task. As Fig. 8 shows, the camera at the black arrow will transfer the task to the cameras in the area enclosed by the blue dashed square. Then, in next cycle, all cameras in green square will execute the task. According to the rule of diffusion, we see that the increased cameras are all located between two squares as the blue and green squares. Hence, the increment of two adjacent times can be expressed by the equation $\Delta_t^D = sr(2t-1), t \geq 2$, where s defined as the side numbers of the square is 4, and r defined as the

cameras of each side is 2.

Thus, the number of working *Task Receiver*s for time t is expressed by the following equation:

$$n_t^D = \begin{cases} n_0, & t = 0 \\ srt^2, & t > 0 \end{cases} \quad (1)$$

2) Location-Direction-Related Diffusion: In the actual scenario, the DD strategy has many disadvantages. For instance, the speed on different roads varys (the speed on the highway is twice the speed allowed on city streets). So, if we set the tracking area's radius according to the highway speed, we waste the computing of local street cameras. On the other hand, if set the radius according to the local, street cameras, our tracking could fail once the kidnapper flees by highway. We therefore propose an LD strategy for A3. In this model, the edge node will transfer the task to edge nodes according to the road topology. For example, the edge node located at the black arrow will transfer the task to the blue ones, because a vehicle in a crossroad has only four choices: go straight, or turn left, right, or around. As with the DD, the initial number of working edge nodes is $n_0^L = n_0$, and in the next cycle, it will include the black arrow and blue arrows shown in Fig. 8. In the second cycle, it will include the black arrow, blue arrows and green arrows. Hence, the increment of two adjacent times can be expressed by the equation $\Delta_t^L = s\frac{r}{2}(2t-1) + s\frac{r}{2}(2(t-1)-1), t \geq 3$.

Thus, the number of working *Task Receiver*s each time t is expressed by the following equation:

$$n_t^L = \begin{cases} n_0, & t = 0 \\ n_0 + s\frac{r}{2}, & t = 1 \\ srt^2 - srt + s\frac{r}{2}, & t \geq 2 \end{cases} \quad (2)$$

Fig. 9 shows the number of working nodes for the two aforementioned strategies, where the time cycle is based on Eqs. (1) and (2). The results in Fig. 9 show that the LD launches fewer edge nodes to track the targeted vehicle, which will save significant computing resources and energy. Actually, the cameras in the city will not be shown regularly in Fig. 8, and neither will the road topology. Both strategies are easy to implement in *Firework*, to modify the application-defined topology. Besides, it is easy to set different timeout values for different edge nodes for optimization. For example, setting the timeout value to 30 seconds for highway cameras and 60 seconds for local cameras is more reasonable and efficient. Thus, the LD strategy will is more efficient, because it significantly reduces the number of participant edge devices.

V. Evaluation

We implemented three types of *Firework.Node* on A3 using three open source software: FFmpeg [38] for video decoding, OpenCV [28] for image processing and OpenALPR [27] for recognizing license plates. To demonstrate our extended version of *Firework* and to evaluate A3, we first evaluate the local edge nodes' collaborative performance. Then, we evaluate the two task-scheduling strategies in A3. All of these experiments compare performance in terms of latency and the number of workers over time.

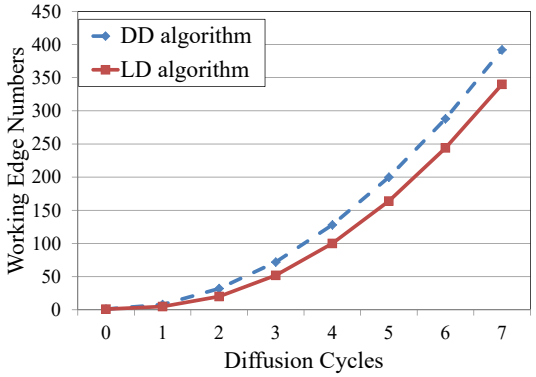


Fig. 9. The number of camera participants, in theory.

