# Firework: Data Processing and Sharing for Hybrid Cloud-Edge Analytics

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

**Abstract**—Now we are entering the era of the Internet of Everything (IoE) and billions of sensors and actuators are connected to the network. As one of the most sophisticated IoE applications, real-time video analytics is promising to significantly improve public safety, business intelligence, and healthcare & life science, among others. However, cloud-centric video analytics requires that all video data must be preloaded to a centralized cluster or the cloud, which suffers from high response latency and high cost of data transmission, given the scale of zettabytes of video data generated by IoE devices. Moreover, video data is rarely shared among multiple stakeholders due to various concerns, which restricts the practical deployment of video analytics that takes advantages of many data sources to make smart decisions. Furthermore, there is no efficient programming interface for developers and users to easily program and deploy IoE applications across geographically distributed computation resources. In this paper, we present a new computing framework, *Firework*, which facilitates distributed data processing and sharing for IoE applications via a virtual shared data view and service composition. We designed an easy-to-use programming interface for *Firework* to allow developers to program on *Firework*. This paper describes the system design, implementation, and programming interface of *Firework*. The experimental results of a video analytics application demonstrate that *Firework* reduces up to 19.52 percent of response latency and at least 72.77 percent of network bandwidth cost, compared to a cloud-centric solution.

**Index Terms**—Distributed big data processing, edge computing, internet of everything

## 1 Introduction

In the big data era, researchers and practitioners have treated cloud computing as the de facto large-scale data processing platform within a cluster or in the cloud. Numerous cloud-centric data processing platforms [1], [2], [3], [4], [5] that leverage a MapReduce [6] programming framework, have been proposed for both batch and streaming data in the last decade. At the same time, we are entering the era of the Internet of Everything (IoE), where billions of geographically distributed sensors and actuators are connected and immersed in our daily life. By 2020, 50 billion things will join the Internet and generate 507.5 Zettabytes (ZB) of data per year due to increasing machine-to-machine communications, according to the projection by Cisco [7], [8]. However, the estimated data center traffic in the year of 2019 is 10.4 ZB (i.e., 2 percent of the data generated by IoE devices), which implies that most IoE data has to be stored and processed close to data sources. Therefore, it is inefficient to process IoE data in a centralized environment due to the response latency and network bandwidth cost. Instead, the emerging *Edge Computing* (a.k.a., fog computing [9], cloudlet [10]) referring 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," [11] enables data processing at the proximity of data sources. Edge computing decentralizes the data processing to the edge of the network, and message brokers [12], [13] and streaming processing platforms are employed to build real-time analytics systems [14], [15].

However, an important and fundamental assumption behind cloud computing and edge computing is that the data is owned by a single stakeholder, in which the data owner has full control privileges of the data. As we mentioned, cloud computing requires the data to be preloaded in data centers before a user runs its applications in the cloud [16], while edge computing processes data at the edge of the network but requires close control of the data producers and consumers. Data owned by multiple stakeholders is rarely shared among data owners. Taking the cooperation in connected health as an example, the health records of patients hosted by hospitals and customer records owned by insurance companies are highly private to the patients and customers and rarely shared. If an insurance company has access to its customers' health records, the insurance company could initiate personalized health insurance policies for its customers based on their health records. Another example is "find the lost" in a city [17], where video streams from multiple data owners across the city are used to find a lost object. It is common that the police department manually collects video data from

- *Q. Zhang and W. Shi are with the Department of Computer Science, Wayne State University, Detroit, MI 48202. E-mail: {quan.zhang, weisong}@wayne.edu.*
- *Q. Zhang is with the Department of Computer Science, Wayne State University, Detroit, MI 48202, and also with the School of Computer Science and Technology, Anhui University, Hefei 230039, China. E-mail: qyzhang@wayne.edu.*
- *H. Zhong is with the School of Computer Science and Technology, Anhui University, Hefei 230039, China. E-mail: zhongh@ahu.edu.cn.*

*Manuscript received 17 Jan. 2017; revised 17 Feb. 2018; accepted 23 Feb. 2018. Date of publication 5 Mar. 2018; date of current version 8 Aug. 2018. (Corresponding author: Quan Zhang.) Recommended for acceptance Z. Lan. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TPDS.2018.2812177*

surveillance cameras on the streets, retailer shops, individual smart phones, or car video recorders in order to identify a specific lost object. If all these data could be shared seamlessly, it could save huge amount of human work and identify an object in real-time fashion. Furthermore, simply replicating data or running an analyzing application provided by a third party on stakeholders' data may break the privacy and security restrictions. Unfortunately, none of the aforementioned can be easily achieved by leveraging cloud computing or edge computing individually.

In this paper, we envision that in the era of IoE, the demand of distributed data sharing and processing applications will dramatically increase because the data produced and consumed are pushed to the edge of the network. We present a new computing framework called *Firework*that facilitates distributed data processing and sharing for IoE analytics while keeping the data and computation within stakeholders' facilities. We have explained our motivating application and highlighted its challenges and our contributions in the following sections.

### 1.1 Motivating Application

Real-time video analytics is promising to significantly improve public safety, business intelligence, and healthcare & life science, among many others. Video analytics from cameras installed on either fixed positions (e.g., traffic light, retail store) or mobile carriers (e.g., vehicles, smart phones) are used for traffic safety and planning, self-driving and smart cars, surveillance and security, as well as user-centric applications including digital assistant and augmented reality [18]. We will use object tracking in the AMBER alert system as the motivating application to illustrate how edge computing could benefit the application and the remaining challenges of implementing the application in a hybrid cloud-edge environment.

Conventionally, cloud-centric object tracking requires video streams to be preloaded to a centralized cluster or cloud, which suffers from extremely high response latency and high cost of data transmission. The emerging edge computing can reduce the response latency and network bandwidth cost by analyzing video streams close to data sources. Fig. 1 shows an example of a video analytics application of object tracking in an AMBER alert system in a hybrid cloud-edge environment. As shown in Fig. 1, video streams are collected by various cameras and owned by different owners. The *video clipping* deployed at the edge, scans video streams to selectively filter out frames with a license plate and send these frames to the cloud. When compare this to a cloud-centric scenario, massive data transmission between the data sources and the cloud can be avoided. In the cloud, *license plate recognition & tracking* is used to recognize the text on the plate and tracking a target plate number. Note that the *license plate recognition & tracking* can also be collocated with *video clipping* at the edge, in which case the response latency and network cost can be further reduced by eliminating the communication between the edge and the cloud.

### 1.2 Challenges

Although edge computing is promising for real-time video analytics, several barriers still prevent it from practical deployment. First, as shown in Fig. 1, the data sources vary

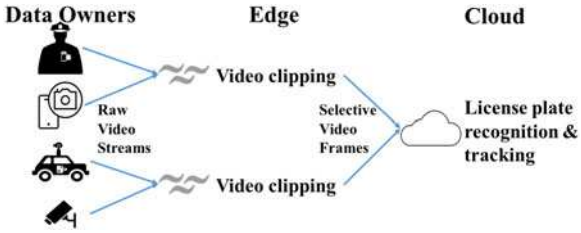



Fig. 1. An example of real-time video analytics (i.e., license plate recognition & tracking in an AMBER alert system) in a hybrid cloud-edge environment.

from on-body cameras to fixed security cameras so that it is hard to share the data in a real-time manner with others due to privacy issues (e.g., faces captured by traffic/security cameras) and resource limitation (e.g., power/computation/network limitations of on-body or dash cameras). Second, services provided by the edge and the cloud (e.g., *video clipping* and *license plate recognition* in Fig. 1) might be implemented in various platforms over geo-distributed computing nodes, which brings heavy overhead for developers to orchestrate data/services among different data sources and computing platforms. Third, programming in such a cloud-edge environment is difficult considering the variants of data ownership, data format, APIs provided by data owners, and computing platforms. Therefore, in this paper we will focus on several major challenges of facilitating data sharing and processing in a hybrid cloud-edge environment and summarize as following:

(1) Applications in a collaborative cloud-edge environment require full control privileges over the data from multiple data sources/owners, which prevents data sharing among stakeholders due to the privacy issue and resource limitation;
(2) Although offloading from the cloud to the edge reduces response latency and network bandwidth cost, offloading of different applications depends on various polices and generates distinct intermediate data/service, which make the data/service orchestration much harder among multiple stakeholders and applications; and
(3) Programming in edge computing is less efficient than that in the cloud where numerous programming frameworks are widely adopted. The heterogeneity of edge devices increases the programming effort of developers and decreases the usability of existing applications.

By overcoming these barriers, we expect a uniform data processing and sharing framework that provides easy-to-use programming interfaces for collaborative cloud-edge applications among multiple stakeholders.

### 1.3 Our Contribution

To attack the aforementioned barriers, *Firework i*) fuses data from multiple stakeholders as a virtual shared data set which is a collection of data and predefined functions by data owners. The data privacy protection can be carried out using privacy preserving functions preventing data leakage by sharing sensitive knowledge only to intended users; *ii*) breaks down an application into subservices so that a user

can directly subscribe to intermediate data and compose new applications by leveraging existing subservices; and *iii*) provides an easy-to-use programming interface for both service providers and end users. By leveraging subservices deployed on both the cloud and the edge, *Firework* aims to reduce the response latency and network bandwidth cost for hybrid cloud-edge applications and enables data processing and sharing among multiple stakeholders. We implement a prototype of *Firework* and demonstrate the capabilities of reducing response latency and network bandwidth cost by using an edge video analytics application developed on top of *Firework*.

