The Fog Development Kit: A Platform for the Development and Management of Fog Systems

Colton Powell, Christopher Desiniotis, and Behnam Dezfouli

***Abstract*—With the rise of the Internet of Things (IoT), fog computing has emerged to help traditional cloud computing in meeting scalability demands. Fog computing makes it possible to fulfill real-time requirements of applications by bringing more processing, storage, and control power geographically closer to end devices. However, since fog computing is a relatively new field, there is no standard platform for research and development in a realistic environment, and this dramatically inhibits innovation and development of fog-based applications. In response to these challenges, we propose the Fog Development Kit (FDK). By providing high-level interfaces for allocating computing and networking resources, the FDK abstracts the complexities of fog computing from developers and enables the rapid development of fog systems. In addition to supporting application development on a physical deployment, the FDK supports the use of emulation tools (e.g., GNS3 and Mininet) to create realistic environments, allowing fog application prototypes to be built with zero additional costs and enabling seamless portability to a physical infrastructure. Using a physical testbed and various kinds of applications running on it, we verify the operation and study the performance of the FDK. Specifically, we demonstrate that resource allocations are appropriately enforced and guaranteed, even amidst extreme network congestion. We also present simulation-based scalability analysis of the FDK versus the number of switches, the number of end devices, and the number of fog devices.**

***Index Terms*—Edge computing, fog computing, Internet of Things (IoT), resource allocation and management, software-defined networking (SDN).**

I. Introduction

In TODAY'S world of smart cars, smart cities, smart homes, Industry 4.0, and mobile healthcare, almost every device is connected to the Internet. With the growing number of interconnected devices and Internet of Things (IoT) applications arises the challenges of communicating and processing a massive amount of data in a highly efficient manner.

Cloud computing offers a partial solution to this dilemma by providing massive infrastructure and powerful applications. However, cloud computing is not suitable for real-time and mission-critical application domains with stringent runtime and latency requirements. Additionally, cloud computing cannot scale sufficiently to handle the processing and communication demands of billions of IoT devices [1], [2]. Fog computing aims to solve this challenge by bringing additional computing capabilities to the network edge, especially to reduce communication delay and overhead. The increased number of powerful computation and networking platforms has made the implementation of fog architectures a worthwhile undertaking [3]–[5]. Fog is intended to work alongside the cloud, forming a things-fog-cloud continuum where applications can be served promptly.

By using the things-fog-cloud continuum, requests generated by end devices (things) can be serviced in the fog, thereby avoiding transmission to the cloud and significantly reducing packet latency and network congestion. Resource-constrained devices, such as medical devices can offload computation- and communication-intensive tasks to nearby fog devices to meet real-time constraints. For example, consider a scenario where medical devices in a hospital monitor patients. Once a medical device detects an anomaly, it can request resources from fog devices for further processing and real-time results. We refer to these systems as *fog systems*, where applications on end devices may offload their computational tasks to nearby fog devices. These systems may optionally connect to cloud data centers for increased accuracy in the decision-making process.

There exist significant obstacles for research and development in the realm of fog systems. First, end devices need to request and *reserve* resources to meet the Quality-of-Service (QoS) demands of underlying applications, meaning that any efficient fog system must operate with a resource allocator. Traditional load balancers are not sufficient in fulfilling the needs of heterogeneous IoT applications, where end devices require *guaranteed* resources to meet their stringent runtime and latency requirements. While many resource allocation platforms have been proposed [6]–[8], few systems allocate both networking and computing resources. Furthermore, to the best of our knowledge, no such platforms have integrated software-defined networking (SDN) into their architecture, where fog device resource allocation, network bandwidth allocation, and customizable routing policies are all consolidated into a single, comprehensive platform. Second, most of the existing works employ simulation to evaluate the efficiency of their resource management approaches [7]–[12]; thereby highlighting an apparent lack of development tools for research in this field. In order to exhaustively test new approaches in realistic environments, and to accelerate research in fog computing, a standard research and development platform is needed. Finally, it can be quite expensive to prototype and

Manuscript received November 25, 2019; accepted December 30, 2019. Date of publication January 13, 2020; date of current version April 14, 2020. *(Corresponding author: Behnam Dezfouli.)*

The authors are with the Internet of Things Research Lab, Department of Computer Science and Engineering, Santa Clara University, Santa Clara, CA 95050 USA (e-mail: ctpowell@scu.edu; cdesiniotis@scu.edu; bdezfouli@scu.edu).

Digital Object Identifier 10.1109/JIOT.2020.2966405

test the performance of a real fog-based application. For example, creating even the most straightforward application requires constructing an infrastructure of end devices, fog devices, and networking hardware, which can be costly. Therefore, the creation of *complex* software components and a *costly* physical infrastructure must precede the development of such applications. This combination of complexity and cost poses an immense barrier of entry for researchers and engineers. Since fog computing is still in its infancy, there is no standard development kit or platform which has solved all of these issues in the form of a single, complete development package. Without such a platform, the advancement of pertinent, real-time applications will be slow, given the barriers of entry.

In this article, we set out to address this problem by proposing the Fog Development Kit (FDK)[1]: A development and management platform for fog systems. The FDK is intended to bring together all of the elements of fog computing into one comprehensive framework, where developers can begin building fog-based applications with ease and without the barriers mentioned above.

The FDK addresses development complexity by providing a cutting-edge resource allocation scheme, which supports any arbitrary fog-based application running on top of it. Specifically, by integrating SDN and virtualization technologies, the FDK enables end devices to utilize its messaging protocol to request for computing and communication resources. If sufficient resources are available, the FDK instantiates a container on a fog device with the desired computing resources, finds an efficient path through the network for communication between the end device and the fog device, and allocates the requested bandwidth along the identified path. The complexity of resource allocation is thus handled by the FDK. For example, suppose a developer plans to build a facial recognition system, where resource-constrained end devices connected to cameras live-stream video data to fog devices for heavy-duty processing. Here, it is only required to develop an application for the end device to collect and stream video data, as well as the containerized services running on fog devices to receive and process the data. The FDK handles all of the underlying system complexities, such as managing computational resources of fog devices, path reservation, and bandwidth slicing between end devices and fog devices.

The FDK supports application development in both physical and emulated environments. Built on top of OpenDaylight (ODL) [13], the FDK utilizes standard SDN protocols to communicate with physical network devices. Moreover, the FDK is designed to be used in unison with Open vSwitch (OVS) [14], which performs network resource allocation using the OVSDB management protocol [15] and enforces data flow routing using the OpenFlow protocol [16]. Therefore, in addition to supporting physical environments, the FDK was designed to be used with emulation technologies so that developers could leverage tools, such as GNS3 [17] and Mininet [18] to prototype fog-based applications. GNS3 and Mininet provide the capability of emulating network topologies on a personal computer. These tools allow virtual machines (VMs) and containers running on the computer to communicate with each other in a virtualized environment. With this, the FDK can run on a completely emulated network consisting of Linux VMs serving as end devices and fog devices, and OVS VMs which handle the messages exchanged between these devices. Therefore, the FDK enables the development of applications in an emulated environment at zero additional cost. In addition, any applications developed on top of the FDK can be ported from an emulated environment to a physical infrastructure.

