2017 IEEE 9th International Conference on Cloud Computing Technology and Science

Supporting Internet-of-Things Analytics in a Fog Computing Platform

Hua-Jun Hong[1], Pei-Hsuan Tsai[1], An-Chieh Cheng[1], Md Yusuf Sarwar Uddin[2], Nalini Venkatasubramanian[2], and Cheng-Hsin Hsu[1]
[1]Department of Computer Science, National Tsing Hua University, Taiwan
[2]Department of Computer Science, University of California Irvine, CA, USA

***Abstract*—Modern IoT analytics are computational and data intensive. Existing analytics are mostly hosted in cloud data centers, and may suffer from high latency, network congestion, and privacy issues. In this paper, we design, implement, and evaluate a fog computing platform that runs analytics in a distributed way on multiple devices, including IoT devices, edge servers, and data-center servers. We focus on the core optimization problem: making deployment decisions to maximize the number of satisfied IoT analytics. We carefully formulate the deployment problem and design an efficient algorithm, named SSE, to solve it. Moreover, we conduct a detailed measurement study to derive system models of the IoT analytics based on diverse QoS levels and heterogeneous devices to facilitate the optimal deployment decisions. We implement a testbed to conduct experiments, which show that the system models achieve reasonably good accuracy. More importantly, 100% of the deployed IoT analytics satisfy the QoS targets. We also conduct extensive simulations for larger-scale scenarios. The simulation results reveal that our SSE algorithm outperforms a state-of-the-art algorithm by up to 89.4% and 168.3% in terms of the number of satisfied IoT analytics and active devices. In addition, our SSE algorithm reduces CPU, RAM, and network resource consumptions by 18.4%, 12.7%, and 898.3%, respectively, and terminates in polynomial time.**

I. Introduction

Internet-of-Things (IoT) is getting tremendously popular: the number of IoT devices worldwide is expected to reach 50.1 billion by 2020 [7]. Modern IoT devices, such as Google Home [4] and Amazon Echo [1], enable IoT *analytics*, which collect, interpret, extract, and communicate hidden patterns from diverse sensor data. These IoT analytics are becoming more computational and data-intensive, and may overload cloud servers and communication networks. On the other hand, modern IoT devices are reasonably capable nowadays, although each of them is still much weaker compared to a cloud server. Nevertheless, using already-deployed IoT devices only for collecting sensor data largely underutilizes their capabilities. To support IoT analytics using deployed resources, we propose to: (i) perform some operations on IoT devices, (ii) divide larger IoT analytics into several *operators*, which are the basic deployable software components, and (iii) dynamically deploy operators to maintain Quality-of-Service (QoS) targets, such as sampling rates of sensors and accuracy of object recognizers. We organize resources from many IoT devices into a *fog computing* platform to realize our proposal.

This work was supported by the Ministry of Science and Technology of Taiwan under grants 106-2221-E-007-101 and 105-2221-E-007-088. The authors also appreciate generous support from Lite-On Technology Corporation.

Fog computing is proposed by Cisco [16] and generalized in several studies [21, 23, 33, 35], which concurrently leverages resources from cloud servers in data centers, application servers in edge networks, and embedded IoT devices and personal computers as end devices. We call these devices as *fog devices*, which are managed by a conceptually centralized server. We envision a fog computing ecosystem with three parties. The *fog service provider* offers a platform to *application developers*, who write and sell IoT analytics (and other) applications. *Fog users* are individuals requesting to deploy specific IoT analytics at diverse QoS levels. The fog service provider then deploys the IoT analytics accordingly. This ecosystem is quite similar to mobile application stores and cloud computing platforms, but offers much higher flexibility.

This paper makes the following contributions on developing and optimizing a fog computing platform to support IoT analytics:

- We formulate and solve a critical optimization problem for IoT analytics.
- We build an end-to-end platform and testbed with heterogeneous IoT devices, virtualized environments, enhanced controller, distributed analytics, and a range of multi-modal applications.
- We conduct trace-driven simulations for large-scale evaluations on efficiency and scalability.

More details on these contributions are given in the following paragraphs.

The crux of optimizing this platform is to intelligently deploy operators on fog devices to maximize the number of IoT analytics requests satisfying target QoS levels. This problem is referred to as the *operator deployment problem*. In particular, we design an *efficient operator deployment algorithm* to maximize the number of satisfied IoT analytics requests. The algorithm takes: (i) network bandwidth constraints, (ii) virtualization overhead, and (iii) required resource models into considerations. Our algorithm, tailored for IoT analytics, is more general than those in earlier work [17, 18, 19].

Furthermore, fog computing platforms require several enablers to manage IoT analytics requests from multiple users. As a proof-of-concept, we construct a *real fog computing platform for IoT analytics* using TensorFlow [12], Docker [3], and Kubernetes [8] to dynamically deploy virtualized multi-operator IoT analytics. TensorFlow [12] supports graph processing [31] and comes with analytics libraries. We adopt Ku-

2330-2186/17 $31.00 © 2017 IEEE
DOI 10.1109/CloudCom.2017.45

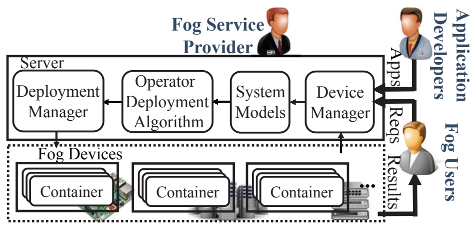


Fig. 1. The fog computing platform proposed for IoT analytics.

bernetes [8] to deploy, upgrade, monitor, and migrate operators in virtualized Docker [3] images. The resulting testbed allows us to evaluate our proposed solution, and may be used by other researchers, engineers, and hobbyists.

We implement three sample IoT analytics: (i) an air quality monitor, (ii) a sound classifier, and (iii) an object recognizer on our testbed. The testbed not only facilities experiments, but also provides data to drive larger-scale simulations. The experiment and simulation results are both very promising. For example, in simulations, our proposed algorithm outperforms a state-of-the-art algorithm [18] by up to (i) 89.4%, (ii) 168.3%, and (iii) 898.3% in terms of the number of satisfied requests, the number of active fog devices, and the consumed network resources, respectively.

