Dynamic Module Deployment in a Fog Computing Platform

Hua-Jun Hong, Pei-Hsuan Tsai, and Cheng-Hsin Hsu
Department of Computer Science, National Tsing Hua University, Taiwan

***Abstract*—Several applications, such as smart cities, smart homes and smart hospitals adopt Internet of Things (IoT) networks to collect data from IoT devices. The incredible growing speed of the number of IoT devices congests the networks and the large amount of data, which are streamed to data centers for further analysis, overload the data centers. In this paper, we implement a fog computing platform that leverages end devices, edge networks, and data centers to serve the IoT applications. In this paper, we focus on implementing a fog computing platform, which dynamically pushes programs to the devices. ' The programs pushed to the devices pre-process the data before transmitting them over the Internet, which reduces the network traffic and the load of data centers. We survey the existing platforms and virtualization technologies, and leverage them to implement the fog computing platform. Moreover, we formulate a deployment problem of the programs. We propose an efficient heuristic deployment algorithm to solve the problem. We also implement an optimal algorithm for comparisons. We conduct experiments with a real testbed to evaluate our algorithms and fog computing platform. The proposed algorithm shows near-optimal performance, which only deviates from optimal algorithm by at most 2% in terms of satisfied requests. Moreover, the proposed algorithm runs in real-time, and is scalable. More precisely, it computes 1000 requests with 500 devices in < 2 seconds. Last, the implemented fog computing platform results in real-time deployment speed: it deploys 20 requests < 10 seconds.**

I. Introduction

The Internet of Things (IoT) is getting popular all over the world. There are 6.4 billion IoT devices deployed in 2016, which is 30% more compared to 2015, and the number is expected to increase to 20.8 billion in 2020 [1]. Currently, the devices are used to sense/collect data and send the data to powerful servers, such as cloud servers through the Internet for processing and further analysis. After that, the analyzed results are sent to IoT users for various IoT applications, such as smart cities, smart homes, and smart hospitals.

The incredible growing speed of the number of IoT devices leads to severe network congestion and surging server loads under the huge amount of incoming data streams generated by many IoT devices. To solve these problems, we may try to pre-process the data using the IoT devices to extract some *higher-level features* for reducing the data size. We then send the reduced data to the server for analysis. Because the size of data is reduced by the IoT devices before being transmitted over the Internet, the network traffic amount is reduced. Moreover, the IoT devices pre-process the data before sending them to the server, so that the workload of the server is also reduced.

We envision a dynamic IoT platform, which allow us to dynamically and efficiently change the programs/algorithms installed in the IoT devices. We propose to adopt the concept of *fog computing* to implement it. The fog computing concept is proposed for IoT [2] and generalized by Vaquero et al. [3]. Fog computing leverages *fog devices* in data centers, edge networks, and end devices simultaneously, as illustrated in Fig. 1. The fog devices are managed by a centralized server, which receives requests from users and decides how to serve the requests on the fog devices. The heterogeneous fog devices help us to pre-process the data on the resource-limited devices, including the edge networks and end devices. Moreover, the fog computing platform extends cloud to end devices, which are closer to end users, and result in lower latency. Shen et al. [4] implement a simple fog computing platform to demonstrate the benefits of latency compared to cloud. They also compare the processing time of heterogeneous fog devices. For brevity, we call fog devices as devices in the rest of this paper.

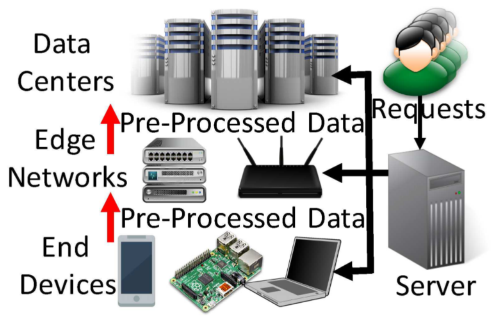


Fig. 1. Overview of our considered fog computing platform.