The rest of this paper is organized as follows. We present the system design in Section 2 and the implementation of a prototype in Section 3. Section 4 shows the case study and results. Section 5 discusses the limitations of our work. We review related work in Section 6. Finally, we conclude in Section 7.

## 2 System Design

*Firework* is a framework for big data processing and sharing among multiple stakeholders in a hybrid cloud-edge environment. Considering the amount of data generated by edge devices, it is promising to process the data at the edge of the network to reduce response latency and network bandwidth cost. To simplify the development of collaborative cloud-edge applications, *Firework* provides a uniform programming interface to develop IoE applications. To deploy an application, *Firework* creates service stubs on available computing nodes based on a predefined deployment plan. To leverage existing services, a user implements a driver program and *Firework* automatically invokes the corresponding subservices via integrated service discovery. In this section, we will introduce the detailed design of *Firework*, including terminologies, system architecture, and programmability, to illustrate how *Firework* facilitates the data processing and sharing in a collaborative cloud-edge environment.

### 2.1 Terminology

We first introduce the terminologies that describe abstraction concepts in *Firework*. Based on the existing definitions of the terminologies in our previous work [19], we extend and enrich their meanings and summarize them as following:

- *Distributed Shared Data (DSD)*: Data generated by edge devices and historical data stored in the cloud can be part of the shared data. DSD provides a virtual view of the entire shared data. It is worth noting that stakeholders might have different views of DSD.
- *Firework.View*: Inspired by the success of object oriented programming, a combination of *dataset* and *function*s is defined as a *Firework.View*. The *dataset* describes shared data and the *function*s define applicable operations on the dataset. A *Firework.View* can be implemented by multiple data owners who implement the same functions on the same type of dataset. To protect the privacy of data owners, the *function*s can be carried out by privacy preserving functions that share sensitive data only to intended users [20].

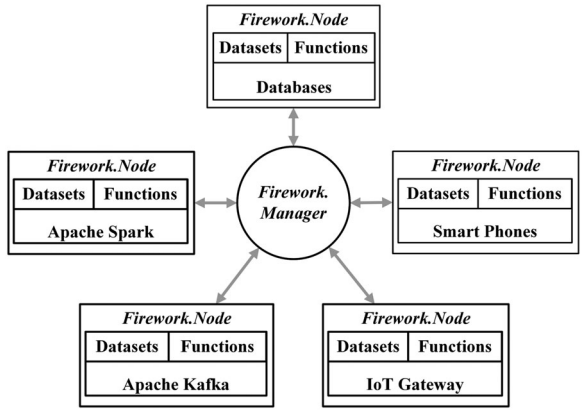



Fig. 2. An example of a *Firework* instance that consists of heterogeneous computing platforms.

- *Firework.Node*: A device that generates data or implements *Firework.View*s, is a *Firework.Node*. As data producers, such as sensors and mobile devices, *Firework.Node*s publish sensing data. As data consumers, *Firework.Node*s inherit and extend *Firework.View*s by adding functions to them, and the new *Firework.View*could be further extended by other *Firework.Node*s.

  An example application could be the city-wide temperature data, in which scenario sensor data is owned by multiple stakeholders and each of them provides public portals for data accessing. A user could reach all temperature data as if he/she operates on a single centralized data set. A sensor publishes a base *Firework.View* containing temperature data and read function, and a *Firework.Node* can provide a new *Firework.View* that returns the highest regional temperature by extending the base *Firework.View*.
- *Firework.Manager*: First, it provides centralized service management, where *Firework.View*s are registered. 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 and optimizes running services by dynamically scaling and balancing among *Firework.Node*s depending on their resource utilizations. Third, it allocates computation resources including CPU, memory, network, and (optional) battery resources to running services. Fourth, it exposes available services to users so that they can leverage existing services to compose new applications.
- *Firework*: It is an operational instance of a *Firework* paradigm. A *Firework* instance might include multiple *Firework.Node*s and *Firework.Manager*s, depending on the topology. Fig. 2 shows an example of a *Firework* instance consisting of five *Firework.Node*s employing heterogeneous computing platforms. If all *Firework.Node*s adopt a homogeneous computing platform, such a *Firework* instance will be similar to cloud computing and edge computing.

### 2.2 Architecture

As a major concept of *Firework*, *Firework.View* is abstracted as a "class-like" object, which can be easily extended. *Firework.Node* can be implemented by numerous heterogeneous computing infrastructures, ranging from big data

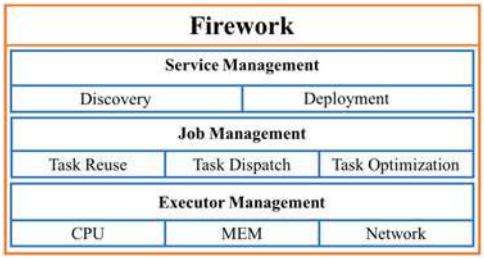



Fig. 3. An abstraction overview of *Firework*.

computation engines (e.g., Apache Spark [3], Hadoop [1], databases) and distributed message queues (e.g., Apache Kafka [12], MQTT [13], ZeroMQ [21], RabbitMQ [22]) in the cloud, to edge devices of smart phones and IoE gateways (e.g., Intel Edison, Raspberry Pi). *Firework.Manager* is the access point of a *Firework* instance that allows users to deploy and execute their services. Both *Firework.Node* and *Firework.Manager* can be deployed on the same computing node, where an edge node acts not only as a data consumer from the viewpoint of actuators, but also as a data producer from the cloud point of view.

To realize the aforementioned abstract concepts, we generalize them as a layered abstraction as shown in Fig. 3, which consists of *Service Management*, *Job Management*, and *Executor Management*. The *Service Management* layer performs service discovery and deployment, and the *Job Management* layer manages tasks running on a computing node. The combination of *Service Management* and *Job Management* fulfills the responsibilities of a *Firework.Manager*. The *Executor Management* layer, representing a *Firework.Node*, manages computing resources. In the following paragraphs, we will describe each layer in detail.

### 2.2.1 Service Management

To deploy a service on *Firework*, a user has to implement at least one *Firework.View* that 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. Note that the application defined deployment topology might be different from the underlying network topology. The reasons for providing a customizable deployment plan are to avoid redundant data processing and facilitate application defined data aggregation. In a cloud-centric application, data is uploaded to the cloud based on a predefined topology, where a developer cannot customize the data collection and aggregation topology. However, in IoE applications, sensors/actuators (e.g., smart phones, on-vehicle cameras) change the network topology frequently, which requires the application deployment to be adapted depending on available resources, network topology, and geographical location. Furthermore, *Firework.View* leverages multiple data sources to form a virtual shared data set, where the data sources can be dynamically added or removed according to the deployment plan of the IoE application.

Upon a service (i.e., *Firework.View*) registration, *Firework.Manager* creates a service stub for 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, input parameters, etc. A service provider can create a *Firework.View* via extending a registered service and register the new *Firework.View* to another *Firework.Manager*. By chaining up all these *Firework.View*s, a complex application can be composed. Depending on the deployment plan, a service (i.e., *Firework.View*) can be registered to multiple *Firework.Manager*s and the same service can be deployed on more than one computing node. An application developer can implement services and corresponding deployment plans via the programming interfaces provided by *Firework*. We will show more details through a concrete example (i.e., VideoAnalytics implemented on *Firework*) in Section 2.3.

To take advantage of existing services, a user retrieves the list of available services by querying *Firework.Manager*. Then the user implements a driver program to invoke the service. Upon receiving the request, *Firework* filters out the computing nodes that implement the requested services. Afterwards, *Firework* creates a new local job and dispatches the request to these computing nodes. Details about job creation and dispatch are explained in Section 2.2.2. By repeating this procedure, *Firework* instantiates a requested service by automatically creating a computation stream. The computation stream implements an application by leveraging computing resources along the data propagation path, which might include the edge devices and the cloud.

Considering the mobility and operational environment of edge devices, it is common that they may fail or change the network condition. To deal with failure or varying network conditions, *Firework* assigns a time-to-live interval to registered services and checks the liveness via heartbeat message periodically. A node will re-register its services after a failover or network condition change. When a node acting as *Firework.Manager* fails, it recovers all service stubs based on metadata from persistent logs. Specifically, it rebuilds the connections based on the out-of-date service stubs and updates these service stubs if the connections are restored successfully; otherwise, the service stubs are removed.

### 2.2.2 Job Management

A user can send *Firework.Manager* a request to start a service. Upon receiving the invocation, a local job is created by the *Job Management* layer, which initializes the service locally. For each job, a dedicated communication port is assigned for exchanging control messages. Note that this port is not used by executors for data communication. Next, *Firework.Manager* forwards the request to available *Firework.Node*s that implement the *Firework.View* of the requested service. Lastly, the local job is added to task queue waiting for execution. When a job is terminated by the user, the *Job Management* layer stops executors and releases the dedicated port of that job.