A. Experimental Setup

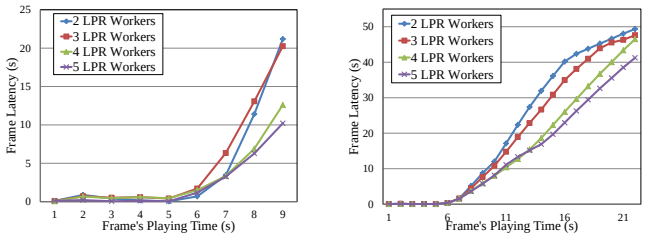
We have built a testbed for A3 consisting of 81 virtual machines on the Amazon EC2. All the machines have the same vCPU (i.e., Intel Xeon CPU at 2.4 GHz) and they can communicate with each other. But we define the application topology, to control communications. Because of Amazon EC2's limitations of service, we deploy these virtual machines in four data centers under two Amazon accounts. In this testbed, we don't consider pulling the video from cameras. We just store the video data in the virtual machines and control the playing speed to simulate a live video stream.

We also built a local testbed comprised of several desktops, to evaluate the local edge nodes' collaborative performance. The reason we built two testbeds is that the cloud one is mainly for demonstrating the application and task-scheduling functions, and the local one is closer to reality.

The experimental video data are collected on a large-scale campus with 25,000 students. The resolution of all the video is 1280 $\times$ 720 pixels and 25 frames per second. The video data are encoded in H.264 format, with a baseline profile where one intra-frame is followed by 49 predictive-frames, which is common for a live video stream.

B. Collaboration of Local Edge Nodes

In this section, first we set up multiple workers on one edge node, to try to achieve real-time video analytics using one edge node. The reason we do this is that video analytics benefits from parallel processing, and this experiment reveals how many worker instances can process video in real time. We use four types of Amazon EC2 virtual machines: t2.small, t2.medium, t2.xlarge, and t2.2xlarge. All of them have the same CPU core (an Intel Xeon at 2.40 GHz), but each one has a different number of cores (one, two, four, and eight cores, respectively). To demonstrate local edge nodes' collaboration, we set up two edge nodes in the cloud and three edge nodes locally, to collaboratively analyze the video and evaluate the performance of these two cases in terms of frame latency (defined as the time duration between when a video frame is generated and recognized). The edge nodes on the cloud are Amazon EC2 t2.xlarge, and the local edge nodes are on a Dell OptiPlex desktop with an Intel i5-4590 at 3.3 GHz.



(a) AWS EC2 t2.small. (b) AWS EC2 t2.medium.

(c) AWS EC2 t2.xlarge. (d) AWS EC2 t2.2xlarge.

Fig. 10. Frame latency over time in different types of AWS nodes with different numbers of LPR workers.

Fig. 10 shows the average frame latency of each second, regarding the different number of LPR worker instances implemented by a *Firework.Node*. Note that we have one video decoding worker instance and two motion detection (MD) worker instances in every *Task Receiver* for video decoding and motion detection. In general, when more LPR instances are running on the edge nodes, we see lower frame latency. According to the experiment in Section II-A, the average processing time for plate recognition is less than 160 ms for the Amazon EC2 node. As Fig. 10(d) shows, though, the Amazon EC2 t2.2xlarge is able to process the video in real time with five LPR instances. However, this is only one that achieves a real-time video analytics; all the others do not. The reason other Amazon EC2 nodes are difficult to process in real time because of the limited number of CPU cores. It is worth nothing, though, that these node still achieve much lower latency–for several subsequent frames–than previous efforts (see, for example, the frames after 31 seconds). According to the CDF analysis shown in Fig. 2 and the processing log, the explanation for this is that there arr no motion areas in some video fragments. Because it does not generate any plate-recognition workloads for plate-recognition subservice, it reduces the LPR worker's workload.

As we mentioned, we also measure the performance on local edge nodes. The results are as shown in Fig. 11, where for all cases it is difficult to process the video in real time, and the case including five license plate instances is worse than the case which only has four instances. This is because of the limitation of the number of CPU threads per core. The local edge node we used is a Dell OptiPlex desktop with Intel i5-4590 at 3.3GHz, which is a 4-cores and 4-threads CPU. When the number of working threads is more than the number of the CPU's threads, it will cost a lot to switch threads. Note that the CPU core's frequency shortens the processing time for each frame, and the number of the CPU's threads per core is related to how many threads run at the same time.