Implementing the fog computing platform that supports heterogeneous devices, in which the actual configurations of the devices and inter-connected networks need to be carefully optimized, resulting in several challenges. First, because of the limited resources of the devices, we have to split requests from users into smaller *modules* running on the devices. The modules collaborate among one another to fulfill the requests. Making decisions on decompositions of requests is hard, because the large secured space and dynamic nature of the platform. Second, we have to connect the modules, which are deployed on different devices. Building the flows among modules is a difficult task, as the networks are heterogeneous and dynamic. Third, different requests can be split into different number and types of modules, which require diverse resources. Deploying these modules on the resource limited devices must be carefully planned to maintain quality-of-service. The current paper focuses on solving the module deployment problem and implements a unified fog computing platform, which can dynamically deploy modules on the devices.

The proposed fog computing platform could be implemented based on several open-source platforms, such as Kubernetes [5], OpenStack [6], and SaltStack [7]. Moreover, virtualization technology is important to dynamic deployments because virtualized modules are easier to be dynamically placed on the devices or migrate among the devices for optimization. There are two kinds of virtualization technologies: the traditional virtual machine and container, which is a light-weight virtual machine. The traditional virtual machine technology, such as VMWare gives better isolation, while the container achieves lower set-up delay. Choosing better open-source platform and virtualization technology is also crucial to build our fog computing platform.

In the rest of this paper, we survey the related work in Sec. II. We present our design of our fog computing platform in Sec. III. We focus on the problem of deployment, and we carefully formulate the module deployment problem and design an efficient solution in Sec. IV. We then implement the proposed fog computing platform based on the existing open-source platforms and state-of-the-art virtualization technologies in Sec. V. In the same section, we evaluate the proposed fog computing platform and algorithms. The experiment results show that our proposed platform and algorithm achieve (i) near-optimality, (ii) real-timeness, (iii) quick deployment, and (iv) high scalability,

II. Related Work

Several studies [2], [3], [4] present the concepts/definitions of fog computing. These studies also discuss the challenges, potential applications, and benefits of using fog computing. Building the programming model for fog computing platform is one of the challenged tasks. Hong et al. [8] proposes a hierarchical fog programming model. The first layer of the hierarchical structure is the cloud server and the leaves are end devices. The sensing data collected by the devices can be processed in the nodes which are placed in the middle layers of the hierarchical structure. If it can be processed in a node near to end devices, the latency becomes much lower. However, this work [8] only proposes the high level programming model without considering the practical deployment and implementation issues

Designing and implementing fog platforms, which can offload some tasks from end devices to edge servers is studied [9], [10]. Cloudlet [9] is proposed before the fog computing concept, but it is somehow similar to the concept of the fog devices. Cloudlet uses a powerful machine, placed near to users for reducing latency, which offers more resources. Users can push virtualized images to the machine to perform their tasks. Unlike our work, the machines running Cloudlets are powerful workstations or small clusters with abundant resources. ParaDrop [10] is a fog device implemented on end-user gateways. ParaDrop framework also has a centralized server to manage the devices. Different from our work, ParaDrop does not consider decomposition of requests and the problem of module deployment.

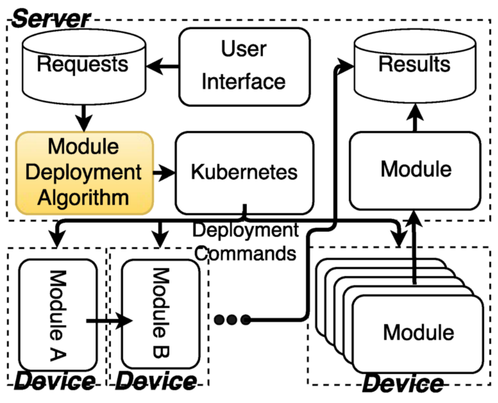


Fig. 2. The architecture of proposed fog computing platform.

III. Kubernetes-based Fog Computing Platform

Several existing open-source platforms and virtualization technologies may be leveraged to construct the fog computing platform. Our platform has to dynamically swap modules running on the devices. Such high dynamics results in several benefits: (i) we can quickly push a new module to a device, (ii) it is easy to update new algorithms as modules, and (iii) it is easy to migrate running modules to optimize the resource utilization or quality of service.