*Firework* provides elasticity of computing resource scaling via task reuse. In *Firework*, all services are public for all users, which potentially means that two different users could request the same service. In such a situation, *Firework* reuses the same running task by dynamically adding an output stream to the task. It is insignificant that the input streams of a service might come from different sources. Extra computing resources are allocated to a task if the resource utilization of an executor exceeds a threshold and vice versa. To reduce the I/O overhead of a task brought by communicating with multiple remote nodes, *Firework* uses a separate I/O manager,

```
/* FWView: an implementation of Firework.View */
public class FWView implements Runnable {
  protected String serviceName;
  protected FWInputStream[] inputStreams;
  protected FWOutputStream[] outputStreams;
  public FWView(String serviceName) {
    this.serviceName = serviceName;
    this.getFWInputStream();
    this.getFWOutputStream();
  }
  /* Get input streams from JobManager. */
  public void getFWInputStream() {
    inputStreams = JobManager.getInputStream(serviceName);
  }
  /* Get output streams from JobManager. */
  public void getFWOutputStream() {
    outputStreams = JobManager.getOutputStream();
  }
  /* Receive data from the input streams. */
  public Data[] read() {
    inputData = inputStreams.read();
  }
  /* Process the input data. */
  public Data[] compute(Data[] inputData) {
    // Do nothing by default.
    return inputData;
  }
  /* Send the processed data to other nodes. */
  public void write(Data[] outputData) {
    outputStream.write(outputData);
  }
  /* Run the computation procedure. */
  public void run() {
    while(true) {
      Data[] inputData = read();
      Data[] outputData = compute(inputData);
      write(outputData);
    }
  }
}
```

Listing 1. The Java code of FWView.

which will be introduced in Section 3, to perform data transmission so that a running service subscribes/publishes the input/output data from the I/O manager. In addition to the resource scaling, *Firework* also optimizes the workload among multiple nodes. A *Firework* node can inherit a base service without extending it. In this case, two consecutive nodes provide exactly the same service. If the node closer to data sources is overloaded, it can delay and offload computation to the other node, which might be less loaded. The offload decision aims to minimize the response latency of the service, which depends on resource utilizations (e.g., CPU, memory, and network bandwidth).

### 2.2.3 Executor Management

A task in *Firework* runs on an executor that has dedicated CPU, memory, and network resources. *Firework* nodes leverage heterogeneous computing platforms and consequently adopt different resource management approaches. Therefore, the *Executor Management* layer serves as an adapter that allocates computing resources to tasks. Specifically, some *Firework* nodes like smart phones or IoE gateways may adopt JVM or Docker [23], while some nodes like commodity servers may employ OpenStack [24] or VMWare, to host an executor. The executor management is fulfilled by the *Job Management* layer and operated by the *Executor Management* layer.

## 2.3 Programmability

*Firework* provides an easy-to-use programming interface for both developers and users so that they can focus on programming the user defined functions. An application on *Firework* includes two major parts: programs implementing *Firework.View*s and a driver program to deploy and interact with the application. A developer can decompose an application into subservices, such as data collecting on sensors, data preprocessing on the edge, and data aggregation in the cloud. Each subservice can be abstracted as a *Firework.View* and deployed on one or more computing nodes. By organizing them with a driver program, a user can achieve real-time analytics in a collaborative cloud-edge environment.

Specifically, *Firework* implements two basic programmable components, *FWView* and *FWDriver* that represent the *Firework.View* and driver program respectively. The *FWView* adopts a continuous execution model, in which an *FWView* continuously receives data from the input streams, processes the data, and sends out the data to other nodes. Listing 1 shows the Java code of the *FWView*. As mentioned in Section 2.2.1, when a *Firework.View* is registered, a service stub is created, which contains the metadata information. Thus, *FWView* could retrieve input and output streams from the *Job Management* layer, as shown by the *getFWInputStream()* and *getFWOutputStream()* functions in Listing 1. Note that the input streams of a service depend on the base services that provide input data for that service. The most important function of *FWView* is *compute()*, in which a user implements the payload. By calling the *run()* function, a *Firework* node repeats the actions of data receiving (via *read()*), processing (via *compute()*), and sending (via *write()*). Since an application on *Firework* is decomposed into several subservices, a developer needs to implement multiple *FWView*s, which perform different functionality.

The other basic programmable component of *Firework* is the *FWDriver*, which provides the capabilities to deploy and launch an application. In Listing 2, we illustrate the basic functionality of an application driver. The *deploy()*, *start()*, and *stop()* functions allow users to manage their applications, and the *retrieveResult()* function pulls final outcomes from a *Firework.Manager*. All these functions are conducted by *FWContext*, which maintains a session between a user and a *Firework* instance. The *DeployPlan* is a supplemental component of *FWDriver*, which describes an application-defined topology. Without providing a deployment plan, *Firework* uses the network topology as the default one, which might lead to redundant computation. A user can define a rule-based deployment plan to compose subservices as needed. An example of a deployment plan could be grouping sensors by regional areas so that a single *Firework* node processes all data in the same region, which is straightforward for certain application scenarios, especially when all sensors are owned by the same stakeholder. However, when a user employs subservices owned by multiple stakeholders, the underlying network topology might not be able to aggregate all data to the user. Therefore, *Firework* provides the *DeployPlan* for users to customize the data propagation routes.

By separating the implementation of the service and driver program, *Firework* allows a third party to leverage existing services by only providing a driver program. A user can also interact with an intermediate node (e.g., the edge node in the above example) to leverage the semi-finished data to build his/her own application. A case study of video analytics is addressed in Section 4 to demonstrate how to program with *Firework*.

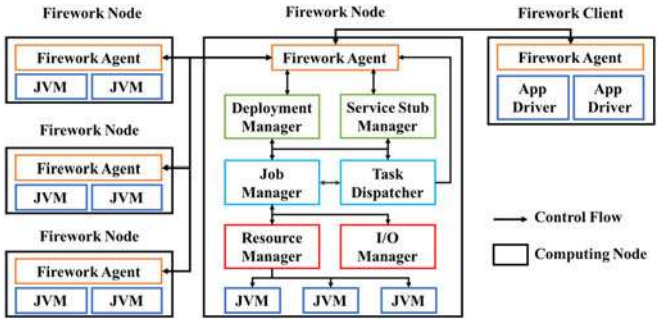



Fig. 4. An architecture overview of *Firework* with major system components.

### 2.4 Execution Model Comparison

We compare the execution model of *Firework* with *Cloud Computing* and *Edge Computing*. Specifically, *Firework* differs from cloud computing and edge computing in the following aspects: i) *Firework* provides virtual data sharing among multiple stakeholders and data processing across the edge and the cloud. In contrast to *Firework*, cloud computing focuses on centralized computation resource sharing and data processing, and edge computing focuses on manipulating local data with low latency and network bandwidth cost; ii) *Firework* allows data owners to define the functions that can be performed on their own data and shared with other stakeholders. Cloud computing collects data from users and defines the functions/services by the owners of clouds; iii) *Firework* reduces the network bandwidth cost by performing the computation at data sources; and iv) *Firework* leverages the cloud, as well as edge devices (and processing units placed close to the edge devices) so that the latency and network bandwidth cost can be reduced.

## 3 Implementation

We implement a prototype of *Firework* using Java. Fig. 4 shows an example architecture of our prototype system, which includes four *Firework* nodes and one *Firework* client. A *Firework* node fulfills the three-layered system design and a *Firework* client delegates an end user to communicate with *Firework* instance.

In the service management layer of a *Firework* node, the service registration is performed by the *Service Stub Manager* (shown in Fig. 4) built on *etcd* [25], which is a key/value store accessible through RESTful interface. When a service is registered on a *Firework.Manager*, the service access portal (e.g., service name and its IP address and port number) is stored in *etcd* for persistent storage, which will also be used for recovering from a failure. *Firework* maintains an in-memory copy of all the key/value pairs to reduce performance degradation caused by querying the *etcd* with REST requests. For the same service registered by multiple *Firework* nodes, we use the same *etcd* entry to store all the service access portals. To obtain the liveness of a registered service, *Firework* periodically sends heartbeat messages to all the portals and refreshes the time-to-live attributes and the list of live portals for the corresponding *etcd* entry. Another reason we choose *etcd* is that it provides a RESTful interface for a user to query available services. It is noteworthy that users can query any *Firework.Manager* to retrieve available services and compose

```
/* DeployPlan: the application defined topology. */
public class DeployPlan {
  private List<Rule> rules;
  /* Add a new rule to the deployment plan. */
  public void addRule(Rule rule) {
    rules.add(rule);
  }
}
/* FWDriver: Firework application driver. */
public class FWDriver {
  /* FWContext: creates a session between user and
       Firework.Manager */
  private FWContext fwContext;
  private DeployPlan deployPlan;
  private UUID uuid;
  private String serviceName;
  public FWDriver(String serviceName, DeployPlan
       deployPlan) {
    this.serviceName = serviceName;
    this.deployPlan = deployPlan;
    this.fwContext = new FWContext();
  }
  /* Deploy an application and get an UUID back. */
  public void deploy() {
    uuid = fwContext.configService(serviceName,
         deployPlan);
  }
  /* Launch a deployed service by uuid. */
  public void start(Parameter[] params) {
    fwContext.startService(uuid, params);
  }
  /* Stop an application by uuid. */
  public void stop() {
    fwContext.stopService(uuid);
  }
  /* Get the final results from Firework.Manager. */
  public Data[] retrieveResult() {
    return fwContext.retrieveResult(uuid);
  }
}
```

Listing 2. The Java code of FWDriver.

their own applications. Another component in the service management layer is the *Deployment Manager* (shown in Fig. 4), which decides if a *Firework* node satisfies the application defined deployment plan and informs the job management layer to launch services.