II. System Overview

Fig. 1 gives an overview of our fog computing platform, which consists of the following major components.

- **Device manager.** We extend Kubernetes [8] to collect crucial device status, including: (i) the resource utilizations, e.g., CPU and RAM usage, (ii) the device locations, e.g., GPS readings, and (iii) sensor availability and capability, e.g., existence of depth cameras at VGA resolution.
- **System models.** The system models are derived through real experiments. The models predict the expected resource consumptions (e.g., CPU and network) under different QoS levels (e.g., sampling rate and recognition accuracy) and different fog devices.
- **Operator deployment algorithm.** The algorithm solves the core research problem of this paper. It makes decisions to deploy which operators on which fog deices for all IoT analytics requests. The requests come with diverse QoS targets and deployed locations. The goal of the algorithm is to maximize the number of satisfied requests, while considering: (i) the expected resource consumptions, (ii) the spatial requirements of requests, (iii) the required sensors, and (iv) the available resources of fog devices. A request is satisfied iff the target QoS is achieved at its specified location. Fig. 2 summarizes our operator deployment problem. Each operator graph (upper layer) represents a multi-operator analytics, and the device graph (lower layer) abstracts away the fog devices and network infrastructure.
- **Deployment manager.** We build a deployment manager on Kubernetes to remotely launch specific Docker images on chosen fog devices. The deployment manager also supports other management tasks, such as killing, restarting, and migrating containers.

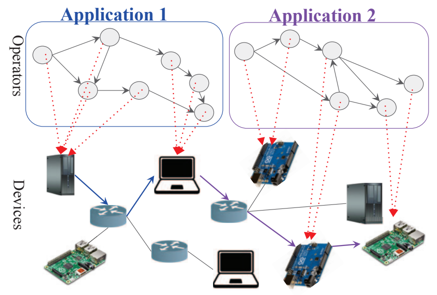


Fig. 2. The overview of our operator deployment problem.

III. Operator Deployment Problem: Formulation and Algorithm

The operator deployment problem is formulated and solved in this section.

A. Notations

Let $\mathbf{A}$ be a set of IoT analytics, $\mathbf{Q}$ be a set of requests, $\mathscr{A}_q$ be the IoT analytic of request q, and s_q be the QoS target of request q, for any $q \in \mathbf{Q}$. We let $\hat{G}_a = <\hat{\mathbf{V}}_a, \hat{\mathbf{E}}_a>$ be the operator graph of IoT analytic a, for all $a \in \mathbf{A}$, where $\hat{\mathbf{V}}_a$ is the operator set and $\hat{\mathbf{E}}_a$ is the edge set. $\dot{F}(\cdot)$ and $\bar{F}(\cdot)$ denote the system models, which capture the required resources of each operator and each edge, respectively. We concertize the system models using our testbed in Sec. IV. Moreover, we use $J_{q,i,i'}$ to represent the parent and children relationship between any two operators. More specifically, $J_{q,i,i'} = 1$ means operator i is the parent operator of i'. We let $G = <\mathbf{V}, \mathbf{E}>$ be the device graph, where $\mathbf{V}$ is a set of the fog devices and $\mathbf{E}$ is a set of the links. Each device k has its resource capacity $R_{k,u}$ of resource type u, where $k \in \mathbf{V}$ and $u \in \mathbf{U}$. $\mathbf{U}$ contains resource types, such as CPU, RAM, and network. Each physical link (k, k') has its capacity $B_{(k,k')}$. We use $T_{(k,k'),l}$ to describe whether link l is on the path from device k to device k'. Last, the operators have sensor and location requirements from the corresponding requests. We let $\mathbf{V}_{q,i}$ be an available device set that satisfies the sensor and location requirements of operator i in request q.The path between two fog devices can be computed by various routing algorithms, and the shortest paths are adopted if not otherwise specified.

B. Problem Formulation

Lemma 1 (Hardness)**.** *The operator deployment problem is NP-hard.*

Proof. (Sketch) The operator deployment problem can be reduced from the NP-hard Multiple Knapsack Problem (MKP). The MKP problem puts as many objects as possible into multiple knapsacks with diverse capacities. If we let $|\hat{\mathbf{V}}_a| = 1$, $|\mathbf{U}| = 1$, and $B_l = \infty$, we can map knapsacks as fog devices and objects as requests without any network constraint. Moreover, the value of each request is 1 and the weight is R_k. In this way, we reduce the MKP problem to our operator deployment problem in polynomial time. $\square$

Since the operator deployment problem is NP-hard, we formulate the problem as an ILP (Integer Linear Programming) problem:

$$\max \sum_{q \in \mathbf{Q}} p_q \tag{1a}$$

$$st: z_q = \sum_{i \in \hat{\mathbf{V}}_{\mathscr{A}_\mathbf{q}}} \sum_{k \in \mathbf{V}_{q,i}} x_{q,i,k} / |\hat{\mathbf{V}}_{\mathscr{A}_\mathbf{q}}| \ \forall q \in \mathbf{Q}; \tag{1b}$$

$$z_q - 1 < p_q \leq z_q \ \forall q \in \mathbf{Q}; \tag{1c}$$

$$\sum_{k \in \mathbf{V}_{q,i}} x_{q,i,k} \leq 1 \ \forall q \in \mathbf{Q}, i \in \hat{\mathbf{V}}_{\mathscr{A}_q}; \tag{1d}$$

$$\sum_{q \in \mathbf{Q}} \sum_{i \in \hat{\mathbf{V}}_{\mathscr{A}_q}} \dot{F}(\cdot) x_{q,i,k} \leq R_{k,u} \quad \forall k \in \mathbf{V}_i, u \in \mathbf{U}; \tag{1e}$$