[1]The FDK is accessible at the following address: https://github.com/SIOTLAB/Fog-Development-Kit.git.

We evaluate the correctness and performance of the FDK by using a physical testbed consisting of eight end devices, four fog devices, and five OpenFlow switches. Our results show that resource allocation and deallocation delays are less than 279 and 256 ms, respectively, for 95% of transactions. We also evaluate the resiliency of the FDK by analyzing the impact of various network conditions and congestion levels on communication bandwidth between end devices and fog devices. We show that bandwidth allocations are accurately enforced and upheld regardless of network conditions. In addition, we present a simulation-based scalability analysis to demonstrate the impact of network size, topology type, number of end devices, and number of fog devices on controller overhead and communication delay.

The remainder of this article is organized as follows. We present the related work in Section II. In Section III, we summarize the goals and features of the FDK. Section IV presents the system architecture and operation of the FDK. In Section V, we present a performance evaluation using a physical testbed and simulation. In Section VI, we highlight potential future work, and finally in Section VII we conclude this article.

II. Related Work

In this section, we overview relevant simulation platforms and justify the importance of the FDK. Also, we summarize existing load balancing and resource allocation schemes and identify their shortcomings when applied to fog-based applications. Finally, we investigate other existing fog architectures and platforms, and highlight the benefits that the FDK holds over these alternatives.

A. Simulation Platforms

Due to the significant cost of creating fog and cloud network infrastructures, a simulation-based study is the most widely used approach to evaluate the performance of proposed mechanisms [7]–[9].

CloudSim [19] is perhaps the most popular cloud simulation platform available, which is used for modeling the cloud and application provisioning environments. It is a discrete event-based simulator written in Java, meaning that it does not actually emulate (virtualize) network entities, such as routers and switches. Instead, CloudSim uses a latency matrix, which contains predefined values for the latency between entities. Additionally, CloudSim can model dynamic user workloads by exposing a set of methods and variables to configure the resources of simulated VMs.

There are also many extensions to CloudSim, such as CloudSimSDN [10], ContainerCloudSim [11], and iFogSim [12], which attempt to broaden CloudSim's model to include SDN, container migration simulation, and fog computing, respectively. However, because CloudSim and these associated extensions are strictly simulation-based, they ultimately do not solve the problems of cost and complexity associated with developing an actual fog application. Rather, they simply avoid the problem altogether by simulating the entire system. Therefore, while CloudSim is a worthy platform for evaluating cloud architectures, load balancing algorithms, etc., it fails to actually serve as a valid fog-based application development platform because projects developed in CloudSim are not portable to a real environment. Likewise, the same can be said for most other simulation platforms for similar reasons. In contrast, the FDK can be used to develop actual fog applications in both physical and emulated environments. Furthermore, after a fog application is developed in an emulated environment, that application can then be seamlessly ported to a physical environment (and *vice versa*).

B. Resource Management and Allocation

Resource management is key to the success of any fog system and consists of two main components: 1) networking resource management and 2) computational resource management.

Typically, networking resource management is accomplished using a load balancer, which attempts to find a suitable path to one or more destinations while spreading traffic throughout the network to avoid congestion. In many cases, equal-cost multipath (ECMP) routing is used to manage network resources by distributing traffic throughout the network. However, ECMP is congestion oblivious and studies reveal that ECMP's performance is far from optimal and results in unevenly distributed network flows and poor performance [20], [21]. In response, Katta *et al.* proposed Clove [20], a congestion-aware load balancer that works alongside ECMP by modifying encapsulation packet header fields to manipulate flow paths, ultimately providing lower flow completion times (FCT) than ECMP. Clove identifies disjoint paths and changes the 5-tuple of the overlay network to distribute traffic over these paths. It also uses ECN to detect congestion. Similarly, Zhang *et al.* proposed Hermes [21], a distributed load balancing system, which offers up to 20% faster FCT than Clove. While Clove can handle link failures and topology asymmetry, Hermes can handle more advanced and complex uncertainties, such as packet black holes and switch failures.

Unfortunately, load balancers do not adequately fulfill the network resource management requirements of fog systems. Load balancers simply find multiple paths for traffic distribution, whereas fog systems need to actually *reserve* bandwidth along paths to fulfill application demands, such as real-time exchange of medical monitoring data.

There are mechanisms that utilize actual network resource allocation to provide timely and reliable services. Akella and Xiong [22] proposed a method for guaranteeing network resources and reliable QoS. They leverage OVS, OVSDB, and SDN technologies to create three tiers of cloud QoS levels, where each tier allocates a specific amount of bandwidth to a user-cloud service. This is performed by dynamically creating packet queues on switches along the communication path, followed by then creating OpenFlow flows on those switches that enqueue traffic belonging to one of the QoS levels onto the appropriate packet queue. Kumar *et al.* [23] proposed a mechanism to extend SDN infrastructure to be "delay aware" by finding paths for data flows to ensure end-to-end delays are guaranteed. To this end, they use a similar scheme where packet queues are dynamically created along a path. Then, one flow entry is created and assigned per queue, and all packets belonging to a critical network flow are forwarded to the packet queue associated with that flow. They also propose a path selection algorithm to meet the desired delay and bandwidth constraints of each flow.

On the other hand, computational resource management often involves the use of VMs and containers, which can be configured to use a specific, limited amount of resources. The amount of resources allocated to a VM or container directly affects the execution time of tasks and services. Therefore, the allocation of these resources is critical in ensuring the timely processing of essential data. Containers hold an advantage over VMs in the context of resource allocation in the fog, as they tend to be more lightweight and, more importantly, provide finer granularity in allocating resources. For example, when allocating processing power to VMs, the available options only allow for the specification of the number of entire CPU cores that a particular VM can use. On the other hand, container technologies like Docker [24] provide interfaces for specifying more in-depth options when running a container. For example, container options allow the specification of a fractional number of cores that can be used (e.g., 1.25 CPU cores), in addition to the proportion of CPU cycles that can be utilized, which enables more precise, granular control of resource allocation.