In the middle layer of a *Firework* node, the *Job Manager* (shown in Fig. 4) is responsible for task decomposition, scheduling, and optimization. First, a job request is analyzed to determine its dependencies (i.e., the services it relies on), and each dependency service is notified through the *Task Dispatcher* (optional) after applying the rules in the deployment plan. Then a local task is created and added to the task queue. In the current implementation, we use a first-come, first-serve queue for task scheduling. Finally, a task is submitted to an executor for execution. When a service is requested by multiple users, *Firework* reuses the existing running task by adding output streams to the task, where a centralized I/O manager is used in the executor management layer (explained in next paragraph) for stream reusing.

The bottom layer is the executor management layer, where we implement the *Resource Manager* and *I/O Manager* (shown in Fig. 4). When a task is scheduled to run, an executor (i.e., a JVM in our implementation) is allocated for it. It is noteworthy that we can extend the *Resource Manager* to be compatible with other resource virtualization tools (e.g., Docker [23], OpenStack [24]) by adding a corresponding adapter. The input and output of executors are carried out by the centralized *I/O Manager*, which is implemented as message queues. An executor subscribes to multiple queues as the input and output streams. The reasons to why we use a separate I/O manager are multifold. First, it is more efficient to manage the data transmission of an executor by dynamically adding or removing data streams to the

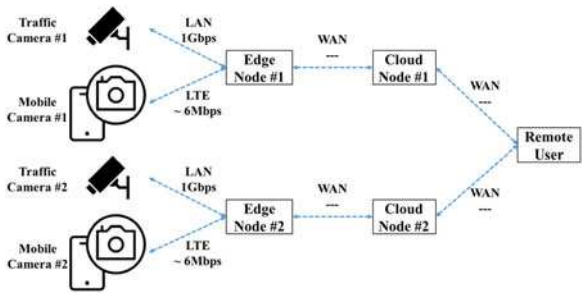



Fig. 5. Network topology of environmental testbed with available upload bandwidths.

message queue of the executor, which can be easily employed for task reuse. Second, by splitting the I/O management out from an executor, it reduces the programming effort of developers so that they can focus on the functionality. Third, such a design make it easy to leverage third party message queuing systems (e.g., Apache Kafka [12] and MQTT [13]). When there are a huge number of sensors reporting to a single aggregation node, it makes *Firework* more scalable by simply adding more aggregation nodes and subscribing to the queuing systems that guarantee exactly-once data processing semantics. Fourth, a unified system level security protection can be applied on top of the I/O communication to guarantee data integrity and fidelity. Therefore, a separate I/O manager is used in *Firework*.

By deploying on multiple computing nodes, an instance of the *Firework* system can be materialized. As shown in Fig. 4, multiple *Firework* nodes communicate with each other via the *Firework Agent* and form different topology based on an application-defined deployment plan. Note that we use a star topology in Fig. 4 as an example topology of a *Firework* instance. A user can interact with *Firework* using the utilities provided by the *Firework Client* and deploy multiple applications on the same *Firework* instance.

Up to this point, we have introduced the implementation details of the prototype system of *Firework*. In the next section, we will show a case study of edge video analytics on *Firework*.

## 4 Case Study: Edge Video Analytics

In the era of IoE, many cameras in either fixed locations (e.g., intersection, light pole, store) or mobile carriers (e.g., smartphone, vehicle), make video analytics a promising technology for public safety. A cloud-centric solution is not time- and cost-efficient for video analytics because it suffers from long response latency and high network bandwidth cost of data transportation, given zettabytes of video data. Another important concern is data privacy, which is unacceptable in certain scenarios to send the raw video data to the cloud. In this section, we illustrate a case study of edge video analytics using *Firework* for searching a target license plate in an urban area, which is common in the AMBER alert system. In such a scenario, when a license plate is wanted, it is very likely that this object is captured by cameras located at either fixed locations or mobile carriers in the urban area. In cloud computing, the video data captured by the cameras has to be uploaded to the cloud to identify the target license plate. However, the data transmission is still costly, which makes it extremely difficult and inefficient to leverage the wide area video data. With a *Firework* paradigm, a request of searching the target license plate is created at a remote user's device. Then the target license plate number is distributed to connected devices with cameras and each device performs the license plate searching using archived local data or real-time video stream. The requester gathers the results from other *Firework.Node*s to locate the object. With *Firework*, the video data is processed at the edge and the response latency and network bandwidth cost will be significantly reduced. An extension of this video analytics could be real-time object tracking or event detection, which is common in public safety applications, where the GPS information (e.g., smart phones and vehicles) can be used for multiple purposes.

### 4.1 Experimental Setup

We simplify the scenario and assume there are four types of computing nodes including camera, edge node, cloud node, and remote user. We deploy the edge video analytics application in the testbed shown in Fig. 5. We compare the performance in terms of response latency and network bandwidth cost between *Firework* with different deployment plans and a cloud-centric baseline. As a baseline, we implement a cloud-centric video analytics system, where a camera directly pushes video data to a cloud node and the cloud node detects and recognizes license plate for each video frame and sends the result to a remote user.

To simulate live video captured by cameras, we replay a prerecorded traffic video clip [26]. We randomly insert a license plate to a video frame when the video clip is repeatedly replayed. The video clip is transformed into four different resolution qualities including $1280 \times 720$ (720P), $1920 \times 1080$ (1080P), $2560 \times 1440$ (1440P), and $3840 \times 2160$ (2160P), and the frames per second is set to 30. The video data is encoded in H.264 format with a baseline profile and we configure that one intra-frame (IFrame) is followed by fifty-nine predictive-frames (PFrames) without bi-directional frame (BFrame), because we simulate a live video stream and cannot compute the differences between the current frame and the next frame. The data is sent to edge nodes using real-time transport protocol (RTP) over UDP/IP network. In our experiments, we calculate the average size of one video frame for 720P, 1080P, 1440P, and 2160P individually and the sizes are 15.08 KB, 29.38 KB, 51.63 KB, and 95.44 KB, respectively. Note that these sizes might vary in different runs.

As shown in Fig. 5, the cameras are connected to two edge nodes using LAN (the upload bandwidth is 1 Gbps) and LTE (the upload bandwidth is around 6 Mbps based on our speed test). The bandwidth of LAN is shared with other users and the available bandwidth for our experiments might vary over time. The edge nodes in Fig. 5 are located at a computer lab within Wayne State University's campus, which are two servers with the same hardware configuration, i.e., Intel E5620@2.4 GHz (8 cores, 16 threads) and 16 GB memory. The cloud nodes in Fig. 5 are two m4.4xlarge virtual machine instances in the Amazon EC2 data center located at US East (N. Virginia). The available network bandwidths of WANs are missing since they depend on the network traffic and certain service providers (e.g., Amazon EC2).

```
/* VideoStream collects video data on a camera. */
public class VideoStream extends FWView {
    private FWInputStream camera;
    private FWOutputStream outputStream;
    public VideoStream(String serviceName) {
        super(serviceName);
    }
    /* Collect data from a camera. */
    @Override
    public Data[] read() {
        return camera.readFrame();
    }
    /* Encode video data with h.264. */
    @Override
    public Data[] compute(Data[] frameData){
        return H264.encode(frameData);
    }
    /* Send video frames via RTP/UDP protocol. */
    @Override
    public void write(Data[] outputData){
        ArrayList<Data> rtpData=RTP.encode(outputData);
        foreach(Data data in rtpData){
            outputStreams.write(data);
        }
    }
}
```

Listing 3. The Java code of VideoStream (VS).

```
/* PlateRecognition reads numbers/letters in an image. */
public class PlateRecognition extends FWView {
    private FWInputStream inputStream;
    private FWOutputStream outputStream;
    public PlateRecognition(String serviceName) {
        super(serviceName);
    }
    /* Receive a plate image. */
    @Override
    public Data[] read() {
        Data[] recvData = inputStream.read();
        Data[] plateImage = JPEG.decode(recvData);
        return plateImage;
    }
    /* Read numbers and letters in a plate image. */
    @Override
    public Data[] compute(Data[] plateImage){
        Data[] plateNumber = LicensePlateRecog(plateImage);
        return plateNumber;
    }
    /* Send the plate number to a user. */
    @Override
    public void write(Data[] plateNumber) {
        outputStream.write(plateNumber);
    }
}
```

Listing 5. The Java code of PlateRecognition (PR).

### 4.2 Service Composition

We compose the edge video analytics application with four components, including *VideoStream* (VS), *PlateDetection* (PD), *PlateRecognition* (PR), and *VideoAnalytics* (VA). We implement these components in Java using FFmpeg [27] and modified openalpr [28]. Then, we package them into JAR files and deploy them to different computing nodes.