$$\sum_{q \in \mathbf{Q}} \sum_{i \in \hat{\mathbf{V}}_{\mathscr{A}_q}} \sum_{i' \in \hat{\mathbf{V}}_{\mathscr{A}_q}} \sum_{k \in \mathbf{V}_{q,i}} \sum_{k' \in \mathbf{V}_{q,i'}} y_{q,(i,i'),(k,k')} J_{q,i,i'} T_{(k,k'),l} \bar{F}(\cdot) \leq B_l \ \forall l \in \mathbf{E}; \tag{1f}$$

$$x_{q,i,k} = \sum_{k' \in \mathbf{V}_{q,i'}} y_{q,(i,i'),(k,k')} \quad \forall q \in \mathbf{Q}, i \in \hat{\mathbf{V}}_{\mathscr{A}_q}, i' \in \hat{\mathbf{V}}_{\mathscr{A}_q}, k \in \mathbf{V}_{q,i}; \tag{1g}$$

$$x_{q,i',k'} = \sum_{k \in \mathbf{V}_{q,i}} y_{q,(i,i'),(k,k')} \quad \forall q \in \mathbf{Q}, i \in \hat{\mathbf{V}}_{\mathscr{A}_q}, i' \in \hat{\mathbf{V}}_{\mathscr{A}_q} k' \in \mathbf{V}_{q,i}; \tag{1h}$$

$$p_q, x_{q,i,k} \in \{0,1\} \ \forall q \in \mathbf{Q}, i \in \hat{\mathbf{V}}_{\mathscr{A}_\mathbf{q}}, k \in \mathbf{V}_{q,i}. \tag{1i}$$

In this formulation, $x_{q,i,k}$ is the decision variable, where $x_{q,i,k} = 1$ iff operator i of request q is deployed on device k. The objective function in Eq. (1a) counts the number of satisfied requests, where an intermediate variable $p_q = 1$ iff request q is satisfied. Eqs. (1b) and (1c) make sure that when request q is satisfied, all the operators of request q are deployed. Eq. (1d) makes sure that each operator is deployed only once. Eq. (1e) ensures that the required resources of the deployed operators do not exceed the resource capacities of the devices. Eq. (1f) makes sure that the required bandwidth of the deployed operators does not exceed the capacity of the network links. Eqs. (1g) and (1h) define another intermediate variable $y_{q,(i,i'),(k,k')}$, which is 1 iff edge (i,i') of request q is mapped to the physical path (k,k'). The intermediate variables $y_{q,(i,i'),(k,k')}$ are used to embed the operator graphs in the device graph.

C. Proposed Algorithm and Analysis

Because our problem is NP-hard, we propose a greedy operator deployment algorithm based on three intuitions (steps):

- **Scarcest resource first.** Because requests demand for different resource types at the same time, the scarcest resource becomes the limiting factor. To satisfy as many requests as possible, we identify the scarcest resource (after normalization), sort requests in the ascending order on the scarcest resource, and then consider the requests consuming less scarcest resource earlier. The result of this step is a sorted set of undeployed requests.
- **Shortest path first.** For each request, we consider all pairs of feasible fog devices for the source and destination operators. Because shorter paths generally lead to lower overall network loads, shorter end-to-end delays, and less processing/forwarding overhead, we select the pair of fog devices with the shortest path for the source and destination operators. The result of this step is the deployment decisions of the source and destination operators. We note that although this intuition is helpful to satisfy more requests, but the shortest path may not result in the optimal solution.
- **Early feature extraction.** Upon the source and destination operators are deployed, we start to deploy the remaining operators. Since the operators in typical IoT analytics extract increasingly higher-level features that are generally more compact, deploying the operators closer to the parent operator reduces the network workload and the transfer time. We employ a system parameter H and only consider deploying a child operator on fog devices within

Inputs: Request info, such as QoS target s_q, analytics info, such as operator graphs $\hat{G}_a$, device info, such as device resource capacity $R_{k,u}$, and link capacity B_l.
Output: The deployment decision $x_{q,i,k}$.

```
1:  let $\hat{C}_q$ be the scarcest resource $u'$ required by $q$
2:  sort $\mathbf{Q}$ by $\hat{C}_q$ in asc. order
3:  for $q \in \mathbf{Q}$ do
4:      let $\mathbf{M}$ to store pairs of devices, including source device $\hat{k}$
        and destination device $\check{k}$, which are feasible to launch source $\hat{i}$
        and destination operators $\check{i}$ of request $q$, respectively
5:      sort $\mathbf{M}$ on the shortest path length from $\hat{k}$ to $\check{k}$ in asc. order
6:      for $(\hat{k}, \check{k}) \in \mathbf{M}$ do
7:          let $x_{q,\hat{i},\hat{k}} = 1$
8:          let $x_{q,\check{i},\check{k}} = 1$
9:          let $\mathbf{W}$ be all the deployed operators
10:         add $\hat{i}$ and $\check{i}$ to $\mathbf{W}$
11:         while $\mathbf{W}$ is not empty do
12:             pop an operator $i'$ from $\mathbf{W}$
13:             let $\mathbf{N}_i'$ be the child operators of $i'$
14:             let $H$ be hop limit
15:             let $\mathbf{D}_h$ be devices that are $h$ hops to the device
        running operator $i'$
16:             sort $\mathbf{D}_h$ on the number of hops to $\check{k}$
17:             for $h = 0, 1, 2, \ldots, H$ do
18:                 for $i \in \mathbf{N}_{i'}$ do
19:                     for $k \in \mathbf{D}_h$ do
20:                         if Eqs. (1d) to (1f) are satisfied then
21:                             let $x_{q,i,k} = 1$
22:                             add $i$ to $\mathbf{W}$
23:         if operators of request $q$ are not all deployed then
24:             clear all deployed operator of request $q$
```

Fig. 3. The pseudocode of our proposed SSE algorithm.

H hops from its parent operator. H is typically a small constant, which allows us to reduce our time complexity. The result of this step is the deployment decisions of operators other than the source and destination ones.