Container management is typically performed through the use of orchestration software, such as Docker swarm mode [25] or Kubernetes [26]. These tools provide functionality for remotely managing, instantiating, and shutting down containers. These container orchestrators currently serve as the backbone for computing resource allocation in fog and cloud systems, and current research involves more advanced use cases, such as investigating and optimizing live container migration techniques [27]. This is critical to the success of such systems, as live migration may interrupt running services, degrading performance and increasing completion delays. Ansari and Sun [9] investigated approaches to resource management and VM migration for fog-based IoT applications in mobile networks. They proposed latency aware proxy VM migration (LAM), which solely considers latency when assigning a fog colony to a mobile IoT device, and energy-aware proxy VM migration (EAM), which considers the energy consumption of colonies. They simulated LAM, EAM, and static VM allocation, compared all the three approaches and discussed the tradeoffs involved. For simulation, they used EveryWare Lab's user movement trace [28] to emulate movement patterns of mobile devices. However, the authors acknowledge the need to conduct further experiments on physical infrastructure.

C. Fog Architectures and Platforms

Many fog architectures involving automated resource management have been proposed. Skarlat *et al.* [8] created a resource provisioning system using a fog-cloud middleware component. The middleware oversees the activity of fog colonies, which are micro data centers consisting of fog cells where tasks and data can be distributed and shared among the cells. This system merely manages fog computing resources and does not *allocate* those resources, nor does it perform any allocation of network resources.

Yin *et al.* [7] built a novel task-scheduling algorithm and designed a resource reallocation algorithm for fog systems, specifically for real time, smart manufacturing applications. However, unlike the previous work, a management software component is not used in their approach, and each fog device is burdened with the task of deciding whether to accept, reject, or send requests to the cloud. Resource reallocation is periodically run on a single fog device, reallocating resources among tasks in order to meet delay constraints. Their results show reduced task delays and improved resource utilization of fog devices. However, their experiments are strictly simulation-based, and the resource management scheme only includes a single fog device during decision making.

Finally, Wang *et al.* [6] proposed a novel resource management framework for edge nodes called ENORM. Upon startup of the system, an edge manager software installed on all edge nodes gathers and stores available system resources. Then, each edge node listens for resource requests from a cloud manager software installed on a cloud server. Each resource request starts with a handshaking process that eventually leads to the initialization of a fog application. In contrast, fog devices in our proposed scheme are managed by a central controller running the FDK. The FDK then receives service requests from end devices requesting the instantiation of a fog application service with a specific amount of resources. If sufficient resources exist, the FDK leverages SDN and containerization technologies to remotely perform both computational and network resource allocation.

III. Fog Development Kit

The FDK addresses all of the problems mentioned in Section II by enabling the creation and deployment of fog-based applications in physical and emulated environments. The FDK also provides a comprehensive SDN-based resource allocation scheme. This section presents the main features offered by FDK.

A. Resource Allocation

The FDK provides comprehensive resource allocation capabilities to ensure that requests made by end devices are fulfilled completely and in a timely manner. This is accomplished by providing a resource allocation scheme where both network resources and fog devices' computational resources can be sliced and allocated. This automated resource allocation offers several benefits. First, it ensures that the services requested by end devices own a dedicated slice of the network, and provides the possibility of guaranteeing network latency and bandwidth for communication with fog devices. Second, to guarantee application processing deadlines, it ensures that fog devices are not overwhelmed by end devices' requests. These two features are essential in many fog systems as they ensure expedited processing and seamless interactions between end devices and fog devices, which are key advantages that fog computing holds over cloud computing.

B. Agility

Working with the FDK does not necessarily require access to a physical testbed. To support emulated topologies and in order to reduce the development and prototyping costs of fog-based applications, developers can use tools, such as GNS3 [17] and Mininet [18] to build a complete network of end devices, fog devices, and OVS nodes using VMs and containers. Another VM can be used as the controller, running the FDK and SDN controller software. The controller VM fulfills the requests made by end devices by allocating resources and instantiating containerized services in fog devices. Therefore, the FDK offers agility to developers by making it possible to quickly begin creating fog-based applications using only a personal computer, while also making the process significantly cheaper.

C. Portability

Fog-based applications running on emulated topologies may need to be ported over to physical, production topologies once they are complete. To meet this need, fog-based applications written on top of the FDK are portable in their entirety to physical systems. To satisfy this, the FDK can be installed on a virtual or physical Linux machine (acting as a central controller) with Python 3, Docker, ODL, and the necessary ODL plug-ins installed. In addition, in order to take advantage of the network resource allocation capabilities of the FDK, the switching devices throughout the topology must support the OpenFlow 1.3 and OVSDB protocols. Considering the widespread acceptance of OVS in large-scale environments [29], we used OVS as our switch software. OVS can be installed on any virtual or physical Linux machine. Finally, large vendors, such as Cisco and Juniper Networks also carry OpenFlow 1.3 and OVSDB compatible switches [30], [31], which could allow for a port of the FDK and any fog-based applications developed on top of it to a production-grade physical network.

D. Application Independence

The key principle that the FDK is designed to fulfill is *application independence*. That is, the FDK aims to support any general fog-based applications in order to ensure that a variety of heterogeneous services can be developed. To this end, the FDK provides a *messaging protocol* for end devices to request resources and instantiate specific containerized applications on the fog devices to handle their processing needs. Conversely, the messaging protocol also provides methods to deallocate resources and terminate containerized applications. Therefore,

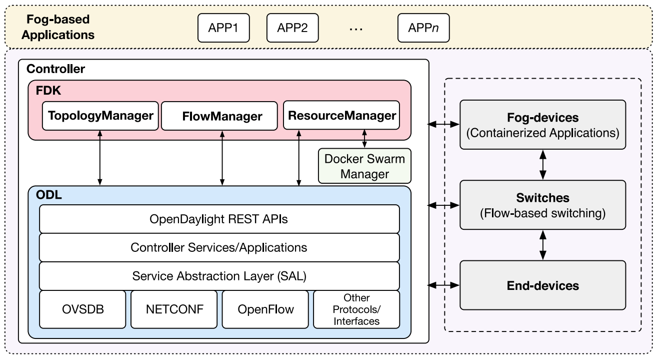

Fig. 1. Overall system architecture.

as long as all resource requests follow this protocol, any fog-based application can request resources and leverage the power of fog devices. In Section IV-C, we show how various types of applications can be adapted to work with the FDK.

IV. System Architecture

Fig. 1 shows the overall fog system architecture including four major components: 1) controller; 2) end devices; 3) switches; and 4) fog devices. End devices are resource-constrained and cannot completely satisfy application requirements [2]–[4], [32]. Therefore, these devices communicate with more powerful machines—fog devices—to offload their computing[2] requirements. For example, an end device may represent a Raspberry Pi board that captures images and streams them to fog devices running object recognition algorithms. Another example of an end device is a smartphone that collects sensory data from multiple medical devices and transmits them to fog devices for anomaly detection. End devices and fog devices are connected through switches that support flow-based forwarding. End devices, switches, and fog devices are referred to as *nodes*. Nodes are monitored and configured by the controller running the FDK.

The FDK itself is a user-space application that operates within the controller and oversees the operation of end devices, fog devices, and switches. The FDK interacts with ODL to control communication paths and manage network resource allocation, and also leverages the Docker containerization technology to remotely instantiate services on fog devices with a specific amount of resources.