The VS (shown in Listing 3) collects video data and sends the video data from a camera to an edge node in H.264 format using real-time transport protocol over a UDP connection. The PD (shown in Listing 4) detects if a license plate appears in a video frame and sends a JPEG image containing the license plate to PR through TCP connection. The PR (shown in Listing 5) reads the numbers and letters from the image generated by PD and sends the plate number to a user.

The VA is a driver program that starts the entire application based on a deployment plan and it uses a default *FWDriver* implementation (shown in Listing 2). As aforementioned, subservices can be loaded on sensors, edge nodes and cloud nodes according to the deployment plan. In this case study, we deploy three subservices (i.e., VS, PD, and PR) and the driver program (i.e., VA) to the camera, edge node, cloud node, and remote user with various policies. We summarize all cases in Table 1. Given the network topology in Fig. 5, the VS and VA are always on cameras and remote user, respectively. However, the PD and PR can be deployed to either edge node, cloud node, or both of them. In Case#1, cameras upload video data to edge nodes. The edge nodes detect and recognize license plates and send the results first to cloud nodes even though the data is not manipulated on the cloud nodes. Finally, the cloud nodes forward the results to a remote user. Similarly, the edge nodes in Case#3 only forward the video data to cloud nodes and the cloud nodes perform the entire computation (i.e., PD and PR). Note that Case#3 is similar to the cloud-centric baseline solution, but in the baseline, a camera directly pushes video data to cloud nodes. In Case#2, the edge nodes detect if a video frame contains a license plate and only send the cropped frames with license plates to cloud nodes. Otherwise, video frames without license plates are dropped to reduce network transmission cost.

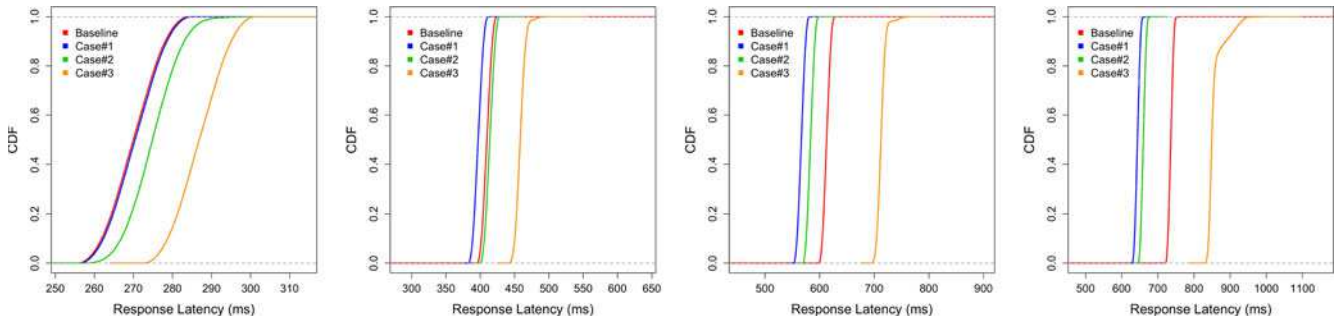
```
/* PlateDetection detects lincese plate in a frame. */
public class PlateDetection extends FWView {
    private FWInputStream RTPStream;
    private FWOutputStream outputStream;
    public PlateDetection(String serviceName) {
        super(serviceName);
    }
    /* Retrieve a video frame from a RTP/UDP stream. */
    @Override
    public Data[] read() {
        Data[] frameData;
        ArrayList<Data> recvData = new ArrayList<>();
        while((frameData = H264.decode(recvData))==null){
            recvData.add(RTPStream.read().decode());
        }
        return frameData;

    }
    /* Detect license plate in a video frame. */
    @Override
    public Data[] compute(Data[] frameData){
        Data[] plateImage = LicensePlateDetect(frameData);
        return plateImage;
    }
    /* Send images containing lincese plate using JPEG
        over TCP. */
    @Override
    public void write(Data[] plateImage) {
        outputStream.write(JPEG.encode(plateImage));
    }
}
```

Listing 4. The Java code of PlateDetection (PD).

Listing 6 illustrates the major components of the deployment plan for Case#2 using a JSON format. In the "topology", it describes nodes hosting subservices with the corresponding network address and port number, as well as the downstream node of the nodes. The subservices of a service is described in "subservice", which is recursively defined. As shown in Listing 6, the remote user hosts VA and VA depends on PR, which is carried out by two cloud nodes. The PR further depends on the PD running at two edge nodes. Note that the downstream node of these two edge nodes is different. Lastly, VS are invoked on four cameras. Based on the deployment plan, each node will start the

TABLE 1
Subservice Deployment Plans

| Case | Camera | Edge Node | Cloud Node | Remote User |
|---|---|---|---|---|
| Case#1 | VS | PD&PR | – | VA |
| Case#2 | VS | PD | PR | VA |
| Case#3 | VS | – | PD&PR | VA |




(a) 720P video stream (b) 1080P video stream (c) 1440P video stream (d) 2160P video stream

Fig. 6. Response latencies regarding different video resolution qualities using a LAN connection.

services running locally and invokes its subservices via service discovery. For Case#1, PD and PR are deployed on the edge nodes and the cloud nodes run a base *Firework.View* that forwards input data to downstream nodes. Similarly, the edge nodes in Case#3 just forward input data to the cloud nodes.

```
{
  "service_name": "video_analytics",
  "topology": [
    {
      "node_id": "remote_user",
      "listen": "192.168.1.100:12999",
      "downstream": ""
    }
  ],
  "subservice": [
    {
      "service_name": "plate_recognition",
      "topology": [
        {
          "node_id": "cloud_node_1",
          "listen": "172.16.1.100:12000",
          "downstream": "remote_user"
        },
        {
          "node_id": "cloud_node_2",
          "listen": "172.16.1.200:12000",
          "downstream": "remote_user"
        }
      ],
      "subservice": [
        {
          "service_name": "plate_detection",
          "topology": [
            {
              "node_id": "edge_node_1",
              "listen": "10.1.1.100:13000",
              "downstream": "cloud_node_1"
            },
            {
              "node_id": "edge_node_2",
              "listen": "10.1.1.200:13000",
              "downstream": "cloud_node_2"
              }
          ],
          "subservice": [
            {
              "service_name": "video_stream",
              "topology": [
                {
                  "node_id": [
                    "traffic_camera_1",
                    "mobile_camera_1"
                  ],
                  "listen": "",
                  "downstream": "edge_node_1"
                },
                {
                  "node_id": [
                    "traffic_camera_2",
                    "mobile_camera_2"
                  ],
                  "listen": "",
                  "downstream": "edge_node_2"
                }
              ],
              "subservice": []
            }
          ]
        }
      ]
    }
  ]
}
```

Listing 6. The deployment plan for Case\#2 in JSON format.

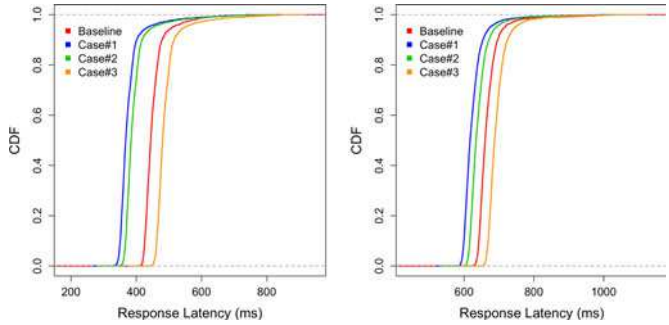
### 4.3 Response Latency

The response latency in our experiments is defined as the time duration between when a video frame is sent out by a camera and a remote user receives the decision if this video frame contains the target license plate. We compare the response latencies under different deployment plans and network conditions. We collect the response latencies for 20,000 video frames that contain license plates in each experiment explained below.

Fig. 6 shows the response latencies regarding different video resolutions using LAN connection. In general, the more workload is offloaded to edge nodes, the lower response latency is achieved for any given video resolution. Compared to the baseline, the higher the video resolution is, the lower the response latency is achieved. More specifically, when the video resolution is relatively low (i.e., 720P shown in Fig. 6a), Case#1 achieves similar response latency as the baseline. Case#3 has higher latency than the baseline but only increased by 17 ms on average. For the video stream of 1080P shown in Fig. 6b, Case#1 outperforms the baseline and Case#2 achieves similar performance as the baseline. When the video resolution is relatively high (i.e., 1440P and 2160P in Fig. 6c and 6d), Case#1 and Case#2 have lower response latencies than the baseline. The response latencies are reduced by up to 6.42 percent and 11.62 percent for 1440P and 2160P, respectively. Case#3 has the highest response latency in all scenarios since the video data is first pushed to the edge nodes and then forwarded to the cloud nodes so that the data transmission time is much higher than the other cases.