We refer to our proposed algorithm as *SSE* algorithm, after the initials of the three intuitions. Fig. 3 gives its pseudocode. Lines 1–2 implement the first intuition, sort the requests in $\mathbf{Q}$ by the scarcest resource type. Lines 4–13 realize the second intuition, where we store source and destination pairs of devices into $\mathbf{M}$ and sort them by their path lengths. Lines 14–19 implement the third intuition to consider the fog devices that are closer to source device first. Lines 20–21 check the constraints and make the final decisions.

IV. Testbed and Experiments

In this section, we realize a testbed of our fog computing platform for: (i) conducting a measurement study to derive our system models and (ii) evaluating the derived models and the performance of our fog computing platform.

A. Testbed Implementations

Compared to our earlier SCALE projects [15, 34], which focus on safe communities, our testbed built for IoT analytics consisting of : (i) an Intel i5 workstation running Ubuntu, Docker [3], and Kubernetes [8] as the server and (ii) 5 Intel PCs (1.8 GHz 8-core i7 CPUs) and 5 Raspberry Pi 3 (1.2 GHz 4-core ARM CPUs) as the fog devices running Ubuntu and HypriotOS, respectively. We implement the software components presented in Fig. 1 on the server and fog devices. The server and devices are connected with an Ethernet switch in a private network. To emulate WiFi-based IoT networks, we adopt a traffic shaper [13] to throttle the bandwidth at 8 Mbps [20]. To avoid overloading the fog devices, we reserve 20% of resources for operations other than IoT analytics, e.g., OS scheduling and resource monitoring.

We implement the multi-operator IoT analytics using TensorFlow [12], in which every application is represented as a graph consisting of *nodes* and *links*. The nodes can be simple tasks, such as additions and subtractions, or very complex machine learning tasks. The links are the data flows among nodes. We use TensorFlow *tags* to split every graph into multiple smaller subgraphs, where each subgraph contains one or multiple nodes and associated links. These subgraphs are essentially the *operators* in our fog computing platform, which are the units of deployments.

However, IoT analytics may require some features that are not available in TensorFlow. For example, OpenCV [11] is needed for processing images and librosa [9] is needed for analyzing audios. These additional libraries are quite large and take hours to compile. For example, compiling OpenCV on a Raspberry Pi takes about 3.5 hours. Therefore, installing these libraries on-the-fly is infeasible. On the other hand, reimplementing the features in TensorFlow incurs high engineering overhead, which may not be worth it. Fortunately, our fog computing platform achieves short deployment time using Docker images. Furthermore, TensorFlow does not natively support stream processing, and we add a queue between any two adjacent operators to increase the overall performance.

We develop three sample multi-operator IoT analytics to show the practicality of our platform. Each of them contains a large number of nodes, e.g., the object recognizer has 348 nodes. We split each TensorFlow graph into a few operators (subgraphs) as follows.

- **The air quality monitor** receives four gas sensor readings in JSON format through the MQTT protocol [10]. It then parses the JSON objects and calculates the moving averages (potentially, other statistics too) of individual gas sensors. The results can be used to detect emergency events, such as explosive gas leaking. We split the IoT analytic into 5 operators. The first operator subscribes to all gas sensor data via MQTT. The other four operators are responsible for calculating the moving averages of the four gas sensors, respectively.
- **The sound classifier** detects different types of sound for community safety. For example, this IoT analytic may detect a gun shot and automatically notify the police. It adopts a pre-trained four-layer neural network for analyzing the audio inputs. The IoT analytic is split into 5 linear operators. The first operator gathers the audio data, while the other four operators execute the four layers of the neural network, respectively.
- **The object recognizer** collects and analyzes camera images for recognizing objects in them. It employs a pre-trained neural network with more than 10 layers. We split the IoT analytic into 3 operators for input, hidden, and output layers, respectively. The first operator also captures and resizes the camera images using OpenCV [11].

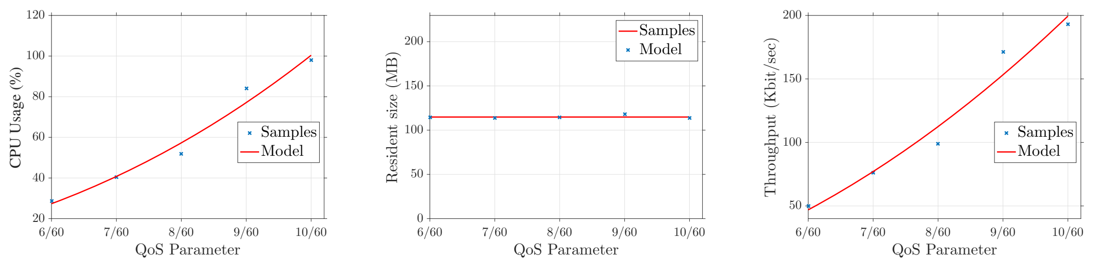
B. System Model Derivation

The system models map the target QoS levels of different IoT analytics to the required resources on heterogeneous hardware specifications. The parameters of operator model $\dot{F}(\mathscr{A}_q, s_q, i, u, k)$ and edge model $\bar{F}(\mathscr{A}_q, s_q, (i, i'))$ are derived as follows. First, we conduct an offline measurement study to derive the system models of Raspberry Pis (the lowest-end devices). For Intel PCs (more powerful devices without existing models), we use the models of less powerful devices for *bootstrapping*. We then update the system models online to iteratively derive the customized system models. Using system models of less powerful devices for bootstrapping prevents us from overloading the fog devices. Although we only have two types of fog devices in the testbed, the same procedure is applicable to more heterogeneous devices.

We model the following three resources:

- **CPU load:** the physical CPU load in percentage reported by Docker stats API.
- **RAM usage:** the total physical memory usage reported by Docker stats API.
- **Network throughput:** total traffic amount (in both directions) reported by Tcpdump.

We adopt different invocation frequencies as the QoS parameters; other QoS parameters may also be adopted if needed.



(a) (b) (c)

Fig. 4. The system models of (a) CPU load, (b) RAM usage, and (c) network throughput. Sample results from an operator and an edge of the object recognizer.