Virtualization technologies, such as Xen [11] and KVM [12] need entire guest operating systems, the necessary binaries, and libraries to create a virtual machine. It needs powerful server to launch a virtual machine, which consumes a lot of storage space and computing power. Currently, an emerging virtualization technology, called container, such as LXC [13] and Docker [14], is used in several areas. Compared to virtual machine, container requires less resources and can be set-up in a short time. More specifically, multiple containers running on the same host share the same operating system kernel, and use the namespaces to distinguish one from another. It allows modules running in containers with mandatory services they need, rather than including the full operating system.

Kubernetes [5], OpenStack [6], and SaltStack [7] are all possible open-source platforms as the starting point to implement our fog computing platform. *OpenStack* is one of the most widely used cloud management systems. It helps cloud service providers to efficiently manage their large pools of machines in data centers. OpenStack offers management services for virtual machines (Nova), images (Glance), networking (Neutron), storage (Swift and Cinder), and security (Keystone). Moreover, OpenStack provides Web-based interface (Horizon) to control all the services. OpenStack supports different virtualization technologies, including Xen [11], KVM [12], and LXC [13]. *SaltStack* is lightweight management software written in Python to remotely configure and execute parallel commands on many computers. SaltStack adopts a master/minion architecture. Master is a centralized server to control all the end devices (minions). SaltStack supports multiple operating systems, and has been integrated with Docker [14]. *Kubernetes* is a framework proposed by Google to manage a cluster of

Docker containers. Kubernetes also consists of a master and many minions, but it is more comprehensive than SaltStack, e.g., Kubernetes helps users to ensure redundancy for better fault tolerance. Each minion hosts several containers, and several containers can be allocated to support a service, termed pod. For example, administrators may create an Apache pod with dozens of containers, and Kubernetes automatically finds one or multiple minions to host these containers.

We selected Kubernetes to implement our fog computing platform, because it supports Docker, which is the latest and the most popular light-weight virtual machine. The proposed fog computing platform, shown in Fig. 2 follows a master/slave architecture. All the devices (slaves) are connected to a centralized server, which consists of three main components: (i) user interface, (ii) Kubernetes, and (iii) module deployment algorithm (MDA), which is developed in Sec. IV. Users send request to the server through user interface and the server stores the requests in the request database. Each request may be split into several modules, which are packaged by Docker as container images. The container images are pushed to the devices and the corresponding modules are started from the container images. After collecting a bunch of requests, the server executes the MDA algorithm to make module deployment decisions, which are essentially the deployment plan of the corresponding modules of requests. The deployment plan is then sent to Kubernetes, which follows the plan and sends commands to deploy the modules on the devices. Finally, the results from the modules stored in the result database and shown to the users through the user interface.

Fig. 2 also reveals that multiple small modules can help one another to finish a request and send the results back to the server. In contrast, traditionally, we use a module to collect some sensing data and use another complex module to process the sensing data on a powerful server. For example, if a user requests for the crowdedness of a specific location, traditionally, we run a face detection module, that receives images from surveillance cameras and counts the number of humans on a server. In our proposed fog computing platform, we may split the same request into three smaller modules: (i) surveillance camera, which is responsible for collecting images, (ii) feature extractor, which is responsible for extracting face features of the collected images, and (iii) face detector, which is responsible for counting the number of humans based on the extracted features. We then deploy these modules on multiple devices to share the load of server and achieve lower latency. In the rest of this paper, we focus on implementing the proposed fog computing platform, which can dynamically and easily deploy multiple modules to the devices, and we rigorously solve the module deployment problem.