ODL is an SDN controller software that enables remote management and configuration of networks. In the case of the FDK, these capabilities are leveraged using ODL's northbound REST interfaces and the model-driven service abstraction layer (MD-SAL). At a high level, the MD-SAL allows developers to define data models for ODL software plug-ins and extend the functionality of ODL. These plug-ins provide additional *northbound* REST APIs. Invocations of these APIs may utilize a variety of *southbound* network management protocols, such as OpenFlow, NETCONF [33], and OVSDB to control and manage network devices. These invocations must also include a data body that is in accordance with the YANG data model [34] defined by the corresponding ODL plugin being utilized. Upon validation of the data body, it is pushed to the MD-SAL's *configurational data store*, which reflects the desired configuration of the network. Then, the corresponding plug-in uses the information placed in the configurational data store to apply the desired changes to the appropriate network devices using southbound protocols and interfaces. Once applied, these changes are reflected in the MD-SAL's *operational data store*, which represents the actual, physical state of the network. In effect, the MD-SAL supports the development of extensions to ODL, making it an extensible, modular, and versatile SDN controller that has the ability to grow and evolve over time. In particular, the FDK utilizes ODL's comprehensive set of northbound REST APIs to perform network control and management using a variety of southbound protocols. For example, the FDK controls and manages the switches via OpenFlow 1.3 and OVSDB, respectively, even though the only interfaces accessed by the FDK are ODL's northbound REST APIs.

Docker [24] is a platform that allows for building, sharing, and executing applications within containers. Each container is defined by an *image* file, which specifies its exact contents. Image files are typically stored in centralized repositories and are accessible by remote compute nodes. Docker deploys containers by downloading the image file from the remote repository (unless the image is already cached locally) and then instantiates the container using this file. Docker swarm mode is a feature that allows for the management and orchestration of such containers on remote machines. Because these containers have specifiable resource allocation parameters, the FDK leverages Docker swarm mode to provide fog device resource allocation capabilities and to instantiate containerized services for end devices.

The FDK combines and builds upon the functionality of Docker and ODL using *three manager objects* that oversee the entire network and provide interfaces for querying data and manipulating the topology. These objects are detailed in the rest of this section.

A. *TopologyManager*

The FDK uses a *TopologyManager* component to query, update, and manage the network topology. The core APIs for this component are described in Table I. On startup, the TopologyManager first issues queries to the MD-SAL's operational data store for data pertaining to the ODL OpenFlow plugin, the ODL node inventory, and the OVSDB plugin to gather data on the entire topology. The results returned by the OpenFlow plugin include information regarding all nodes (end devices, fog devices, switches) and links. The results returned by the ODL node inventory contain more in-depth information on the OpenFlow switches and their network interfaces, and provide information on the speed of the interfaces and how much data has been transmitted across them since ODL started. Finally, the results returned by the OVSDB plugin contain information about the configuration of OpenFlow switches as well as the information required to configure them remotely.

[2]Here, *computing* is a general term that includes processing, storage, and communication.

TABLE I
TOPOLOGYMANAGER APIs

API	Description
`update_topology()`	Query topology information from ODL and update the topology
`create_queue()`	Create/update rate-limited queue on switch
`delete_queue()`	Delete queue from switch
`create_qos()`	Create QoS entry on switch
`delete_qos()`	Delete QoS entry from switch
`place_queue_on_qos()`	Place queue on QoS entry
`remove_queue_from_qos()`	Remove queue from QoS entry
`place_qos_on_port()`	Place QoS entry (containing queues) on switch port
`remove_qos_from_port()`	Remove QoS entry from switch port

The TopologyManager consolidates all the information returned by these calls within a single topology object, which models the network topology as a graph. Links are modeled as directed graph edges, with each one containing multiple data fields, such as the current utilization of the link, the current bandwidth allocations on the link, and port identifiers at the endpoints. The nodes across the network are modeled as end devices, fog devices, and switches using a set of device type classes provided within the topology object. The data stored for each node varies depending on its type. For example, each fog device object contains information, such as the total amount of processing and memory resources on the device, which is later used by the FDK to slice the resources and prevent over-allocation. Similarly, the OpenFlow switch objects store information regarding their current configurations and the flows installed in their flow tables, which is later used by the FDK to shape network traffic paths and to manage the allocation of communication resources. Therefore, the TopologyManager serves as a comprehensive directory of information pertaining to the state and structure of the network and the availability of resources across it.

After building the topology object, the TopologyManager creates a background thread to continuously update the network topology over time. This thread issues the previously mentioned queries to the ODL operational data store to gather information on the latest state of the topology. Then, the thread analyzes the differences between the returned data and the current topology object, and then updates the topology object to reflect the more recent topology information returned by ODL by making the appropriate changes (such as adding links and/or nodes).

The TopologyManager also provides a large number of APIs for managing OpenFlow switches via the OVSDB management protocol. These interfaces provide capabilities for creating and deleting constructs, such as packet queues, QoS entries, and ports, which are used by the ResourceManager component of the FDK when allocating network resources. It should be noted that all OpenFlow data and OVSDB data are originally returned as separate topologies by ODL, and there is no immediately apparent way to relate data between the two. In the case of the OVSDB data, the MAC address of the bridge being controlled by ODL is returned in the query to the OVSDB plugin, which can then be converted to an OpenFlow node ID by stripping out the colons in the MAC address, converting the remaining hex value to a decimal value, and prepending "`openflow:`" to the remaining decimal value. The FDK then uses this relationship when storing data in topology objects, and effectively merges the two separate OpenFlow and OVSDB data sets into the single aforementioned topology object.

Finally, the TopologyManager provides a *greeting server* thread used to handle greeting messages sent by end devices and fog devices. End devices and fog devices are configured to send greeting messages upon boot up. Each message contains a device type and a node ID field, as well as some supplementary information. The device type field specifies whether the device is a fog device or an end device, and the node ID correlates the device with one that was found in the MD-SAL operational data store. By building this association via greeting messages, the TopologyManager can identify all of the nodes in the topology and establish if they are an end device, a fog device, or neither. These associations are key for differentiating devices and establishing what actions are appropriate to perform on a particular device. For example, the FDK only instantiates services on fog devices, as such an action would not be appropriate for other devices. Section IV-C presents this mechanism in detail.