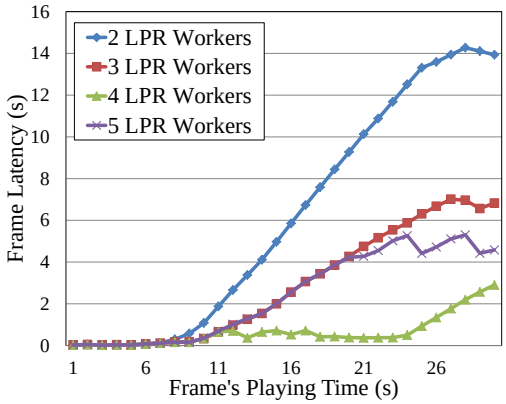


Fig. 11. Frame latency over time on a Dell OptiPlex with different numbers of LPR workers.

TABLE II
THE CASE DESCRIPTION FOR THE CLOUD ENVIRONMENT.

	Firework.Node	MD instances' No.	LPR instances' No.
Case 1	*Task Receiver*	2	2
	Data Processor	-	1
Case 2	*Task Receiver*	2	2
	Data Processor	-	3
Case 3	*Task Receiver*	2	3
Case 4	*Task Receiver*	2	5

The aforementioned experiments show that when an edge device wants to process video in real time, the CPU's threads maximum must large enough. Generally, it should be more than 4 threads, but this cannot always be satisfied. Thus, *Firework* allows the *Task Dispatch* module to dispatch an overloaded task to other *Firework.Node*s, which also provides the same subservice. In A3, *Data Processor*s play this role.

According to the results of Amazon EC2 t2.xlarge in Fig. 10(c), it still cannot process the video data in real time by itself. To demonstrate the *Firework*'s collaboration on the local edge, we set up two Amazon EC2 t2.xlarge nodes for collaboration, one of which is a *Task Receiver*, and another is a *Data Processor*. Table II shows the cases we used in the cloud. For case 1 and 2, we set different numbers of LPR instances on the *Data Processor*s. And for comparison, all LPR instances running on one edge node are essentially case 3 comparing with case 1, in which one RP instance runs on the *Task Receiver* and other two instances run on the *Data Processor*. In a similar vein, we also set up a comparison for case 2.

Fig. 12 illustrates the average frame latency of each second regarding all the different cases. From the results, we see that collaborative solutions are better. As we mentioned, case 4 cannot process video in real time even though it used five LPR instances (it is limited by the core number of Amazon t2.xlarge node's CPU). However, case 2 processes the video in real time, which has the same number of RP instances. Thus, a collaborative solution avoids the limitation of the number of cores in a multithread data processing application.

We also evaluate the collaborative performance using several local edge devices, which is closer to reality. Table III describes the configuration of each case we used. Because the

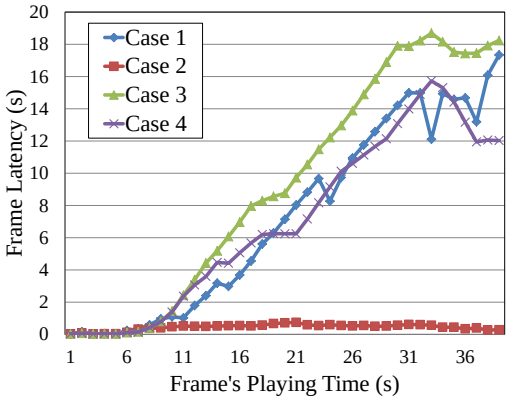


Fig. 12. Frame latency over time as AWS nodes collaborate.

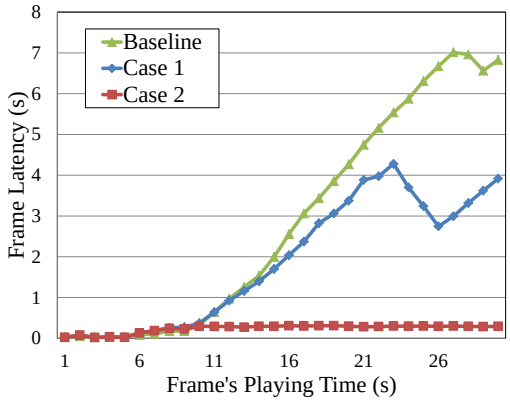


Fig. 13. Frame latency over time under the collaboration of local edge nodes.

TABLE III
THE CASE DESCRIPTION FOR THE LOCAL ENVIRONMENT.