IV. Module Deployment Problem and Solution

A. Problem Formulation

We consider a distributed system, which receives requests from users and serve the requests on $\mathbf{D}$ devices. We batch $\mathbf{R}$ requests and split these requests into $\mathbf{M}$ modules, which are implemented by module developers and thus are heterogeneous. Each request $r \in \mathbf{R}$ consists of $\mathbf{M_r}$ modules, where $\mathbf{M_r} \subseteq \mathbf{M}$. Moreover, the modules $\mathbf{M_r}$ of each request r have to be deployed on user-specified *feasible devices* $\mathbf{D_r}$, where $\mathbf{D_r} \subseteq \mathbf{D}$. That is, a user may specify the possible locations of the devices that are considered in the module deployment problem. We let $\mathbf{C_{max}}$ be the capacity of each device. That is, at most $\mathbf{C_{max}}$ modules can be installed on each device. Last, we let the number of (already) deployed modules on d as α_d.

$$max \sum_{r \in \mathbf{R}} \prod_{m \in \mathbf{M_r}} y_{r,m} \tag{1a}$$

$$st : y_{r,m} = \min\Big(\sum_{d \in \mathbf{D_r}} x_{d,m}, 1\Big), \forall r \in \mathbf{R}, \forall m \in \mathbf{M}; \tag{1b}$$

$$\sum_{m \in \mathbf{M}} x_{d,m} + \alpha_d \leq C_{max}, \forall d \in \mathbf{D}; \tag{1c}$$

$$\sum_{d \in \mathbf{D}} x_{d,m} \leq 1, \forall m \in \mathbf{M}; \tag{1d}$$

$$x_{d,m} \in \{0,1\}, \forall d \in \mathbf{D}, m \in \mathbf{M}. \tag{1e}$$

The objective function in Eq. (1) maximizes the number of satisfied requests. Eq. (1b) determines if module m of the request r has been deployed on device d, where $d \in \mathbf{D_r}$. Eq. (1c) guards the resource constraints of each device d. More specifically, it limits the maximal number of modules that can be deployed on d. Eq. (1d) ensures that each module m is only deployed on a device. Eq. (1e) presents the decision variable, which represents whether a module m is deployed on device d. If m is deployed on d, $x_{d,m} = 1$. The resulting formulation can be solved by existing optimization problem solvers, such as CPLEX [15], GLPK [16], or CBC [17]. Doing so, however, may be computationally intensive, as the considered problem is a variation of the NP-hard knapsack problem. Hence, we develop a heuristic algorithm in the next section.

B. Module Deployment Algorithm

The design principles of our proposed Module Deployment Algorithm (MDA) are described as follows. The MDA algorithm needs to make two decisions: (i) which request should be considered first and (ii) which device should be used for deployment first. More specifically, the MDA algorithm considers the requests with fewer modules earlier (we assume that the consumed resources of individual modules are the same). If the number of modules of the requests are the same, the MDA algorithm chooses the request with the least feasible devices. After sorting the requests, the MDA algorithm iteratively selects the device for deploying each module. It stops once all the requests are fulfilled or all the resources are used up.

Fig. 3 shows the pseudocode of the MDA algorithm. Line 1 decides the order of requests. The for-loop starts from line 2 iterates through all sorted requests and line 3 decides the order of choosing the feasible devices on popularity p_d, which is the number of modules that may be deployed on device d, based on the given requests. Line 6 checks the violation of the constraint in Eq. (1c) and line 7 assigns the value to the decision variable $x_{d,m}$. In terms of complexity, the

```
1:  sort requests $r \in \mathbf{R}$ on $|\mathbf{M_r}|$ in the asc. order; use $|\mathbf{D_r}|$ (also in
    the asc. order) to break ties
2:  for $r \in \mathbf{R}$ do //iterate with the sorted requests
3:      sort  the devices $d \in \mathbf{D_r}$ on $p_d$ in the asc. order
4:      for $m \in \mathbf{M_r}$ do
5:          for $d \in \mathbf{D_r}$ do //iterate with the sorted feasible devices
6:              if Eq. (1c) is satisfied then
7:                  let $x_{d,m} = 1$
8:                  $p_d - = 1$
9:                  $\alpha_d + = 1$
10:             end if
11:         end for
12:     end for
13: end for
```

Fig. 3. The pseudocode of our MDA algorithm for the module deployment problem.

MDA algorithm results in polynomial time, which is proved in Lemma 1.