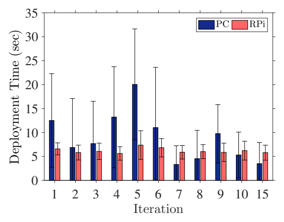
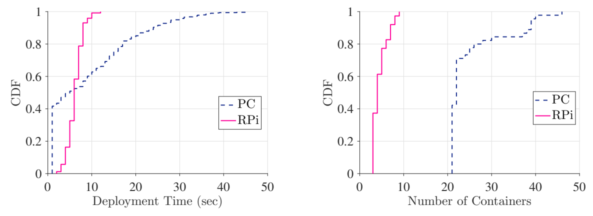


Fig. 5. The average deployment time of operators in each iteration.



(a) (b)

Fig. 6. (a) Most operators deployed on Intel PCs take less time than on Raspberry Pis and (b) the number of containers created on Intel PCs are far more than on Raspberry Pis.

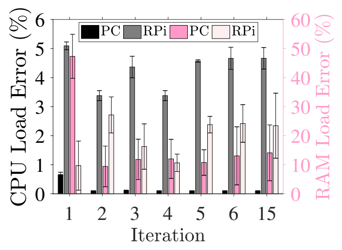


Fig. 7. Online updates of our system models are effective. Iterations 7–14 are omitted due to the space limitations.

Particularly, in our experiments, we choose the following sample parameters: (i) $\{0.25, 0.5, 1, 2, 4\}$ Hz in the air quality monitor, (ii) $\{5/60, 6/60, 7/60, 8/60, 9/60\}$ Hz in the sound classifier, and (iii) $\{6/60, 7/60, 8/60, 9/60, 10/60\}$ Hz in the object recognizer. Each QoS level of every IoT analytic is executed 5 times with 60-sec duration. We report the average results, after filtering out the results that deviate from the average for more than 1.7 times of the standard deviation.

TABLE I
THE MEAN (STANDARD DEVIATION) R-SQUARE VALUES AND P-VALUES AMONG ALL OPERATORS AND EDGES.

IoT Analytics	**CPU Load**		**Network Throughput**	
	R-Square	**P-Value**	**R-Square**	**P-Value**
Air Quality Monitor	0.99 (0.000023)	<0.001	0.99 (0.000001)	<0.001
Sound Classifier	0.85 (0.023263)	<0.010	0.84 (0.069494)	<0.010
Object Recognizer	0.99 (0.012540)	<0.010	0.97 (0.004982)	<0.010

Fig. 4 shows the measured results and system models of a sample operator (CPU and RAM) and a sample edge (network) from the object recognizer. We make a few observations on the sample results and adopt the following regression models. First, we find that the RAM usage (Fig. 4(b)) remains the same under different QoS parameters. Therefore, we model it as a constant value, which gives us an average error of 0.4% and a maximum error of 1%. For the CPU load (Fig. 4(a)) and network throughput (Fig. 4(c)), we adopt power models (after considering several common functions), which give 0.99 R-square scores in both cases. The operator and edge models are summarized as:

$$\dot{F}(\mathscr{A}_q, s_q, i, u, k) = \begin{cases} d_{\mathscr{A}_q, s_q, i, k} & u = \text{RAM}; \\ \dot{a}_{\mathscr{A}_q, i, k} s_q^{\dot{b}_{\mathscr{A}_q, i, k}} + \dot{c}_{\mathscr{A}_q, i, k} & u = \text{CPU}; \end{cases} \quad (2)$$

$$\bar{F}(\mathscr{A}_q, s_q, (i, i')) = \bar{a}_{\mathscr{A}_q, (i, i')} s_q^{\bar{b}_{\mathscr{A}_q, (i, i')}} + \bar{c}_{\mathscr{A}_q, (i, i')}, \quad (3)$$

where $\dot{a}, \dot{b}, \dot{c}, d, \bar{a}, \bar{b}, \bar{c}$ are model parameters.

Similar observations can be made on other operators and edges, and model parameters can be derived by regression. We report the mean R-square scores of the CPU and network models with their standard deviations in Table I. This table shows that our system models are fairly accurate. Moreover, the same table also reports p-values, which are < 0.01, showing the regression is statistically significant.

C. Validation

We run the following experiments for 15 iterations, where the offline system models are initially used for both Raspberry Pis and Intel PCs to drive our operator deployment algorithm. Based on the actual resource consumptions, we update our system models after each iteration that lasts for about 3 minutes. In each iteration, we randomly generate requests of three sample IoT analytics with random QoS levels chosen from the sample parameters of each IoT analytic. We up-sample the QoS levels by 10% as a small buffer for workload surges. Specifically, we generate a large number of requests, and execute our operator deployment algorithm to deploy as many requests as possible. Although network conditions affect the download time of Docker images, disseminating Docker images efficiently is out of the scope of this paper. Therefore, we assume all the Docker images are already cached on fog devices. Sample experiment results with 95% confidence intervals are given below.

Operators are deployed almost instantly in our platform. Fig. 5 reports the average deployment time of operators in each iteration. The figure reveals that the operators take less than 20 seconds on average to be deployed, which is sufficient for most IoT analytics. IoT analytics that need to have even shorter response time can be *pre-deployed*. Fig. 6 gives the detailed Cumulative Distribution Function (CDF) curves. As Fig. 6(a) shows, over 50% of operators deployed on Intel PCs take less time than on Raspberry Pis because Intel PCs have higher computational power. The rest of deployments take more time on Intel PCs than on Raspberry Pis because there are considerably more operators running on PC than Raspberry Pis, as reported in Fig. 6(b). When a large number of containers are simultaneously created on the same device, the I/O overhead is significantly increased.

Our system models are reasonably accurate and support heterogeneous fog devices. We report the system model errors of two sample models: (i) power models of CPU loads and (ii) constant models of RAM usage in Fig. 7. This figure shows that our system models become increasingly more accurate over iterations. More specifically, the CPU load errors are no worse than 5% and the RAM usage errors are no worse than 25% on average after 5 iterations.