	Firework.Node	MD instances' No.	LPR instances' No.
Baseline	*Task Receiver*	2	3
Case 1	*Task Receiver*	2	1
	Data Processor	-	2
Case 2	*Task Receiver*	2	1
	Data Processor 1	-	2
	Data Processor 2	-	2

computation resource of local edge devices is less than the Amazon EC2 t2.xlarge, we set a three-node case (see case 2 in Table III) to try getting real-time video processing.

Fig. 13 shows the results. Case 1 is better than the baseline, although they have the same number of LPR instances (but case 1 benefits from the edge nodes' collaboration). When we increase the number of *Data Processor*s to two, it can process video in real time.

C. Task Scheduling

Here, we evaluate the performance of A3's task-scheduling part on our testbed. Fig. 8 shows the road topology we used in this experiment. We deployed 80 *Task Receiver*s to simulate 80 edge nodes connected with road cameras. All of these *Task Receiver*s are hosted on the Amazon EC2 t2.2xlarge. We deployed one *Control Center* on a low-performance Amazon EC2 virtual machine (that is, the t2.small with one core CPU and 512 MB memory). We did not deploy any *Data Processor*s on the testbed, because we apply 8 core CPU for our virtual machine, which can analyze the video in real time using five PR worker instances. Note that the reasons we apply such powerful CPUs are multifold. First, we achieve real-time video analytics when the local edge nodes collaborate. Second, in this section, we mainly want to demonstrate the task-scheduling function. Last, one Amazon account can apply only a limited number of Amazon EC2 nodes. In this experiment, we concentrate on simulating task scheduling and evaluating the performance. Because of the Amazon EC2 service's limitations, we deployed those 81 nodes in four data centers, including Ohio, northern California, Oregon, and northern Virginia. We deployed the applicable video data in *Task Receiver*s, to make sure that four of them will locate the

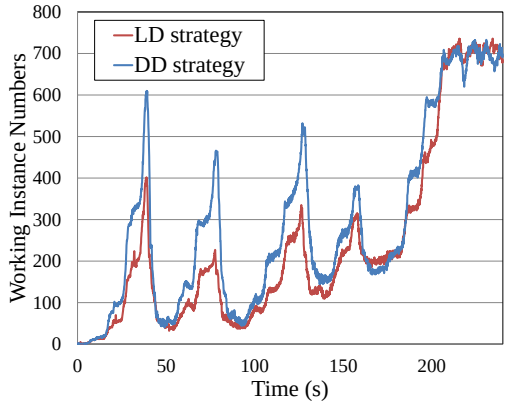


Fig. 14. Comparison results of two task-scheduling strategies.

vehicle four times, around the 37th, 75th, 125th, and 150th seconds. In this experiment, the number of working instances is recorded per 100 milliseconds, to quantize the workloads.

We demonstrate our two task-scheduling strategies by setting the corresponding application defined topology in the *Control Center*. Then, we run the experiments several times for an average result. Fig. 14 shows the results. Case 1 is the result of applying the DD strategy, and case 2 applies the LD strategy. The workload of all edge nodes in case 2 is less than case 1 before 200s, and once the targeted vehicle is located, the workload reduces immediately. Then, as the searching area extends again, the workload increases. Last, the workload of two cases will increase to the same value. This is because we only use 80 edge nodes to track the vehicle, and as time passes, all of them will participate in tracking the vehicle.

VI. Related Work

Inspired by low-latency analytics, edge computing [21] (also known as fog computing [13], mobile edge computing [14], and Cloudlet [15]) processes data at the proximity of data sources. Satyanarayanan *et al.* [15] proposed Cloudlet, for example, which uses servers located at the edge of the network, so that computationally intensive processing can be offloaded to these edge servers. Habak *et al.* [39] proposed a dynamic, self-configuring, and multidevice mobile cloud out of a cluster of mobile devices, which provides a cloud service at the

edge. Fernando *et al.* [40] also proposed a similar cloud of mobile devices. Saurez *et al.* [41] proposed a programming infrastructure for the geodistributed computational continuum represented by fog nodes and the cloud, called Foglets. *Firework* differs from each of these systems, because it leverages not only mobile devices and the cloud, but also edge nodes to complete tasks collaboratively; the other aforementioned systems are not for large-scale data processing and sharing among multiple stakeholders.