Lemma 1 (Time Complexity)**.** *The MDA algorithm terminates in polynomial time.*

Proof. In terms of complexity, creating the sorted units list in lines 1 has a complexity of $O(|\mathbf{R}|\log(|\mathbf{R}|))$. The for-loops starting from lines 4 – 5 goes through modules $m \in \mathbf{M_r}$ in line 4 and devices $d \in \mathbf{D_r}$ in line 5, which leads to a complexity of $O(|\mathbf{M}|\,|\mathbf{D}|)$. Moreover, the devices d are sorted in line 3, which leads to a complexity of $O(|\mathbf{D}|\log(|\mathbf{D}|))$. Because line 2 goes through the requests $r \in \mathbf{R}$, the for-loops from lines 2 – 5, we have to a complexity of $O(|\mathbf{R}|\,|\mathbf{D}|\,(\log(|\mathbf{D}|)+|\mathbf{M}|))$. Hence, the overall complexity of MDA is $\max\{O(|\mathbf{R}|\log(|\mathbf{R}|)), O(|\mathbf{R}|\,|\mathbf{D}|\,(\log(|\mathbf{D}|)+|\mathbf{M}|))\}$. This yields the proof.

□

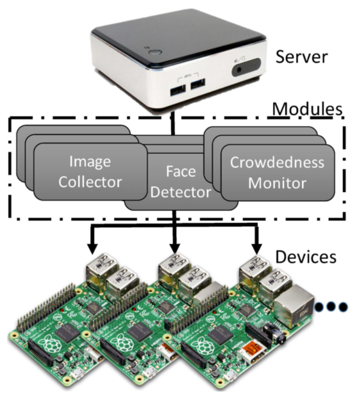
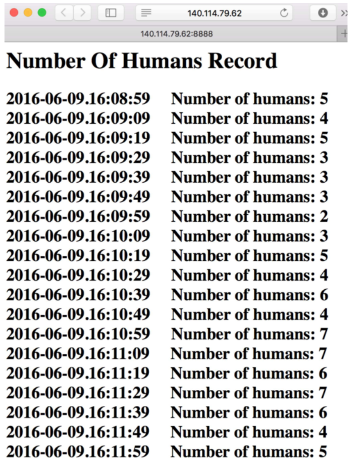


Fig. 4. Architecture of our experiment setup.

V. Experiments

A. Implementation

As illustrated in Fig. 4, we have implemented a testbed of the proposed fog computing platform using Linux machines. The server is built on a mini PC with i5 CPU and we install our MDA algorithm on it. Moreover, we also implement and install an optimal module deployment algorithm (OPT) using CPLEX [15] for comparisons on the server. The devices can be general-purpose computers, mobile devices, and IoT (embedded) devices. We adopt Raspberry PI as the devices to build the testbed. We can install many kind of sensors, such as temperature, pollution, and camera sensors on Raspberry PIs for sensing data. The server and devices are connected with each other via Ethernet. We install the Kubernetes (ver. 1.2.0) on both the server and the devices. The devices are managed by the server via the Kubernetes. Moreover, we install Docker (ver. 1.10.3) on the devices as the virtualization technology to create Docker images and virtualize the modules. The server executes the MDA algorithm to make deployment decisions. Following the decisions, the server deploys the virtualized modules to corresponding devices via Kubernetes.



Number Of Humans Record

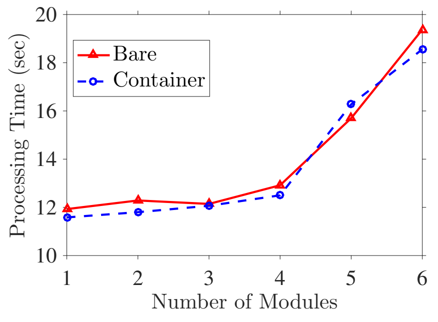
```
2016-06-09.16:08:59    Number of humans: 5
2016-06-09.16:09:09    Number of humans: 4
2016-06-09.16:09:19    Number of humans: 5
2016-06-09.16:09:29    Number of humans: 3
2016-06-09.16:09:39    Number of humans: 3
2016-06-09.16:09:49    Number of humans: 3
2016-06-09.16:09:59    Number of humans: 2
2016-06-09.16:10:09    Number of humans: 3
2016-06-09.16:10:19    Number of humans: 5
2016-06-09.16:10:29    Number of humans: 4
2016-06-09.16:10:39    Number of humans: 6
2016-06-09.16:10:49    Number of humans: 4
2016-06-09.16:10:59    Number of humans: 7
2016-06-09.16:11:09    Number of humans: 7
2016-06-09.16:11:19    Number of humans: 6
2016-06-09.16:11:29    Number of humans: 7
2016-06-09.16:11:39    Number of humans: 6
2016-06-09.16:11:49    Number of humans: 4
2016-06-09.16:11:59    Number of humans: 5
```

Fig. 5. The sample outputs of the user interface from the face detector module.