Our operator deployment algorithm satisfies most of the requested QoS levels. Table II shows the *QoS satisfaction rate* and *the number of rejected requests* under different numbers of requests. The QoS satisfaction rate is the fraction of deployed requests that meet the QoS targets, and the rejected requests are not deployed by our algorithm due to lack of resources. The table leads to three observations: (i) the QoS satisfaction rate is getting higher over iterations. At the fifth iteration, the QoS satisfaction rate reaches to 100%, (ii) more received requests require more iterations to satisfy all the requests, and (iii) once our platform is fully loaded, some requests are rejected to maintain the QoS levels of deployed requests.

Our experiments demonstrate the practicality and efficiency of our models, algorithm, and platform. For larger scale evaluations, we perform detailed simulations in the next section.

TABLE II
QoS SATISFACTION RATE AND NUMBER OF REJECTED REQUESTS

	Iter. 1		Iter. 2		Iter. 3		Iter. 4		Iter. 5	
Recv. Req.	**Rej. Req.**	**Satis. Rate**	**Rej. Req.**	**Satis. Rate**	**Rej. Req.**	**Satis. Rate**	**Rej. Req.**	**Satis. Rate**	**Rej. Req.**	**Satis. Rate**
4	0	100%	0	100%	0	100%	0	100%	0	100%
8	0	100%	0	100%	0	100%	0	100%	0	100%
16	0	79%	0	100%	0	100%	0	100%	0	100%
32	0	89%	10	95%	10	100%	10	100%	10	100%
64	22	71%	42	95%	42	95%	42	100%	42	100%

V. SIMULATIONS

In this section, we exercise a wider spectrum of system parameters of a fog computing platform.

A. Setup

We build a detailed simulator in Python. In this simulator, we implement our SSE and three baseline algorithms, which are Random, Linear, and Optimal Data Stream Processing Placement (ODP) [18]. The Random algorithm mimics a fog computing platform without a central controller, and deploys operators on random devices. The Linear algorithm greedily deploys operators from the source fog devices to adjacent devices, while taking the resource, sensor, and location constraints into considerations. The ODP algorithm is a state-of-the-art operator deployment algorithm with a flexible objective function. We adapt its objective function to maximize the number of satisfied requests for fair comparisons. The ODP algorithm employs a commercial ILP solver: CPLEX [6], which may take prohibitively long time to terminate. Therefore, it is recommended to set a 500-sec cap on the execution time [18]. We run simulations with the four algorithms for comparisons.

BRITE [2] is used to generate the network topology among the fog devices and the server. The device coordinates generated by BRITE are fit to the geographical area of Taipei city, Taiwan. Each fog device has a random CPU capacity between 100% and 800% (corresponding to the number of cores) and a random RAM capacity between 1 and 16 GB. Each fog device comes with a random wireless network interface with the following link capacity: 45 kbps with LoRa [26], 8 Mbps with WiFi [20], and 25 Mbps with 4G [5].

Poisson processes with 1-min average arrival time and 10-min average departure time are used for generating requests. In this way, the number of active requests steadily increases over time. Each request randomly picks one of the three IoT analytics with random QoS targets between 0.1 and 4 Hz. The system models (of PCs and Raspberry Pis) derived in our experiments are used in the simulations. Each request randomly occurs in one of the 12 districts of Taipei city. We consider random sensors, including cameras, microphones, (4 types of) gas sensors, and 4G cellular dongles. IoT analytics are associated with the corresponding sensors.

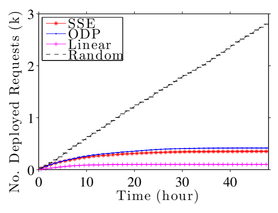
We consider the following performance metrics.

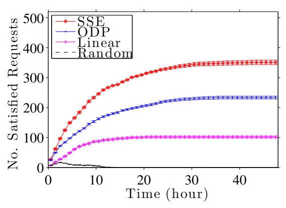
- **Number of deployed requests.**
- **Number of satisfied requests** that comply with the constraints in Eqs. (1d) and (1f).
- **Number of overloaded links.**
- **Number of active devices** that host one or more operators.
- **Resource loads** that are in terms of CPU, RAM, and network usages.
- **Running time** of the operator deployment algorithms.

We run 48-hour simulations on an AMD 64-core workstation. Our operator deployment algorithm is invoked once every 30 mins. We consider the number of fog devices $|\mathbf{V}| \in \{10, 25, 50, 75, 100\}$. We let $|\mathbf{V}| = 100$ and the maximal number of hop $H = 10$ if not otherwise specified. We repeat each simulation 5 times and report means with 95% confidence intervals if applicable.

B. Results

Our proposed SSE algorithm results in QoS improvements. Fig. 8 reports the overall QoS achieved by different algorithms. Fig. 8(a) shows that Random and ODP deploy up to 11.53 times and 18.9% more requests compared to our SSE algorithm. Moreover, our SSE algorithm deploys up to


(a)


(b)

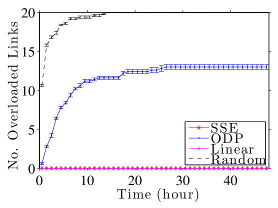

(c)

Fig. 8. QoS improvement of our SSE algorithm, in: (a) the number of deployed requests, (b) the number of satisfied requests, and (c) the number of overloaded links.

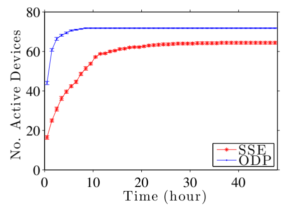
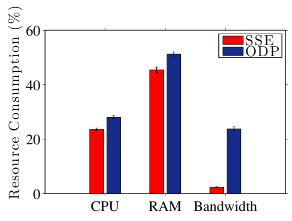
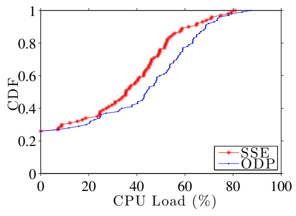
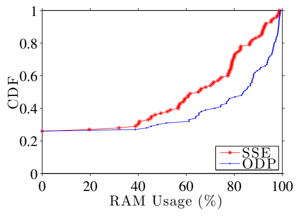


Fig. 9. The number of active fog devices.