TABLE II
FLOWMANAGER APIs

API	Description
`create_flow()`	Push OpenFlow flow to switch
`delete_flow()`	Delete OpenFlow flow from switch
`track_flow()`	Track flow information
`untrack_flow()`	Untrack flow information

B. FlowManager

The *FlowManager* component provides a comprehensive interface for the management of OpenFlow flows throughout the network. The core APIs for this component are described in Table II. First, the FlowManager provides a set of APIs to simplify the process of creating flow table entries on OpenFlow switches. For example, this component provides a method for creating a *flow skeleton*, which contains all of the basic fields needed to create the flow table entries used by the FDK to enforce traffic paths between end devices and fog devices. Then, the FlowManager's flow-modification APIs can be utilized to further build and shape entries by adding flow actions, flow match fields, and other constructs to a flow skeleton. For example, flows can be created to match packets by source and destination IP address (or additional identifiers). Upon a match, multiple actions can be applied to a packet—such as transmitting it through a specific port (used to create network traffic paths) and placing it on a packet queue. Once a flow table entry is built, the FlowManager's flow-creation APIs can be leveraged to push a newly built entry to an OpenFlow switch. Similarly, the FlowManager offers flow-deletion APIs that can be used to remove such entries.

C. ResourceManager

The FDK uses the *ResourceManager* component to manage and allocate all networking and computing resources.

TABLE III
RESOURCEMANAGER APIs

API	Description
`service_end_device()`	Process service requests from end-devices, run the RAA, and instantiate containers
`service_shutdown_request()`	Process shutdown requests, run the RDA, and shutdown containers
`service_fog_device()`	Receive and process resource reporting messages from fog-devices
`resource_alloc_algorithm()`	Attempt to allocate all resources for requested service
`resource_dealloc_algorithm()`	Attempt to deallocate all resources for a service

Algorithm 1: RAA

```
Input:
$e_i$ = end-device requesting resources
$R_B(e_i)$ = Bandwidth requirement of request from $e_i$
$R_P(e_i)$ = Processing requirement of request from $e_i$
$R_M(e_i)$ = Memory requirement of request from $e_i$
Complete topology and resource data (from TopologyManager)
Output:
A response for $e_i$ indicating success or failure

1  $T_B(l)$ = Total bandwidth capacity on link $l$
2  $T_P(f)$ = Total processing capacity on fog device $f$
3  $T_M(f)$ = Total memory capacity on fog device $f$
4  $A_B(l)$ = Allocated bandwidth on link $l$
5  $A_P(f)$ = Allocated processing on fog device $f$
6  $A_M(f)$ = Allocated memory on fog device $f$
7  $\mathcal{N}$ = Set of all nodes
8  $\mathcal{L}$ = Set of all links
9  $\mathcal{F}$ = Set of all fog devices
10 $\mathcal{F}' = \emptyset$ //Request servicers
11 $\mathcal{P} = \emptyset$ //Shortest-path tree
12 $\mathcal{B} = \emptyset$ //Best known link dictionary
13 //identify request servicers
14 for $f_j \in \mathcal{F}$ do
15     if $T_P(f_j) - A_P(f_j) > R_P(e_i)$ &
16     $T_M(f_j) - A_M(f_j) > R_M(e_i)$ then
17         Add $f_j$ to $\mathcal{F}'$
18 if $\mathcal{F}' == \emptyset$ then return FAILURE response
19 $k = \max(2, \text{size}(\mathcal{L})/\text{size}(\mathcal{N}))$
20 $\mathcal{H} = minHeap(k)$ //K-ary min heap
21 $init\_link = (src{:}e_i, dst{:}e_i, weight{:}0)$
22 $\mathcal{H}$.push($init\_link$)
23 $\mathcal{B}[e_i] = init\_link$
24 //find least-cost paths from $e_i$ to fog devices
25 while size($\mathcal{H}$) > 0 do
26     $u = \mathcal{H}$.pop_min()
27     if $u.src \neq u.dst$ then
28         $\mathcal{P}[u.dst] = u$
29     for $v \in$ {outgoing links of $u.dst$} do
30         //v.src is equivalent to u.dst
31         $v.weight = \mathcal{B}[v.src].weight + 1/(T_B(v) - A_B(v))$
32         if $(T_B(v) - A_B(v)) < R_B(e_i)$ then $v.weight = \infty$
33         if $v.dst \notin \mathcal{B}$ then
34             $\mathcal{H}$.push($v$)
35             $\mathcal{B}[v.dst] = v$
36         else if $v.weight < \mathcal{B}[v.dst].weight$ then
37             //Update link, shift based on weight
38             $\mathcal{H}$.decrease_key($\mathcal{B}[v.dst], v$)
39             $\mathcal{B}[v.dst] = v$
40 //find the best fog device to fulfill the request
41 $min = \infty$
42 for $f_j \in \mathcal{F}'$ do
43     if $\mathcal{P}[f_j].weight < min$ then
44         $min = \mathcal{P}[f_j].weight$
45         $f_{min} = f_j$
46 if $min == \infty$ then return FAILURE response
47 //configure switches along the path $f_{min}$ to $e_i$
48 $v = \mathcal{P}[f_{min}]$
49 while $true$ do
50     if $v.src == e_i$ then return SUCCESS response
51     Create rate-limited queues on $v.src$
52     Place queues on appropriate QoS entry in $v.src$
53     Create flows on $v.src$ to redirect traffic to rate-limited queues
54     $v = \mathcal{P}[v.src]$
```

The core APIs of this component are described in Table III. The ResourceManager maintains data structures regarding all resources available in the network. This is possible with the help of an agent running on every fog device. This agent continually collects and relays information (such as processor and memory utilization) back to the ResourceManager over time. Similarly, the ResourceManager also repeatedly queries the ODL node inventory to gather current link utilization information. This information is then stored in the topology data structure managed by the TopologyManager, which ultimately provides a complete overview of all available resources throughout the network.

The main functionalities provided by the ResourceManager lie within the servers that enable the end devices to request/release computing resources. These servers act as an interface for managing containerized services and the allocation of resources. For example, the *service request server* receives and processes requests from end devices, where each request specifies parameters, such as an image name of a containerized service to run and a set of resource requirements for the request. The image name refers to the type of application processing requested. For example, an end device may specify an image implementing a medical classification application.

Once a request is received, the ResourceManager executes the resource allocation algorithm (RAA) presented in Algorithm 1. If sufficient resources exist, the desired containerized service with the appropriate amount of resources is instantiated on a fog device, a communication path between the end device and the fog device is selected, and a bandwidth allocation along that path is enforced. Conversely, the *shutdown request server* provides an interface to revert this process by shutting down containers and deallocating resources.

The RAA uses a modified version of Dijkstra's shortest-path algorithm in addition to some pre- and post-processing steps. The implementation of Dijkstra's algorithm leverages a k-ary min heap for optimal real-world performance [35]. If n is the number of nodes and m is the number of links, then $k = \max(2, m/n)$ is the number of children per node in the k-ary heap. It has been shown that this algorithm has a runtime complexity of $O(m \log_k n)$[36]. Although there are theoretically faster implementations of this algorithm using a Fibonacci heap, the k-ary heap implementation is known to be significantly faster in real-world scenarios [35]. The RAA's inputs are an end device e_i, the resources requested by e_i, and complete topology data.