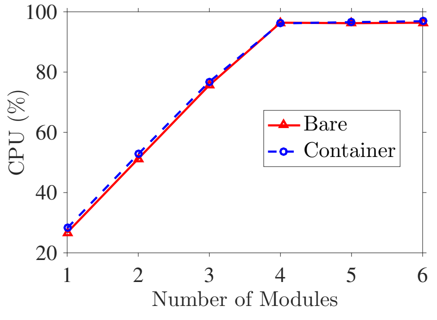
B. Experiment Setup

We set up one server and five devices in our lab. We use Wonder Shaper [18] to throttle the bandwidth the links between the devices and the links between the devices and the server to emulate environment of IoT networks. We assume that the devices communicate using WiFi mesh networks to stream data among the devices and connect to the Internet using 4G networks. The 4G network is only used for pushing container images to the devices, for the sake of high access fees. We limit the bandwidth between devices as 300 Mbps, which is the upper bound of 802.11n. We also limit the bandwidth of 4G networks as 150 Mbps. Moreover, we assume that the server uses the state-of-the-art IoT communication technology, such as LoRa [19], to connect the devices to the server for receiving deployment commands. LoRa provides long transmission distance, low power consumption and low bandwidth wireless communications. The upper bound bandwidth of LoRa is 50 kbps [19], so that we limit the bandwidth between the server and the devices with such bandwidth.

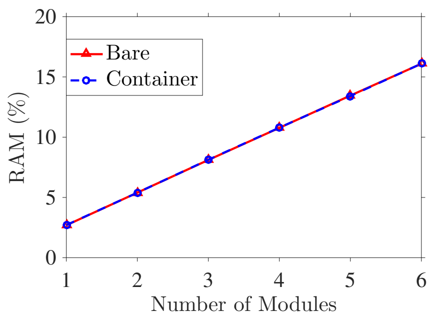
We implement three modules: (i) image collector, (ii) face detector, and (iii) crowdedness monitor. The image collector module captures images using the camera sensor installed on Raspberry PI. The face detector module counts number of



(a)

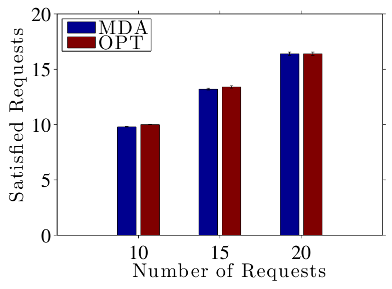


(b)

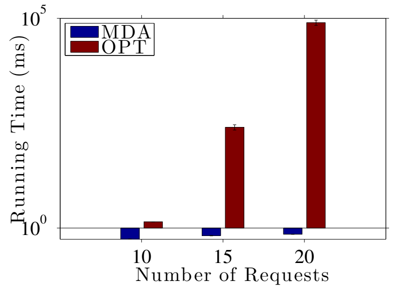


(c)

Fig. 6. Implications of different number of modules: (a) Processing Time (sec), (b) CPU utilization (%), and (c) RAM utilization (%).



(a)



(b)

Fig. 7. Implications of different number of requests: (a) satisfied requests and (b) running time.