As a killer application, several edge video analytics platforms have been proposed. Ananthanarayanan *et al.* [42] present a geodistributed framework for large-scale video analytics, which meets the strict requirements of real time. It carries out different computation modules using computer vision by leveraging the public cloud, private clouds, and edge nodes. The difference between *Firework* and Ananthanarayanan's work is that our work expands data sharing, along with attached computing modules (such as functions in *Firework.View*) and programming interfaces are provided for developers to build their application on edges and the cloud. Wang *et al.* [43] proposed a real-time face recognition and tracking framework, called OpenFace, which also used edge computing to analyze live video. To protect privacy, OpenFace selectively blurs faces in video data, depending on user-specific privacy policies. However, OpenFace only considers edge nodes. In our work, *Firework* and A3 leverage both edge nodes and the cloud.

Zhang *et al.* [44] proposed a real-time video analytics platform, called VideoStorm, which leverages large clusters. It pushes all the video to clusters, and it has prohibitive costs. Our system use edge nodes to reduce costs in terms of latency and network bandwidth. Yi *et al.* [45] proposed a latency-aware video analytics platform, called LAVEA. In LAVEA, the video data will be pushed to an edge-front node and each video frame will be decoded and analyzed at other local edge nodes. This is similar to the collaboration of local edges. They also proposed several scheduling strategies to reduce latency. As with OpenFace, it only leverages edge nodes. Long *et al.* [46] proposed an edge computing framework for cooperative video processing in the IoT domain. They use mobile devices as edges to enhance the computing power and network quality by multiple uploading paths. Grassi *et al.* [47] proposed an application called ParkMaster, which detects parking spaces and reports vacant ones to the cloud for sharing this information with other people. It uses smartphones as edge devices in the vehicle for detection. Because detection algorithms usually cost much less than the recognition algorithms, smartphones can process the video in real time.

VII. Conclusion

In this paper, we investigated the barriers of designing and implementing a distributed collaborative execution on the edge, such as a real-time vehicle-tracking application that improves the *AMBER Alert* system significantly. To attack these barriers, we extended a big data processing and sharing framework in an edge-cloud environment, to support collaboration of local edge nodes and a customizable task-scheduling scheme. Inspired by a real project (Project Green Light in Detroit), we abstracted out the network model, applying in A3. Based on *Firework*'s extensions, we implemented the *AMBER Alert Assistant* (A3), which supports different tracking strategies. Then, we evaluated this application's performances. The results show that A3 can analyze video streams in real time by collaborating with several edge nodes, and the proposed location-direction-related task-scheduling strategy (LD) is more efficient at controlling the search area for vehicle tracking. This demonstrates that A3 is ready to deploy on top of Project Green Light, and we believe A3 will improve the *AMBER Alert*.

Currently, *Firework* does not provide an interface to adjust the deployment plan, so A3 cannot dynamically adjust the diffusion rate. This is helpful in a real scenario, considering road conditions. For future work, we will design and implement such module. Besides, an elastic message queue module is also our future work, which is able to dynamically launch the best suitable message queue system for each edge device and a lightweight inner message queue for modules in *Firework*.

Acknowledgment

This work is supported in part by National Science Foundation (NSF) grant CNS-1741635, and Qingyang Zhang and Hong Zhong are in part supported by National Science Foundation of China (No. 61572001) and Key Technology R&D Program of Anhui Province (1704d0802193). The authors are very grateful to the anonymous referees for their detailed comments and suggestions regarding this paper.

References

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica *et al.*, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[2] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *ACM SIGOPS operating systems review*, vol. 37, no. 5. ACM, 2003, pp. 29–43.

[3] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proceedings of IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2010, pp. 1–10.

[4] (2017, Feb.) Apache storm. [Online]. Available: https://storm.apache.org/

[5] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, vol. 10, 2010, p. 10.

[6] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 423–438.

[7] Q. Zhang, Y. Song, R. Routray, and W. Shi, "Adaptive block and batch sizing for batched stream processing system," in *Proceedings of IEEE International Conference on Autonomic Computing (ICAC)*, July 2016, pp. 35–44.

[8] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[9] D. E. Culler, "The once and future internet of everything," *GetMobile: Mobile Computing and Communications*, vol. 20, no. 3, pp. 5–11, 2017.