The RAA begins with a preprocessing step, where it iterates over all fog devices f_j and assesses their available resources to create a list of *request servicers* $\mathcal{F}'$ (line 14). Specifically, $\mathcal{F}'$

is a list of fog devices that have sufficient resources to fulfill the request. Afterwards, if no request servicers exist, then the RAA returns a failure response that is subsequently sent back to e_i by the ResourceManager (line 18).

If at least one request servicer exists, then the RAA continues and executes Dijkstra's shortest-path algorithm to find the shortest path from e_i to all other nodes in the topology. The algorithm defines the cost across any link l, from the node l.src to the node $l.dst$, as $1/(T_B(l) - A_B(l))$. It is important to note that the amount of available bandwidth on the link l, computed as $T_B(l) - A_B(l)$, is never affected by control and management traffic (e.g., OVSDB messages, service requests, etc.) because a separate allocation for this traffic is made when the FDK initially starts. However, the actual weight of l is defined as the total cost required to reach $l.dst$ from e_i (unless there is insufficient bandwidth on l to fulfill the request, in which case the weight is ∞). To this end, the algorithm uses a dictionary $\mathcal{B}$ which tracks the best-known links used to reach nodes from e_i. Therefore, we say that l.weight $= \mathcal{B}[l$.src$]$.weight $+ 1/(T_B(l) - A_B(l))$, where $\mathcal{B}[l$.src$]$ is the best-known link used to reach l.src from e_i (line 31). Using this weight relies on the fact that links are stored on a k-ary min heap $\mathcal{H}$, which then keeps the link with the lowest weight at the top. This implies that any link l at the top of the heap can be used to reach $l.dst$ with the lowest possible total cost from e_i (assuming l.src $\neq l.dst$), meaning l is suitable to be added to the shortest-path tree $\mathcal{P}$. This link weight metric also results in the selection of paths that tend to be short and have a high amount of available bandwidth. Furthermore, $\mathcal{H}$ is continually updated during algorithm execution with the help of the best-known link dictionary $\mathcal{B}$. Specifically, $\mathcal{B}[n]$ returns the best-known link to reach node n. If $n \notin \mathcal{B}$ (n has not been visited already), then the link used to reach n is pushed onto $\mathcal{H}$ (line 34). Otherwise, n has already been visited and the RAA checks if the new link used to reach n has a lower weight than the best-known link $\mathcal{B}[n]$. If it does, then a modified decrease-key operation is performed on $\mathcal{H}$ which replaces the link $\mathcal{B}[n]$ with the cheaper new link (line 38). The new link is then shifted upward in the heap. This process repeats as the algorithm continues to visit nodes using different paths, eventually shifting the best links to the top of $\mathcal{H}$ and choosing to include them in the shortest-path tree dictionary $\mathcal{P}$ (line 28).

Once Dijkstra's algorithm is finished, dictionary $\mathcal{P}$ contains the shortest-path tree. To be more precise, $\mathcal{P}[n]$ returns the link attached to n facing e_i that is included in the shortest path from e_i to n, as well as its weight and both nodes at the endpoints of the link. To this end, $\mathcal{P}$ can be used to traverse and gather information on the shortest path between e_i and any other device in the network.

$\mathcal{P}$ is then used in the subsequent post-processing step. First, $\mathcal{P}[f_j]$.weight is checked for all $f_j \in \mathcal{F}'$ and a fog device $f_{\min} \in \mathcal{F}'$, where $\mathcal{P}[f_{\min}]$.weight $= \min(\mathcal{P}[f_j]$.weight$)\ \forall f_j \in \mathcal{F}'$ is selected to fulfill the service request (line 45). If $\mathcal{P}[f_{\min}]$.weight $= \infty$, then no paths with sufficient bandwidth between e_i and any request servicers exist, and a failure response is returned to e_i as a result (line 46). Otherwise, the path between fog device $f_{\min}$ and end device e_i has a sufficient amount of bandwidth and f_{min} is chosen to fulfill the request from e_i.

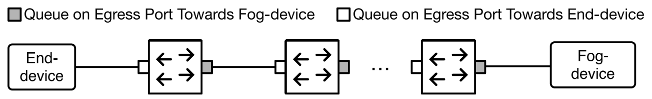
▨ Queue on Egress Port Towards Fog-device
□ Queue on Egress Port Towards End-device
End-device
...
Fog-device

Fig. 2. Enforcing bandwidth reservation using rate-limited queues. For each path reservation, rate-limited packet queues are created and attached to QoS configurations located on the egress ports toward the fog device as well as those toward the end device. Then, flow table entries are pushed via OpenFlow to enqueue traffic traveling from the end device to the fog device, and *vice versa*, on these queues.

The next step is to allocate network resources along the identified path between e_i and $f_{\min}$. The nodes along this path are accessed by traversing dictionary $\mathcal{P}$. Network resource allocation begins with the creation of *rate-limited* queues on each switch along this path. The ResourceManager accomplishes this by making a call to the TopologyManager function `create_queue()`, which leverages the OVSDB management protocol to create and configure the queues (line 51). The rate-limit is specified in the queue configuration data and is equal to $R_B(e_i)$. Once created, these queues are placed on QoS entries (created on startup of the FDK by the TopologyManager) using a similar TopologyManager function `place_queue_on_qos()` (line 52). These QoS entries map to switch ports connected to the network links along this path, effectively resulting in each port having a set of packet queues that limit egress traffic.

In addition to queues, flows must also be created to ensure that traffic is directed along the identified path between e_i and $f_{\min}$ and that packets exchanged between the two devices are placed on the proper queues within each switch. Therefore, as the ResourceManager installs packet queues on each switch, it also uses OpenFlow to redirect traffic along the identified path and to the appropriate queues along that path by leveraging the FlowManager flow-creation APIs (line 53). Each OpenFlow flow specifies a set of actions for the reserved path. Therefore, on each switch along the path the FDK uses one OpenFlow flow that specifies multiple actions: one for redirecting traffic to the desired port (therefore reserving a one-way path for communications between e_i and $f_{\min}$), and another to place packets on the appropriate queue for that port. Similarly, for communications in the opposite direction from $f_{\min}$ to e_i, another packet queue and OpenFlow flow is installed on each switch. Therefore, the overhead of enforcing a path and reserving communication bandwidth for one service involves the creation of two packet queues and two OpenFlow flows on each switch along the identified path. Fig. 2 depicts the creation of rate-limited queues along a path to ensure network bandwidth allocation in both directions. Finally, because the FDK never over-allocates resources, the rate-limiting of bandwidth effectively results in the *allocation* of bandwidth.