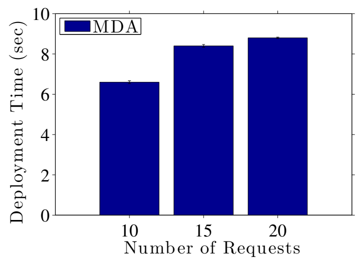


Fig. 8. The deployment time of different number of requests.

humans based on the image from the image collector module. The crowdedness monitor module measures the crowdedness based on the numbers from the face detector module. More specifically, there are three kinds of requests can be requested by users, including (i) crowdedness, which uses all the three modules, (ii) number of humans, which uses first and second modules, and (iii) image, which uses only the image collector module. For example, if a user requests for the crowdedness in a cross street, we run the MDA algorithm and deploy these three modules on corresponding devices. The image collector module collects the image of the street and sends it to the face detector module. The face detector module receives the image, extracts number of humans, and sends it to crowdedness monitor. The crowdedness monitor module receives the detected number of humans and computes the crowdedness for the user.

The results of the requests can be viewed by the users through websites. Fig. 5 shows a sample website of the face detector module. The size of the container images of these modules are 121.9 MB, 251.6 MB, 117.1 MB, respectively. The size of the container image of face detector modules is about 2 times then others because the face detector module requires the OpenCV libraries, which need more spaces. We conduct a experiment to measure the time of transmitting the container images from server to the devices over 4G networks. It takes 345, 698, and 332 seconds to transmit these three container images, respectively. In practice, these container images can be transmitted to the devices while the network is not busy, e.g., in the midnight. Hence, in the rest of the experiments, we assume the container images are already existed in the devices .

We generate the requests, where $|\mathbf{R}| \in \{10, 15, 20\}$ to evaluate our system. The requests are randomly selected from the three kinds of requests and $\mathbf{D_r}$ is also randomly selected. We conduct a measurement study to determine the maximal number of modules C_{max}, which can be deployed in a device. We use the face detector module, which consumes the most resources among the three modules to conduct the measurement. Fig. 6 shows the resources usages and processing time with/without container under various numbers of modules. The results from bare metal system (without container) are annotated as Bare in the figures. More specifically, the CPU utilization is fulled with four modules, the slope of processing time is increased while the number of modules is larger than four, and the RAM utilization is never full in our measurements. Hence, we select $C_{max} = 4$ for our experiments. Furthermore, the figure shows that the performance with/without container are almost the same. That is, the overhead of using container can be ignored. It demonstrates that using container as the virtualization technology on the resource-limited devices is a good choice.

We consider the following performance metrics and report the average performance with 95% confidence intervals whenever applicable.

- **Satisfied requests**: The number of requests, which have been successfully deployed on the devices.

- **Running time**: The running time of the module deployment algorithm.
- **Deployment time**: The time taken by deploying the modules.
- **Network traffic**: The amount of network traffics produced by the modules.

C. Results

Near-optimality and high-efficiency of the MDA algorithm. Fig. 7 reports the number of satisfied requests and the running time of MDA and OPT algorithms. Fig. 7(a) shows near-optimality of the MDA algorithm in terms of satisfied requests. More specifically, the largest gap of satisfied requests between the MDA and OPT algorithm is only 2%. Fig. 7(b) shows the efficiency of the MDA algorithm. The OPT algorithm can only solve small size of problems. The running time of the OPT algorithm is increased exponentially. The OPT algorithm takes 80 seconds to solve 20 requests, while the MDA algorithm computes it in real time (< 1 second). It also reveals that the scalability of the MDA algorithms is much higher than the OPT algorithm. We conduct another experiment to measure the running time with more requests for verifying scalability, which shows that the MDA algorithm computes 1000 requests with 500 devices in < 2 seconds (figures are not shown due to the space limitations).

Fast deployment time of the implemented fog computing platform. Because the number of satisfied requests are almost the same with the MDA and OPT algorithms, we reports some statistics of the running testbed using the MDA algorithm in this part. Fig. 8 reports the deployment time, which is the time from the server sends the commands to the devices start the corresponding modules, and to the time all the modules running. There are three states can be monitored by Kubernetes: (i) container creating, (ii) pending, and (iii) running. We send all the commands to the devices and wait for all the containers get into the running state to measure the deployment time. The figure reveals that our fog computing platform can serve the requests with very low delay. More specifically, serving 20 requests only needs 9 seconds on average.