(a)


(b)


(c)

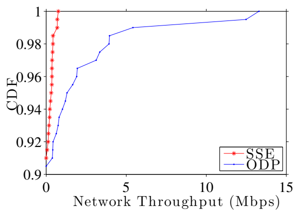

(d)

Fig. 10. Resource efficiency improvement of our SSE algorithm: (a) peak overall resource consumption, (b) CPU load, (c) RAM usage, and (d) network throughput distributions of fog devices.

1.7 times more requests than Linear. In Fig. 8(b), we observe that our SSE algorithm satisfies up to 171% and 89.4% more requests than Linear and ODP, respectively. On the other hand, Random satisfies zero request after running for 14 hours. The inferior performance of Random and ODP is because they do not capture the detailed resource constraints, and thus are too aggressive and overload the fog computing platform. In contrast, Linear algorithm limits its operator deployment decisions to adjacent devices, and thus is too conservative. To demonstrate this, we plot the number of overloaded links in Fig. 8(c), where Random and ODP result in as many as 13 and 21 overloaded links, while Linear (and our SSE algorithm) leads to none. Since Random and Linear suffer from significantly fewer number of satisfied requests, we no longer consider them in the rest of this paper.

Our proposed SSE algorithm reduces energy cost and carbon footprint. Fig. 9 reports the number of the active devices over time. It shows that, compared to ODP, our SSE algorithm turns off up to 168.3% more devices, yet satisfying more requests as shown in Fig. 8(b). Putting more (inactive) fog devices into asleep reduces energy cost and carbon footprint.

Our proposed SSE algorithm is resource efficient. Fig. 10 shows the peak overall and distributions of resource consumptions. In particular, Fig. 10(a) reports the peak total resource consumptions, which shows that our SSE algorithm reduces 18.4%, 12.7%, and 898.3% resource consumptions in terms of CPU, RAM, and network. Figs. 10(b)–10(d) plot CDF curves of the resource consumptions from individual fog devices. These figures clearly reveal that our SSE algorithm results in much better resource efficiency, as it consumes fewer resources and satisfies more requests. It is because ODP tries to deploy more requests, which may not be satisfied, as illustrated in Figs. 8(a) and 8(b).

Our proposed SSE algorithm is scalable. Table III reports the maximal running time of the operator deployment algorithms throughout 48-hour simulations under different number of fog devices. This table shows that the running time of ODP exceeds its 500-sec threshold with merely 100+ fog devices. Moreover, ODP's running time grows exponentially. In contrast, our SSE algorithm only takes 12.22 sec to make the deployment decisions with 100 fog devices, and its running grows linearly.

TABLE III
MAXIMAL RUNNING TIME (S) OF OPERATOR DEPLOYMENT ALGORITHMS

No. Devices	10	25	50	75	100
SSE	0.41	1.25	6.11	10.27	12.22
ODP	3.87	24.66	195.93	300.83	>500

VI. Related Work

Fog computing has been used in IoT platforms [21, 33, 35]. However, these studies do not dynamically deploy IoT analytics to the heterogeneous fog devices. Several studies [14, 27, 28, 30] apply virtualization technologies on fog computing platforms for dynamic and transparent deployment, remote management, and application isolation. However, they do not rigorously solve the operator deployment problem. Saurez et al. [32] split applications for distributed deployment, but the decisions are statically determined by developers. In contrast, our platform is more flexible when deploying operators.

Graph processing, such as Pregel [25], regards an application as a graph with nodes and links. Khayyat et al. [24] extend Pregel for dynamic graph partitioning. This graph partitioning problem is essentially the same as the operator deployment problem in the literature of *streaming processing* [22]. Salihoglu et al. [31] enhance Pregel to reduce the communication traffic. However, these studies do not consider IoT analytics and virtualization, nor solve the operator deployment problem.

The operator deployment problem has been solved under rather strong assumptions. For instance, Eidenbenz and Locher [19] propose a tree-based approximation algorithm for well-connected servers in data centers. In the literature, similar problems have been solved in more distributed environments. For example, Pietzuch et al. [29] adapt to network latency and data rates, when computing the deployment decisions. Cardellini et al. [17, 18] consider the constraints on both nodes and links. They formulate an ILP problem and solve it with a commercial solver, which may be time-consuming. Compared to our work, these studies [17, 18, 19, 29] don't conduct detailed measurements nor model resource consumptions. Furthermore, the operators considered in their papers are primitive, compared to the ones in IoT analytics.

VII. Conclusion

In this paper, we study the operator deployment problem to maximize the number of satisfied IoT analytics requests with diverse target QoS levels in a heterogeneous fog computing platform. This platform features: (i) graph processing as the programming model, (ii) lightweight virtualization for resource provisioning, and (iii) a novel operator deployment algorithm for deployment decisions. Our algorithm carefully captures the resource constraints, such as CPU, RAM, and network, which are modeled by our proposed system models. We design, develop, and implement a testbed to quantify: (i) the error rates of our system models, which are no more than 5% in terms of CPU load, (ii) QoS satisfaction rate, which reaches 100% after at most 5 iterations, and (iii) operator deployment time, which is less than 20 secs on average. For larger-scale evaluations, we conduct extensive simulations using traces collected from the testbed. The simulation results show that Random and Linear algorithms suffer from unacceptable number of satisfied requests. Moreover, our algorithm outperforms the state-of-the-art ODP algorithm by up to 89.4%, 168.3%, and 898.3% in terms of the number of the satisfied requests, the number of active fog devices, and the consumed network bandwidth, respectively. Last, our algorithm runs fast (12 secs for 100 fog devices) and scales well (linear growth rate).

This paper can be extended in several directions. For example, Software-Defined Network (SDN) may be integrated for network QoS guarantees. Moreover, intelligent docker migrations may be employed to adapt to system dynamics. Last, predictions on available resources may be leveraged for non-dedicated fog devices.

References

[1] Amazon Echo. http://tinyurl.com/j3calfj.

[2] BRITE. https://www.cs.bu.edu/brite/.

[3] Docker. https://www.docker.com.