[10] (2017, Feb.) First workshop on video analytics in public safety. [Online]. Available: https://www.nist.gov/sites/default/files/documents/2017/01/19/ir_8164.pdf

[11] (2016, Apr.) Cisco global cloud index: Forecast and methodology 20142019 white paper. [Online]. Available: http://www.cisco.com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/Cloud

[12] D. Evans, "The internet of things: How the next evolution of the internet is changing everything," *CISCO white paper*, vol. 1, no. 2011, pp. 1–11, 2011.

[13] F. Bonomi, R. Milito, J. Zhu, and 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.

[14] M. Patel, B. Naughton, C. Chan, N. Sprecher, S. Abeta, A. Neal *et al.*, "Mobile-edge computing introductory technical white paper," *White Paper, Mobile-edge Computing (MEC) industry initiative*, 2014.

[15] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *Pervasive Computing*, vol. 8, no. 4, pp. 14–23, 2009.

[16] M. Satyanarayanan, P. Simoens, Y. Xiao, P. Pillai, Z. Chen, K. Ha, W. Hu, and B. Amos, "Edge analytics in the internet of things," *IEEE Pervasive Computing*, vol. 14, no. 2, pp. 24–31, 2015.

[17] W. Hu, B. Amos, Z. Chen, K. Ha, W. Richter, P. Pillai, B. Gilbert, J. Harkes, and M. Satyanarayanan, "The case for offload shaping," in *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications*. ACM, 2015, pp. 51–56.

[18] M. Satyanarayanan, "A brief history of cloud offload: A personal journey from odyssey through cyber foraging to cloudlets," *GetMobile: Mobile Computing and Communications*, vol. 18, no. 4, pp. 19–23, 2015.

[19] M. Satyanarayanan, R. Schuster, M. Ebling, G. Fettweis, H. Flinck, K. Joshi, and K. Sabnani, "An open ecosystem for mobile-cloud convergence," *IEEE Communications Magazine*, vol. 53, no. 3, pp. 63–70, 2015.

[20] S. Agarwal, M. Philipose, and P. Bahl, "Vision: the case for cellular small cells for cloudlets," in *Proceedings of the fifth international workshop on Mobile cloud computing & services*. ACM, 2014, pp. 1–5.

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

[22] K. Pulli, A. Baksheev, K. Kornyakov, and V. Eruhimov, "Real-time computer vision with opencv," *Communications of the ACM*, vol. 55, no. 6, pp. 61–69, 2012.

[23] Q. Zhang, X. Zhang, Q. Zhang, W. Shi, and H. Zhong, "Firework: Big data sharing and processing in collaborative edge environment," in *2016 Fourth IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*, Oct 2016, pp. 20–25.

[24] Q. Zhang, Q. Zhang, W. Shi, and H. Zhong, "Firework: Data processing and sharing for hybrid cloud-edge analytics," *Technical Report MIST-TR-2017-002*, 2017.

[25] (2017, Mar.) Amber alert. [Online]. Available: https://en.wikipedia.org/wiki/AMBER_Alert

[26] (2017, Mar.) Project green light. [Online]. Available: http://www.greenlightdetroit.org/

[27] (2017, Mar.) Openalpr. [Online]. Available: https://github.com/openalpr/openalpr

[28] (2017, Mar.) Opencv. [Online]. Available: http://www.opencv.org/

[29] (2017, Mar.) Docker. [Online]. Available: https://www.docker.com/

[30] (2017, Mar.) Openstack. [Online]. Available: http://www.openstack.org/

[31] (2017, Mar.) Vmware. [Online]. Available: https://www.vmware.com/

[32] (2016, Sep.) etcd. [Online]. Available: https://github.com/coreos/etcd

[33] (2017, Mar.) Introducing json. [Online]. Available: http://www.json.org/

[34] L. Guo, C. Zhang, J. Sun, and Y. Fang, "Paas: A privacy-preserving attribute-based authentication system for ehealth networks," in *2012 IEEE 32nd International Conference on Distributed Computing Systems*, June 2012, pp. 224–233.

[35] J. Shao, R. Lu, and X. Lin, "Fine-grained data sharing in cloud computing for mobile devices," in *2015 IEEE Conference on Computer Communications (INFOCOM)*, April 2015, pp. 2677–2685.