The pre-processing modules reduces large amount of network traffics. We conduct an experiment with 3 devices and crowdedness request, which requires three modules. We deploy these three modules on different devices and measure the network traffic of the devices using *ifconfig*. We take average of the measured 3-minutes network traffics. The sending rate of the devices running image collector module, face detector module, and crowdedness module are 49.04 Mbps, 2.67 Mbps, and 0.5 Mbps, respectively. The image collector module dominates the load because it is responsible for sensing images and the crowdedness module only need 0.5 Mbps to send the final results to the server. It reveals that compared to traditional approach, which sends the image to the server using 49.04 Mpbs link, our fog computing platform only needs 0.5 Mbps. That is, it reduces 98 times of the network traffics to the server.

VI. Conclusion and Future Work

In this paper, we implemented a fog computing platform for dynamically deploying modules on fog devices. The fog devices consist of heterogeneous devices, including the devices in data centers, edge networks, and end users. We also studied the module deployment problem to maximize number of satisfied requests. We carefully formulate the problem and propose an efficient Module Deployment Algorithm (MDA). Moreover, we use an existing optimization problem solver, called CPLEX to provide optimal solutions. We implemented a real testbed to evaluate our proposed algorithms and the dynamics of the proposed fog computing platform. The evaluation results show that the proposed MDA algorithm: (i) provides near-optimal solutions, which results in at most 2% gap compared to the OPT algorithm and (ii) solves the module deployment problem in real-time and achieve high scalability, which solves 1000 requests with 500 devices in < 2 seconds, while the OPT takes 80 seconds to solve 20 request with 4 devices. Moreover, the implemented fog computing platform achieves fast deployment: it can deploy 20 requests in 9 seconds.

This paper is a starting point to use fog computing platform for dynamically deploying modules on heterogeneous devices with smaller modules. There are still many open challenges of the proposed platform. First, the decompositions to create smaller modules is difficult because of the heterogeneous devices. Second, the smaller modules are placed everywhere. Connecting these modules with dynamic flows is a critical problem to solve.

References

[1] “Gartner says 6.4 billion connected ”things” will be in use in 2016, up 30 percent from 2015,” http://www.gartner.com/newsroom/id/3165317.

[2] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, “Fog computing and its role in the internet of things,” in *Proc. of ACM Workshop on Mobile Cloud Computing (MCC)*, 2012.

[3] L. Vaquero and L. Merino, “Finding your way in the fog: Towards a comprehensive definition of fog computing,” *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 5, pp. 27–32, 2014.

[4] S. Yi, Z. Hao, and Q. Li, “Fog computing: Platform and applications,” in *Proc. of IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*, 2015.

[5] “Kubernetes,” http://kubernetes.io/.

[6] “OpenStack,” https://www.openstack.org/.

[7] “Saltstack,” https://saltstack.com/.

[8] K. Hong, D. Lillethun, U. Ramachandran, B. Ottenwalder, and B. Koldehofe, “Mobile fog: a programming model for large-scale applications on the internet of things,” in *Proc. of ACM SIGCOMM workshop on Mobile Cloud Computing (MCC)*, 2013.

[9] M. Satyanarayanan, P. Bahl, R. Cceres, and N. Davies, “The case for VM-based cloudlets in mobile computing,” *IEEE Transactions on Pervasive Computing*, vol. 8, no. 4, pp. 14–24, 2009.

[10] D. Willis, A. Dasgupta, and S. Banerjee, “ParaDrop: a multi-tenant platform for dynamically installed third party services on home gateways,” in *Proc. of ACM SIGCOMM Workshop on Distributed Cloud Computing (DCC)*, 2014.

[11] “Xen,” http://www.xenproject.org/.

[12] “KVM,” http://www.linux-kvm.org/.

[13] “LXC,” https://linuxcontainers.org.

[14] “Docker,” https://www.docker.com.

[15] “IBM CPLEX optimizer,” http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/.

[16] “GLPK,” https://www.gnu.org/software/glpk/.

[17] “CBC,” http://www.coin-or.org/projects/Cbc.xml.

[18] “Wonder Shaper,” http://lartc.org/wondershaper/.

[19] “LoRa,” https://www.lora-alliance.org/What-Is-LoRa/Technology.