The flows installed on switches match packets based on source IP address, destination IP address, source or destination port number (depending on the traffic direction), and protocol type. For communications from e_i to $f_{\min}$, the source IP address is e_i's IP address, the destination IP address is $f_{\min}$'s IP address, and the destination port is a proxy port on

$f_{\min}$ assigned to the containerized service. The protocol type specifies the transport layer protocol used by the application. The transport layer protocols supported are UDP, TCP, SCTP, and any user-space protocol that relies on these protocols. For example, QUIC [37], [38] is a widely used user-space protocol that is implemented on top of UDP, and is therefore supported by the FDK.

Finally, a success response containing $f_{\min}$'s IP address and the proxy port (if the end device has not asked for a particular port number) is returned to the service request server (line 50), which then remotely instantiates a container on $f_{\min}$ using Docker swarm. The success response is then forwarded to the end device as well. At this point, all computational and networking resources have been allocated, and once e_i receives the success response message, it can begin communicating with the newly created containerized service running on $f_{\min}$.

There are multiple mechanisms available to direct packets to the appropriate container when they arrive at $f_{\min}$. The first mechanism is to dedicate a unique proxy port on the fog device to each service it is hosting. To this end, for each containerized service, the FDK finds a unique port number that has not been used on the fog device hosting the container. The FDK also allows end devices to specify their desired destination port number when making requests. However, without adding additional capabilities, this mechanism does not allow two or more end devices to request the same port number on a fog device. To address this issue, the fog device demultiplexes (using reverse proxy or OVS) the received packets to different containers based on their source IP address. Therefore, once a service request is fulfilled, the FDK only needs to return the IP address of the identified fog device to the end device. An alternative approach for supporting multiple containers using the same port numbers on the same fog device is to assign each container an IP address in the same subnet as that of the fog device. In this case, the IP address assigned to the container is returned to the end device, instead of the IP address of the fog device. Also, the container's IP address is used to configure the flow tables on switches along the communication path. This approach, however, is not officially supported by Docker due to its security issues. Specifically, this approach does not allow the protection of containers from the outside world and from each other. In contrast, using a proxy port requires ingress access to be explicitly granted, which offers higher security. Therefore, although both mechanisms are supported by the FDK, in this article, we particularly focused on the former due to its higher security and widespread adoption [39].

As seen throughout this section, applications must be adapted to the FDK in order to benefit from fog resources. Therefore, end devices must be programmed to issue service requests so that these resources may be allocated. However, this may not be possible with commercial, nonopen-source applications running on the end devices. A simple solution is to use a middleware that issues service requests on behalf of the application. Also, since the middleware can translate the destination port number of packets originating from end devices, a unique port number can be assigned to each request, and therefore there is no need to use a demultiplexing tool on the fog devices to deliver incoming packets to the appropriate container. It is also worth noting that the middleware does not need to be implemented on the end devices. As an example, consider a gateway node (such as a smartphone or an IoT gateway) collecting data from multiple sensing devices. The gateway can then request for resources on behalf of these devices, and therefore there is no need to modify the software stack of the sensing devices.

To summarize, consider a scenario where multiple end devices communicate with multiple containers that run on a single fog device and listen on the same port. In this case, source IP address is the 5-tuple's element that is used to classify these flows by the switches as well as the fog device. Alternatively, if a gateway that includes a middleware is used to issue requests on behalf of multiple end devices, since the source IP address of all the requests generated by the gateway are the same, the FDK generates a unique port number assigned to each service. In this case, port number is the 5-tuple's element that is used to classify these flows by the switches as well as the fog device. Therefore, the FDK offers a robust flow classification mechanism on switches and fog devices as a part of its resource management capabilities. With this mechanism, end devices are provided with the capability to make multiple service requests in parallel. This implies that any end device may request an arbitrary amount of services (as long as sufficient resources exist), and therefore run an arbitrary number of fog applications.

As the ResourceManager continues allocating resources over time, it keeps track of all allocated resources. Once an end device decides to terminate a service, it issues a shutdown request to the *shutdown request server*, which then runs the resource deallocation algorithm (RDA). The RDA identifies and releases the resources allocated for the corresponding service: OpenFlow flows along the reserved path are deleted, network bandwidth is deallocated by deleting the appropriate packet queues, and the containerized service in the fog device is shutdown.

V. Evaluation

In this section, we first verify the correctness and performance of the FDK using sample applications running on a physical testbed. We then present a simulation-based scalability analysis of the FDK.

A. Verification and Evaluation Using Physical Testbed

In this section, we verify the correctness and performance of the FDK using a physical testbed running various applications.

Testbed: Fig. 3 shows our testbed, which includes five OpenFlow switches, four fog devices, and eight end devices. This testbed implements the network presented in Fig. 4. Each end device is a Raspberry Pi Model 3 B+ (running Raspbian Linux) which is connected to a switch via a 1-Gb/s cable. The machine hosting the four fog devices includes a 4-port Intel 82580 NIC, where each fog device is a VM associated with a physical port. Another machine includes five 4-port Intel 82580 NICs as well as a 2-port NIC to build the five OpenFlow

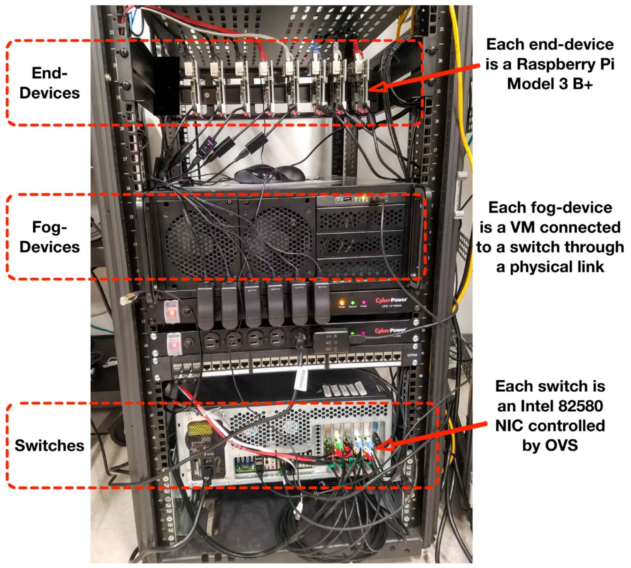
End-Devices
Each end-device is a Raspberry Pi Model 3 B+
Fog-Devices
Each fog-device is a VM connected to a switch through a physical link
Switches
Each switch is an Intel 82580 NIC controlled by OVS

Fig. 3. Physical testbed used to implement the topology depicted in Fig. 4. End devices, switches, and fog devices are connected through physical links.