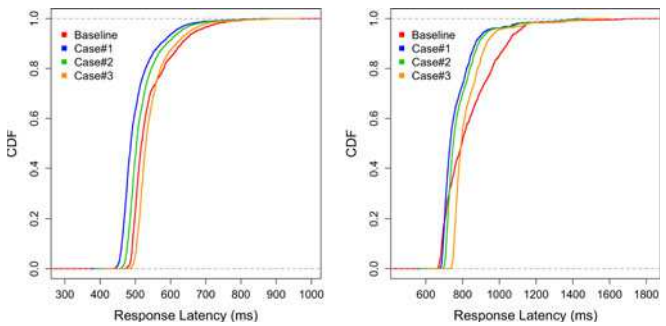
When cameras are connected with an LTE connection, the video data can be uploaded to either edge nodes or cloud nodes only when the video resolution is low (i.e., 720P and 1080P). Transmitting video data with higher resolutions requires higher upload network bandwidth, which is limited with an LTE connection. The LTE connection also suffers from a significantly high frame loss rate so that the edge nodes and cloud nodes cannot detect and recognize a license plate with incomplete frame data (explained in Section 4.4). Therefore, we only show the response latencies for 720P and 1080P video streams when using an LTE connection.

In Fig. 7, we show the response latency when cameras are physically fixed in one location and connected to a static LTE connection. Which means, there is no cellular base station switching during the experiments. The response latencies are reduced by up to 19.52 percent and 8.15 percent




(a) 720P video stream (b) 1080P video stream

Fig. 7. Response latencies regarding different video resolution qualities using a *static* LTE connection.




(a) 720P video stream (b) 1080P video stream

Fig. 8. Response latencies regarding different video resolution qualities using a *dynamic* LTE connection.

for 720P and 1080P video streams, respectively. Case#3 achieves higher latencies than the baseline for both 720P and 1080P video streams since the video data is transmitted to the cloud nodes through the edge nodes.

We also conduct the experiments in a driving condition to simulate a vehicle video analytics scenario, where the LTE connection is dynamically changing and involves cellular base station switching. We denote this as *dynamic LTE connection* in the rest of this paper. We upload video data on a vehicle driving at 35 miles per hour circling around the urban area of Detroit. Fig. 8 illustrates the response latencies when using a dynamic LTE connection. Similarly, Case#1 also outperforms the baseline solution by reducing up to 11.12 percent and 7.9 percent of response latencies for 720P and 1080P video streams, respectively. Differentiating from the case of a static LTE connection, a dynamic LTE connection suffers from long tail latencies and higher variation due to varying signal strength in a moving vehicle and cellular base station switching.

We further break down the response latency into three parts, including transmission time, encoding/decoding time, and detection/recognition time. Fig. 9 shows the average response latency under different scenarios. Generally, the detection and recognition time contributes to the majority of response latency with a LAN connection, while the data transmission time contributes to the majority with an LTE connection. Case#3 has the highest transmission time with any video resolution for both LAN and LTE connections. This is because edge nodes receive and forward video data to cloud nodes instead of directly uploading the video data to cloud nodes from cameras. This also leads to higher response latency as explained in Fig. 6.

More specifically, as shown in Fig. 9, the transmission time and encoding/decoding time increase as the video resolution enhances using either a LAN or LTE connection. When the video resolution is high (i.e., 1440P or 2160P) with the LAN connection, Case#1 and Case#2 reduce the transmission time since the video data is only uploaded to edge nodes that are closer to the cameras in terms of network distance compared to cloud nodes. When using an LTE connection, the transmission time of Case#1 is reduced significantly compared to the baseline even with relatively low video resolutions. However, the average response latencies for both 720P and 1080P video streams are increased when using an LTE connection compared to the cases with a LAN connection because the bandwidth of the LTE connection is much less than that of a LAN connection.

### 4.4 Frame Loss

In addition to the response latency, we count the number of lost frames under LAN and LTE connections during the experiments. When using a LAN connection, the frame loss rate is less than 0.9 percent for all four different video resolutions. However, when using an LTE connection, the frame loss rates of 720P and 1080P are less than 2.73 percent, while for 1440P and 2160P, the frame loss rates are more than 63.71 percent, due to the limited network bandwidth, which makes it impossible to decode a valid frame based on incomplete IFrames or PFrames. Note that the network package loss rate of an LTE connection when uploading 1440P and 2160P is less than 9.02 percent, which is much lower than the frame loss rate. Given the size of one video frame, it is common that the data of one video frame is transmitted using multiple network packages. We count one frame loss if at least one of the packages of the frame is lost. Moreover, one IFrame loss incurs the loss of the following fifty-nine PFrames since each video frame is decoded by

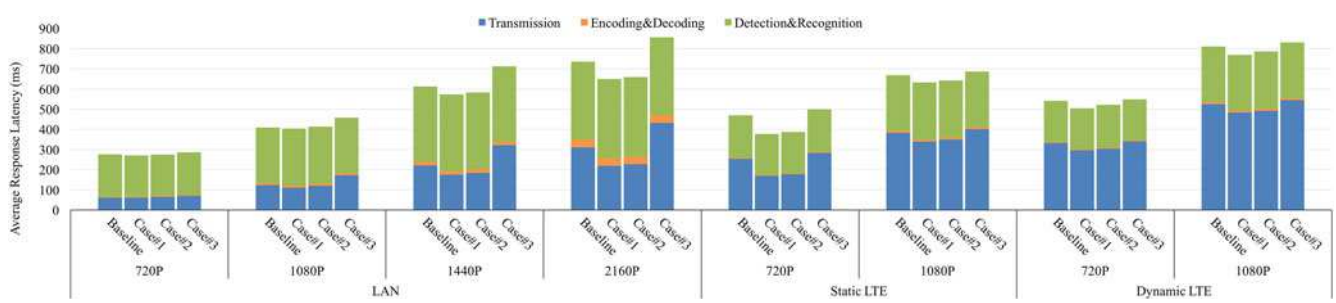



Fig. 9. The time breakdown of average response latencies for different deployment plans, video resolutions, and network conditions.

TABLE 2
Network Bandwidth Cost Reductions Compared to Baseline

| | Case#1 | Case#2 | Case#3 |
|---|---|---|---|
| 720P | ~100% | 73.47% | 0% |
| 1080P | ~100% | 72.77% | 0% |
| 1440P | ~100% | 78.69% | 0% |
| 2160P | ~100% | 86.38% | 0% |

using both IFrame and PFrame. Therefore, the frame loss rate is much higher than the network package loss rate.

### 4.5 Network Bandwidth Cost

With respect to the network bandwidth cost, *Firework* leverages edge nodes to perform part of the computation so that the amount of data transmitted over edge-to-cloud connections (i.e., the connections between EdgeNode and CloudNode in Fig. 5) is reduced. Table 2 summarizes the bandwidth cost reductions of the edge-to-cloud connections compared to the baseline solution for the three cases in Table 1, regarding four different video resolutions. In Case#1, the bandwidth cost is almost eliminated since all computation (i.e., PD and PR) is performed on the edge nodes and only negligible data (i.e., the recognition result) is sent to the cloud nodes. Thus, compared to the baseline in which the entire video data is transmitted to the cloud nodes, the bandwidth cost reduction is close to 100 percent. On the contrary, in Case#3 the edge nodes transmit the entire video data to the cloud nodes, thus the network bandwidth cost is the same as the baseline, which means 0 percent reduction.

For Case#2, the reduction rates vary over video resolutions. In Case#2, data transmitted over the edge-to-cloud connections is the extracted area of the license plate in a video frame. The reduction rate is also affected by the size of the area of a license plate, which varies significantly over different video resolutions and the distance between a camera and an object (e.g., a license plate). Therefore, we take the average size of the output images generated by *PlateDetection* on the edge nodes to compute the reduction rate. To calculate the reduction rate for Case#2, we assume that every video frame contains exactly one license plate, in which the bandwidth cost reduction is the lower bound because a video frame without a license plate is dropped. In an extreme case, the reduction rate is close to 100 percent (i.e., no license plate appears in any video frame). Given the aforementioned assumption for Case#2, Table 2 shows that the higher the video resolution quality, the larger reduction rate that is achieved. This is because most areas of a video frame are cropped except the license plate so that the network bandwidth reduces significantly when video resolution is higher.

## 5 Discussion

In this paper, we narrow down the scope of *Firework* to prototyping and programming interface implementation. In this section, we discuss potential issues and limitations of *Firework*, in terms of system design and performance optimization.

*Privacy*: Data captured by IoE devices can contain private information, e.g., GPS data, streams of video or audio, which might be used for complex analytics somewhere other than where the data is generated. Thus, it is critical that only data that is privacy compliant be sent to the edge or the cloud for further analysis. As we mentioned, *Firework* supports the privacy preserving function, which can be achieved by implementing a function, such as face blurring of video frames in [29], [30], as a predefined function of a *Firework.View*. Since a privacy preserving function is attached to the shared subservices of each service owner, it is feasible for a downstream subservice to apply different privacy policies by extending existing subservices (i.e., extending a *Firework.View* to add/override existing privacy preserving functions). In addition, *Firework* manages data communication using a separate I/O controller, where an easy security enhancement can be added by using secure communication protocols.

*Fault Tolerant*: In the prototype of *Firework*, the *Job Manager* (shown in Fig. 4) tries to restart a job when the job fails due to software failure (e.g., out of memory, uncaught exceptions). However, the *Job Manager* cannot restart a job when the underlying hardware fails. In our license plate recognition example, if all cameras fail and the VS is unavailable, the PD and PR are still running on the edge nodes and/or the cloud but a user cannot get any output. In such a case, *Firework* restores the VS whenever a camera is restored. Since *Firework* leverages computing resources that are owned by stakeholders and not controlled by *Firework*, there is no guarantee that an unavailable subservice would be available in the near future. Thus, the fault tolerance in *Firework* depends on the underlying fault tolerant mechanisms of stakeholders that might be very different. Thus, we leave the fault/failure detection in *Firework* as future work.