[36] U. Hunkeler, H. L. Truong, and A. Stanford-Clark, "Mqtt-s: A publish/subscribe protocol for wireless sensor networks," in *Communication Systems Software and Middleware and Workshops, 2008. COMSWARE 2008. 3rd International Conference on*, Jan 2008, pp. 791–798.

[37] J. Kreps, N. Narkhede, J. Rao *et al.*, "Kafka: A distributed messaging system for log processing," in *Proceedings of the NetDB*, 2011, pp. 1–7.

[38] (2017, Mar.) Ffmpeg. [Online]. Available: https://ffmpeg.org/

[39] K. Habak, M. Ammar, K. A. Harras, and E. Zegura, "Femto clouds: Leveraging mobile devices to provide cloud service at the edge," in *Proceedings of the IEEE 8th International Conference on Cloud Computing*. IEEE, 2015, pp. 9–16.

[40] N. Fernando, S. W. Loke, and W. Rahayu, "Computing with nearby mobile devices: a work sharing algorithm for mobile edge-clouds," *IEEE Transaction on Cloud Computing*, 2016.

[41] E. Saurez, K. Hong, D. Lillethun, U. Ramachandran, and B. Ottenwälder, "Incremental deployment and migration of geo-distributed situation awareness applications in the fog," in *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*. ACM, 2016, pp. 258–269.

[42] G. Ananthanarayanan, P. Bahl, P. Bodk, K. Chintalapudi, M. Philipose, L. Ravindranath, and S. Sinha, "Real-time video analytics: The killer app for edge computing," *Computer*, vol. 50, no. 10, pp. 58–67, 2017.

[43] J. Wang, B. Amos, A. Das, P. Pillai, N. Sadeh, and M. Satyanarayanan, "A scalable and privacy-aware iot service for live video analytics," in *Proceedings of the 8th ACM on Multimedia Systems Conference*, ser. MMSys'17. New York, NY, USA: ACM, 2017, pp. 38–49.

[44] H. Zhang, G. Ananthanarayanan, P. Bodik, M. Philipose, P. Bahl, and M. J. Freedman, "Live video analytics at scale with approximation and delay-tolerance." in *NSDI*, 2017, pp. 377–392.

[45] S. Yi, Z. Hao, Q. Zhang, Q. Zhang, W. Shi, and Q. Li, "Lavea: Latency-aware video analytics on edge computing platform," in *Proceedings of 2nd ACM/IEEE Symposium on Edge Computing (SEC)*, ser. SEC'17. New York, NY, USA: ACM, 2017.

[46] C. Long, Y. Cao, T. Jiang, and Q. Zhang, "Edge computing framework for cooperative video processing in multimedia iot systems," *IEEE Transactions on Multimedia*, vol. PP, no. 99, pp. 1–1, 2017.

[47] G. Grassi, P. Bahl, J. Kyle, and G. Pau, "Parkmaster: An invehicle, edgebased video analytics service for detecting open parking spaces in urban environments," in *Proceedings of 2nd ACM/IEEE Symposium on Edge Computing (SEC)*, ser. SEC'17. New York, NY, USA: ACM, 2017.

Qingyang Zhang received the B. Eng. degree in computer science and technology from Anhui University, China in 2014. He is currently a Ph.D. candidate at Anhui University. He is also a visiting student in Wayne State University. His research interest includes edge computing, and security protocol for wireless network.

Quan Zhang received his Ph.D. degree in the Department of Computer Science at Wayne State University in 2018, and his M.S. degree in Computer Science at Wayne State University in 2016. Now he is a Data Engineer at Salesforce. His research interests include Cloud Computing, Edge Computing, Real-time Streaming Processing, and Energy-efficient Systems.

Weisong Shi is a Charles H. Gershenson Distinguished Faculty Fellow and a professor of Computer Science at Wayne State University. His research interests include Edge Computing, Computer Systems, energy-efficiency, and wireless health. He received his BS from Xidian University in 1995, and Ph.D. from Chinese Academy of Sciences in 2000, both in Computer Engineering. He is a recipient of National Outstanding PhD dissertation award of China and the NSF CAREER award. He is an IEEE Fellow and ACM Distinguished Scientist.

Hong Zhong received her B.S. degree in applied mathematics in Anhui University, China, in 1986, and the Ph.D. degree in computer science and technology from University of Science and Technology of China (USTC), China, in 2005. Now she is a professor and Phd Advisor of Anhui University. Her research interests cover network and information security.