[4] Google Home. https://madeby.google.com/home/.

[5] How fast is 4G? http://www.4g.co.uk/how-fast-is-4g/.

[6] IBM CPLEX optimizer. http://tinyurl.com/mnh8mlp.

[7] Internet of Things (IoT): number of connected devices worldwide from 2012 to 2020 (in billions). https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/.

[8] Kubernetes. http://kubernetes.io/.

[9] librosa. https://github.com/librosa/librosa.

[10] MQTT. http://mqtt.org.

[11] Opencv. http://opencv.org.

[12] Tensorflow. https://www.tensorflow.org.

[13] Wonder Shaper. http://lartc.org/wondershaper/.

[14] P. Bellavista and A. Zanni. Feasibility of fog computing deployment based on docker containerization over RaspberryPi. In *Proc. of ACM Conference on Distributed Computing and Networking (ICDCN)*, Hyderabad, India, January 2017.

[15] K. Benson, C. Fracchia, G. Wang, Q. Zhu, S. Almomen, J. Cohn, L. Darcy, D. Hoffman, M. Makai, J. Stamatakis, and N. Venkatasubramanian. SCALE: Safe community awareness and alerting leveraging the internet of things. *IEEE Communications Magazine*, 53(12):27–34, 2015.

[16] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. Fog computing and its role in the internet of things. In *Proc. of ACM SIGCOMM workshop on Mobile Cloud Computing (MCC)*, Helsinki, Finland, August 2012.

[17] V. Cardellini, V. Grassi, F. Lo, and M. Nardelli. Distributed QoS-aware scheduling in storm. In *Proc. of ACM International Conference on Distributed Event-Based Systems (DEBS)*, Oslo, Norway, June 2015.

[18] V. Cardellini, V. Grassi, F. Presti, and M. Nardelli. Optimal operator placement for distributed stream processing applications. In *Proc. of ACM International Conference on Distributed and Event-based Systems (DEBS)*, Irvine, CA, June 2016.

[19] R. Eidenbenz and T. Locher. Task allocation for distributed stream processing. In *Proc. of IEEE INFOCOM*, San Francisco, CA, April 2016.

[20] R. Friedman, A. Kogan, and Y. Krivolapov. On power and throughput tradeoffs of WiFi and Bluetooth in smartphones. *IEEE Transactions on Mobile Computing*, 12(2):1363–1375, 2013.

[21] K. Giang, M. Blackstock, R. Lea, and C. Leung. Developing IoT applications in the fog: A distributed dataflow approach. In *Proc. of IEEE International Conference on Internet of Things (IoT)*, Seoul, South Korea, December 2015.

[22] M. Hirzel, R. Soule, S. Schneider, B. Gedik, and R. Grimm. A catalog of stream processing optimizations. *ACM transactions on Computing Surveys*, 46(4), 2014.

[23] H. Hong, J. Chuang, and C. Hsu. Animation rendering on multimedia fog computing platforms. In *Proc. of IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, Luxembourg, December 2016.

[24] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis. Mizan: a system for dynamic load balancing in large-scale graph processing. In *Proc. of ACM European Conference on Computer Systems (EuroSys)*, Prague, Czech Republic, April 2013.

[25] G. Malewicz, H. Austern, J. Bik, C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proc. of ACM SIGMOD*, Indianapolis, IN, June 2010.

[26] K. Mikhaylov, J. Petaejaejaervi, and T. Haenninen. Analysis of capacity and scalability of the LoRa low power wide area network technology. In *Proc. of European Wireless Conference (EW)*, Oulu, Finland, May 2016.

[27] C. Pahl, S. Helmer, L. Miori, J. Sanin, and B. Lee. A container-based edge cloud paas architecture based on Raspberry Pi clusters. In *Proc. of IEEE International Conference on Future Internet of Things and Cloud Workshops (FiCloudW)*, Vienna, Austria, August 2016.

[28] C. Pahl and B. Lee. Containers and clusters for edge cloud architectures–a technology review. In *Proc. of IEEE International Conference on Future Internet of Things and Cloud Workshops (FiCloudW)*, Rome, Italy, August 2015.

[29] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-aware operator placement for stream-processing systems. In *Proc. of IEEE International Conference on Data Engineering (ICDE)*, Atlanta, GA, April 2006.

[30] D. Pizzolli, G. Cossu, D. Santoro, L. Capra, C. Dupont, D. Charalampos, F. Pellegrini, F. Antonelli, and S. Cretti. Cloud4IoT: a heterogeneous, distributed and autonomic cloud platform for the IoT. In *Proc. of IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, Luxembourg, December 2016.

[31] S. Salihoglu and W. Jennifer. GPS: A graph processing system. In *Proc. of ACM International Conference on Scientific and Statistical Database Management (SSDBM)*, Baltimore, MD, July 2013.

[32] E. Saurez, K. Hong, D. Lillethun, U. Ramachandran, and B. Ottenwalder. Incremental deployment and migration of geo-distributed situation awareness applications in the fog. In *Proc. of ACM International Conference on Distributed and Event-based Systems (DEBS)*, Irvine, CA, June 2016.

[33] S. Shin, S. Seo, S. Eom, J. Jung, and H. Lee. A Pub/Sub-Based fog computing architecture for Internet-of-Vehicles. In *Proc. of IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, Luxembourg, December 2016.

[34] M. Uddin, A. Nelson, K. Benson, G. Wang, Q. Zhu, Q. Han, N. Alhassoun, P. Chakravarthi, J. Stamatakis, D. Hoffman, L. Darcy, and N. Venkatasubramanian. The scale2 multi-network architecture for iot-based resilient communities. In *Proc. of IEEE International Conference on Smart Computing (SMARTCOMP)*, St. Louis, MO, USA, May 2016.

[35] D. Wu, I. Arkhipov, M. Kim, L. Talcott, C. Regan, A. McCann, and N. Venkatasubramanian. ADDSEN: adaptive data processing and dissemination for drone swarms in urban sensing. *IEEE Transactions on Computers*, 66(2):183–198, 2016.