*Optimization*: To simplify the scenario, we assume that an application can be decomposed and represented by a sequence of $n$ functions, and $m$ computing nodes are connected in a line. The goal of optimizing computation offload is to minimize the end-to-end latency by optimizing the allocation of $n$ functions over $m$ nodes, where the functions have to be allocated sequentially. In the current implementation of *Firework*, we use a simple deployment policy, in which the optimization target is a weighted sum of response latency and network bandwidth cost. Using the default deployment policy, it is possible to assign all subservices on one edge node (e.g., a smart phone), in which case the response latency (e.g., only including the time used for license plate detection and recognition) and network bandwidth cost (e.g., there is no data transmitted through a network since all data are consumed locally on the smart phone) are minimized. However, it leads to high power consumption, which is infeasible and leads to short battery life. Furthermore, automatic functionality decomposition of an application increases the difficulty of optimizing the function placement because the optimization goal of function decomposition might be in contrast to that of function placement. Therefore, we leave the automatic functionality decomposition and workload placement/migration as a future work so that *Firework* provides an efficient algorithm that co-optimizes these goals with little user intervention.

## 6 Related Work

In this section, we discuss the related work in the following primary areas: the stream processing system, mobile cloud, crowdsourcing, and edge computing.

*Stream Processing*: To cope with unbounded continuous data, stream processing systems [4], [5], [31], [32] have been explored and employed massively. Storm [31] divides a streaming application into several stages and each stage is carried out by meshed and dedicated operators that adopt a continuous operator model, which is the most similar to the data processing model in *Firework*. However, in most cases, the operators in Storm are deployed in the same data facility running over homogeneous hardware platforms with centralized resource management, while *Firework* leverages heterogeneous resources so that operators could be implemented on different hardware and software platforms. *Firework* also provides layered distributed service management to support dynamic topology composition. Furthermore, the data in a Storm application cannot be shared with other applications at the operator level, while *Firework* achieves such data sharing by reusing existing tasks. A recent work in [33] extends Storm so that it can be deployed at edge nodes that are close to the interacting objects (e.g., sensor, on-site database, backend database) of an operator, to reduce response latency. However, the proposed framework in [33] is application specific and the operators cannot be shared among multiple applications.

To provide comprehensive support for IoE applications, cloud-centric stream processing frameworks [14], [15], [34], [35] have been proposed. The shared underlying architecture is that the data generated by IoE devices are aggregated and delivered using data ingestion systems (e.g., Apache Kafka, MQTT, and Amazon Kinesis Firehose) to analytics systems in the cloud. Analyzing all IoE data in the cloud suffers from large data transmission latency and consumes a lot of network bandwidth. In *Firework*, the cloud can offload a substantial amount of work to edge nodes to reduce both response latency and transmission cost.

*Mobile Cloud*: In mobile computing, several works have been studied to offload part of a task on a mobile device to remote cloud. MAUI [36] proposes a fine-grained code offload with managed runtime management to improve energy efficiency on mobile devices. COSMOS [37] achieves high speedup by efficient resource allocation between a mobile device and the cloud. Cuckoo [38] proactively executes methods at reachable cloud resources via intercepting the interprocess communication (IPC) in Android operating system and redirecting the local method call to a remote cloud, to improve performance and reduce energy consumption on the smartphones. However, these systems cannot collaborate among multiple mobile devices.

*Crowdsourcing*: Mobile devices are widely used in crowdsourcing systems, which outsource a single task to a crowd of people for contributions. By leveraging an online crowdsourcing platform, i.e., Amazon Mechanical Turk [39], numerous crowdsourcing tools have been proposed. TurKit [40] provides JavaScript-like programming language for Amazon Mechanical Turk to automatically schedule and price tasks, and accept or reject results. CrowdDB [41] extends SQL to embed human input into a query that only machines/databases can adequately answer. Since human labor is involved in crowdsourcing, different game theory based incentive mechanisms are studied in [42] and [43], which maximize a worker's profit, to raise the participation of more workers. In *Firework*, we focus on processing data without human effort in loop but provide easy-to-use programming interface for users to interact with existing services available geographically.

*Edge Computing*: Inspired by low-latency analytics, edge computing [11] (a.k.a. fog computing [9], cloudlet [10]) is proposed to process data at the proximity of data sources. To leverage computing resources on edge nodes, mobile device cloud [44], femto clouds [45], mobile edge-clouds [46], and Foglets [47] have been proposed to orchestrate multiple edge devices for intensive applications that are difficult to run on a single device. Different from these systems, *Firework* leverages not only mobile devices and the cloud, but also edge nodes to complete big data processing tasks collaboratively, while the aforementioned systems are not for large-scale data processing and sharing among multiple stakeholders.

GigaSight [29] has been proposed as a reversed content distribution network using VM-based cloudlets for scalable crowd-sourcing of video from mobile devices. GigaSight collects personal video at the edges of Internet with denaturing for privacy that automatically applies a contributor-specific privacy policy. The captured video in GigaSight is tagged for search and shared using network file system (NFS). However, GigaSight is designed to share video data and cannot apply video analytics functions. In contrast to GigaSight, *Firework* provides APIs for data owners to create customized video analytics functions, which can be used by a user to compose his/her IoE application.

Vigil [48] is a distributed wireless surveillance system that prioritizes video frames that are most relevant to the user's query and maximizes the number of query-specified objects while minimizing the wireless bandwidth cost. Vigil partitions video processing (e.g., object/face recognition or trajectory synthesis) between edge nodes and the cloud with a fixed configuration. Differencing from this prior work, *Firework* allows a user to define workload partitioning and deployment and provides dynamic workload migration (e.g., JVM migration) depending on the available resources on the edge nodes and the cloud.

Wang et al. [30] propose OpenFace that is an open-source face recognition framework to provide real-time face recognition/tracking by using edge computing (i.e., cloudlet [10]). Integrated with video stream denaturing, OpenFace selectively blurs faces depending on a user-specific privacy policy. However, OpenFace leverages only edge computing whose computation is conducted on edge nodes. In contrast to OpenFace, *Firework* leverages both edge nodes and the cloud to reduce the response latency and network bandwidth cost. A programming interface is provided to manipulate data from multiple data sources.

Panoptes [49] presents a cloud-based view virtualization system to share steerable cameras among multiple applications by moving the cameras in a timely manner to the expected view for each application. A mobility-aware scheduler prioritizes virtualized views based on motion prediction to minimize the impact on application performance caused by camera moving and network latency. Zhang et al. [50]

propose VideoStorm to support real-time video streams analytics over large clusters. An offline profiler generates a query-resource quality profile, and an online scheduler allocates resources to each query to maximize performance on quality and lag based on the quality profile. Resource demand for a query can be reduced by sacrificing the lag, accuracy, and quality of outputs. These prior works are orthogonal to *Firework*. Panoptes can be adopted by *Firework.View* (e.g., integrated with VS in license plate recognition) to provide customized camera views which are most relevant to user interests. The online scheduling algorithm of VideoStorm can also be used to adjust the resources allocated for a *Firework.View*. Furthermore, *Firework* can reduce impact on application performance (e.g., latency and network bandwidth cost) by carrying out the analytics on edge nodes, while both Panoptes and VideoStorm assume video data are preloaded in the cloud and clusters, which is infeasible given the scale of zettabytes data.

Ananthanarayanan et al. [18] present a geo-distributed framework for large-scale video analytics that can meet the strict requirements of real-time. The proposed framework in [18] leverages public cloud, private clusters, and edge nodes to carry out different computation modules of vision analytics. The prior works, Panoptes [49] and VideoStream [50], are integrated with the framework to optimize the resource allocation and minimize latency. *Firework* differs from [18] because our work expands data sharing along with attached computing modules (e.g., functions in a *Firework.View*) and provides programming interfaces for users to compose their IoE application.

## 7 Conclusion

Real-time video analytics becomes more and more important to IoE applications due to the richness of video content and the huge potential of unanticipated value. To undertake the barriers of deploying IoE applications, we introduce a new computing framework called *Firework* that is a data processing and sharing platform for hybrid cloud-edge analytics. We illustrate the system design and implementation and demonstrate the programmability of *Firework* so that users are able to compose and deploy their IoE applications over various computing resources at the edge of the network and in the cloud. The evaluation of an edge video analytics application shows that *Firework* reduces response latencies and network bandwidth cost when using either a LAN or LTE connection, compared to a cloud-centric solution. For a future work, we will explore automatic service/functionality decomposition so that *Firework* could dynamically optimize the subservice deployment according to the usage of computing, network, and storage resources on computing nodes.

## Acknowledgments

This work is supported in part by US National Science Foundation grant CNS-1741635.

## References

[1] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proc. Symp. Mass Storage Syst. Technol.*, 2010, pp. 1–10.

[2] Apache storm. [Online]. Available: https://storm.apache.org/, Accessed on: Apr. 20, 2016.

[3] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proc. USENIX Conf. Hot Topics Cloud Comput.*, 2010, vol. 10, Art. no. 10.

[4] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proc. Symp. Operating Syst. Principles*, 2013, pp. 423–438.

[5] Q. Zhang, Y. Song, R. R. Routray, and W. Shi, "Adaptive block and batch sizing for batched stream processing system," in *Proc. Int. Conf. Autonomic Comput.*, Jul. 2016, pp. 35–44.

[6] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[7] 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_Index_White_Paper.pdf, Accessed on: Apr. 20, 2016.

[8] D. Evans, "The internet of things: How the next evolution of the internet is changing everything," *CISCO White Paper*, vol. 1, pp. 1–11, 2011.

[9] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proc. Mobile Cloud Comput.*, 2012, pp. 13–16.

[10] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for VM-based cloudlets in mobile computing," *IEEE Pervasive Comput.*, vol. 8, no. 4, pp. 14–23, Oct.-Dec. 2009.

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

[12] J. Kreps, N. Narkhede, and J. Rao, "Kafka: A distributed messaging system for log processing," in *Proc. NetDB*, 2011, pp. 1–7.

[13] U. Hunkeler, H. L. Truong, and A. Stanford-Clark, "MQTT-S: A publish/subscribe protocol for wireless sensor networks," in *Proc. Int. Conf. Commun. Syst. Softw. Middleware Workshops*, 2008, pp. 791–798.

[14] Amazon Kinesis. [Online]. Available: https://aws.amazon.com/kinesis/, Accessed on: Sep. 1st, 2016.

[15] Apache Quarks. [Online]. Available: http://quarks.incubator.apache.org/, Accessed on: Sep. 1st, 2016.

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

[17] W. Shi and S. Dustdar, "The promise of edge computing," *IEEE Comput. Mag.*, vol. 49, no. 5, pp. 78–81, May 2016.

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

[19] Q. Zhang, X. Zhang, Q. Zhang, W. Shi, and H. Zhong, "Firework: Big data sharing and processing in collaborative edge environment," in *Proc. Workshop Hot Topics Web Syst. Technol.*, Oct. 2016, pp. 20–25.

[20] R. Agrawal and R. Srikant, "Privacy-preserving data mining," *ACM SIGMOD Rec.*, vol. 29, no. 2., pp. 439–450, 2000.

[21] P. Hintjens, *ZeroMQ: Messaging for Many Applications*. Sebastopol, CA, USA: O'Reilly Media, Inc., 2013.

[22] RabbitMQ. [Online]. Available: https://www.rabbitmq.com/, Accessed on: Dec. 1, 2016.

[23] Docker. [Online]. Available: https://www.docker.com/, Accessed on: Sep. 1, 2016.

[24] Openstack. [Online]. Available: https://www.openstack.org/, Accessed on: Sep. 1, 2016.

[25] etcd. [Online]. Available: https://github.com/coreos/etcd, Accessed on: Sep. 1, 2016.

[26] M. Popovski, Random cars driving by 4k stock video. [Online]. Available: https://www.videezy.com/urban/4298-random-cars-driving-by-4k-stock-video, Accessed on: Sep. 1., 2016.

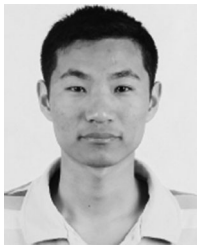
[27] FFmpeg. [Online]. Available: https://ffmpeg.org/, Accessed on: Dec. 1, 2016.

[28] Openalpr. [Online]. Available: https://github.com/openalpr, Accessed on: Dec. 1, 2016.

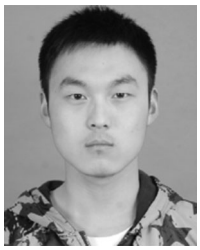
[29] P. Simoens, Y. Xiao, P. Pillai, Z. Chen, K. Ha, and M. Satyanarayanan, "Scalable crowd-sourcing of video from mobile devices," in *Proc. Int. Conf. Mobile Syst., Appl., Serv.*, 2013, pp. 139–152.

[30] 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 *Proc. Multimedia Syst. Conf.*, 2017, pp. 38–49.

[31] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donhamet al., "Storm@ twitter," in *Proc. Int. Conf. Manage. Data*, 2014, pp. 147–156.
[32] Samza. [Online]. Available: http://samza.apache.org/, Accessed on: Sep. 1, 2016.
[33] A. Papageorgiou, E. Poormohammady, and B. Cheng, "Edge-computing-aware deployment of stream processing tasks based on topology-external information: Model, algorithms, and a storm-based prototype," in *Proc. Int. Congr. Big Data*, 2016, pp. 259–266.
[34] Google cloud platform: IoT solution. [Online]. Available: https://cloud.google.com/solutions/iot/, Accessed on: Sep. 1, 2016.
[35] AWS IoT. [Online]. Available: https://aws.amazon.com/iot/, Accessed on: Sep. 1, 2016.
[36] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: Making smartphones last longer with code offload," in *Proc. Int. Conf. Mobile Syst. Appl. Serv.*, 2010, pp. 49–62.
[37] C. Shi, K. Habak, P. Pandurangan, M. Ammar, M. Naik, and E. Zegura, "COSMOS: Computation offloading as a service for mobile devices," in *Proc. Int. Symp. Mobile Ad-hoc Netw. Comput.*, 2014, pp. 287–296.
[38] R. Kemp, N. Palmer, T. Kielmann, and H. Bal, "Cuckoo: A computation offloading framework for smartphones," in *Proc. Int. Conf. Mobile Comput., Appl., Serv.*, 2010, pp. 59–79.
[39] Amazon Mechanical Turk. [Online]. Available: https://www.mturk.com/mturk/welcome, Accessed on: Sep. 1, 2016.
[40] G. Little, L. B. Chilton, M. Goldman, and R. C. Miller, "Turkit: Human computation algorithms on mechanical turk," in *Proc. Symp. User Interface Softw. Technol.*, 2010, pp. 57–66.
[41] M. J. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin, "CrowdDB: Answering queries with crowdsourcing," in *Proc. Int. Conf. Manag. Data*, 2011, pp. 61–72.
[42] D. Yang, G. Xue, X. Fang, and J. Tang, "Crowdsourcing to smartphones: Incentive mechanism design for mobile phone sensing," in *Proc. Annu. Int. Conf. Mobile Comput. Netw.*, 2012, pp. 173–184.
[43] Y. Zhang and M. van der Schaar, "Reputation-based incentive protocols in crowdsourcing applications," in *Proc. Int. Conf. Comput. Commun.*, 2012, pp. 2140–2148.
[44] A. Fahim, A. Mtibaa, and K. A. Harras, "Making the case for computational offloading in mobile device clouds," in *Proc. Conf. Mobile Comput. Netw.*, 2013, pp. 203–205.
[45] K. Habak, M. Ammar, K. A. Harras, and E. Zegura, "Femto clouds: Leveraging mobile devices to provide cloud service at the edge," in *Proc. Int. Conf. Cloud Comput.*, 2015, pp. 9–16.
[46] N. Fernando, S. W. Loke, and W. Rahayu, "Computing with nearby mobile devices: A work sharing algorithm for mobile edge-clouds," *IEEE Trans. Cloud Comput.*, vol. PP, no. 99, p. 1, 2017, doi: 10.1109/TCC.2016.2560163.
[47] 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 *Proc. Int. Conf. Distributed Event-Based Syst.*, 2016, pp. 258–269.
[48] T. Zhang, A. Chowdhery, P. V. Bahl, K. Jamieson, and S. Banerjee, "The design and implementation of a wireless video surveillance system," in *Proc. Int. Conf. Mobile Comput. Netw.*, 2015, pp. 426–438.
[49] S. Jain, V. Nguyen, M. Gruteser, and P. Bahl, "Panoptes: Servicing multiple applications simultaneously using steerable cameras," in *Proc. Int. Conf. Inform. Process. Sensor Netw.*, 2017, pp. 119–130.
[50] 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 *Proc. USENIX Symp. Networked Syst. Des. Implementation*, 2017, pp. 377–392.



**Quan Zhang** received the PhD degree from the Department of Computer Science, Wayne State University, in 2018, and the MS degree in computer science from Wayne State University, in 2016. Now he is a data engineer with Salesforce. His research interests include cloud computing, edge computing, real-time streaming processing, and energy-efficient systems.



**Qingyang Zhang** received the BEng degree in computer science and technology from Anhui University, China, in 2014. He is currently working toward the PhD degree at An-hui University. He is also a visiting student in Wayne State University. His research interest includes edge computing, and security protocol for wireless network.

**Weisong Shi** received the BS degree from Xidian University, in 1995, and the PhD degree from the Chinese Academy of Sciences, in 2000, both in computer engineering. He is a Charles H. Gershenson distinguished faculty fellow and a professor of computer science with Wayne State University. His research interests include edge computing, computer systems, energy-efficiency, and wireless health. He is a recipient of the National Outstanding PhD dissertation award of China and the NSF CAREER award. He is a fellow of the IEEE and ACM distinguished scientist.

**Hong Zhong** received the BS degree in applied mathematics from Anhui University, China, in 1986, and the PhD degree in computer science and technology from the University of Science and Technology of China (USTC), China, in 2005. Now she is a professor and PhD advisor with Anhui University. Her research interests cover network and information security.