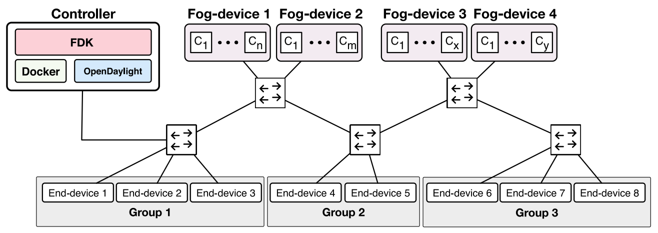

Fig. 4. Network topology used for the development and testing of the FDK. C_i represents a container running on a fog device.

switches. The 2-port NIC is paired with one of the aforementioned 4-port NICs to build a 6-port switch which is connected via a 1 Gb/s cable to the controller. Both machines include two 16-core Intel Xeon CPUs and 64 GB RAM. Each fog device and OpenFlow switch uses Ubuntu Server 18.10 and leverages 4 CPU cores and 8 GB of RAM. The OpenFlow switches run OVS 2.10.0 and support both OpenFlow 1.3 and OVSDB. Docker daemons run on each fog device and are configured to listen for remote TCP connections from the controller. The controller (including the FDK) is hosted on an external server.

We partitioned the end devices into three groups. Referring back to Fig. 4, we placed end devices 1, 2, and 3 into group 1, end devices 4 and 5 into group 2, and end devices 6, 7, and 8 into group 3. This grouping helps us identify the effect of the FDK on network overhead, which may vary depending on the location of the end devices. For example, assume that all the end devices issue service requests concurrently. The OpenFlow switch connected to the devices in group 1, which is the switch closest to the controller, would be placed under higher stress compared to those switches further from the controller, such as the switch connected to group 3. In this case, all service and shutdown request messages, OpenFlow messages, OVSDB messages, Docker swarm container instantiation messages, etc., pass through the switch connected to group 1. At the same time, only a fraction of these messages pass through the switch connected to group 3. We created the three groups in an attempt to capture the effect of these variations.

Applications: We evaluate the application development capabilities of the FDK by creating a set of sample applications. The first application, which includes an iperf3 server and an iperf3 client, is called *iperf-app* and enables an end device (client) to communicate through TCP with a containerized service on a fog device (server). To develop *iperf-app*, we first created a Python script that hosts an iperf3 server using the iperf-python library [40]. We then packaged this script into a Docker image and modified the server script to communicate all bandwidth readings to a background process running on each fog device. This process receives and saves the readings. On the end devices, we created another Python application that issues a service request to instantiate the aforementioned Docker image as a container, starts the client that streams data to the server running in the container, and then issues a shutdown request once the client terminates. The second application developed is *sleep-app*, which sends a service request, sleeps for a particular duration, and then sends a shutdown request. These applications are used to analyze the impact of service requests and varying levels of bandwidth utilization on the FDK's ability to service those requests. The third application developed is an object detection application named *detection-app*. The application streams image data from end devices to the services in the fog devices, which run object detection algorithms to identify different objects found in images. The transport protocol used by this application is QUIC. A real-world example of this application is an object classification and packaging system. Another application is a real-time surveillance system supporting facial recognition.

Verification: In order to confirm the functionality of the FDK, and before running any tests, we issued service requests to the FDK from the end devices and verified that resources are allocated properly. To this end, we made temporary modifications to the fog-side Docker images that would consume as many resources as possible and then confirmed that the containers instantiated from these images did not exceed the resources allocated to them. For example, we modified *iperf-app* in one test to spin up an infinite `while` loop script that consumed all processing resources. Then, by using performance monitoring tools, such as `top`, we confirmed that the container did not exceed the resource allocations requested by the end device. Similarly, we confirmed that network resources were appropriately allocated using *iperf-app*, which revealed that bandwidth allocations were not exceeded. Finally, we used *detection-app* to represent a real-world scenario, where configured end devices randomly wake-up, issue a service request, and then capture and stream images to the fog devices running object recognition algorithms. Each service request specifies a desired bandwidth allocation of 40 Mb/s. Each end device ceases its image streaming and sends a shutdown request after about 7 s into its streaming period. We performed similar verification steps to ensure that resources were all allocated and deallocated properly. Fig. 5 shows the total bandwidth of streaming data received by the fog devices. To generate this figure, all fog devices were configured to be time-synchronized, and

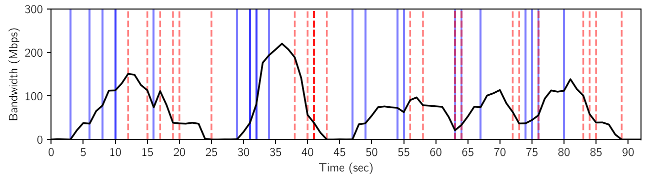

Fig. 5. Overall bandwidth of the data received by fog devices corresponding to the images captured and sent by end devices via *detection-app*. Blue (solid) and red (dashed) bars denote service requests and shutdown requests made by end devices, respectively. Bars that are less transparent indicate a greater amount of service or shutdown requests made during a particular second. This figure shows the dynamics of allocating and deallocating resources by the FDK when end devices randomly issue service and shutdown requests.

each container was configured to record the number of bytes received per second through its network interface.

Given that these applications use a variety of transmission rates, transport layer protocols, and randomized request patterns, the operation of the FDK was carefully verified before proceeding with performance evaluation tests. In the rest of this section, we present performance evaluation of the FDK.

1) Test 1 (Resource Allocation and Deallocation): The goal of test 1 is to characterize the computational and communication overhead of the FDK. This is accomplished by running applications across all end devices and recording the runtimes of various operations under different circumstances. We track the duration of key operations, including resource allocation (RAA), resource deallocation (RDA), service request fulfillment, and shutdown request fulfillment. Service request fulfillment duration refers to the total duration between the time an end device sends a service request to the FDK and the time the end device receives a success response from the FDK. Similarly, shutdown request fulfillment duration refers to the total duration between sending a shutdown request and the reception of confirmation. For this experiment, we ran *sleep-app* across all eight end devices in the topology and measured the duration of the aforementioned performance parameters. We repeated this experiment 250 times for a total of 2000 *sleep-app* runs, and ran two different versions of this test, bringing the number to 4000. These different test versions are *test 1a* and *test 1b*, as follows.

Test 1a: In this test, the end devices *sequentially* run *sleep-app*. For example, end device 1 issues a service request, sleeps for 3 s after receiving service, and then issues a shutdown request. After completion, the rest of the end devices perform the same operation sequentially. Fig. 6 presents the results of test 1a. The duration of various operations are averaged out among the end devices of each group and are then displayed as empirical cumulative distribution function (ECDF) graphs. As seen in Fig. 6, more than 95% of all operations completed within 0.33 s across all groups. In addition, resource allocation times and service request fulfillment times are nearly identical, as are the resource deallocation times and shutdown request fulfillment times. This means that resource allocation is the main source of overhead in the process of fulfilling service requests, and that resource deallocation is the main source of overhead in the process of fulfilling shutdown requests. Also, operations performed for devices in group 1 tend to finish slightly faster than those for group 2, which finish faster than those for group 3. This is caused by the shorter queuing and packet processing delays along the path to the controller with a fewer number of switches. However, the difference in timing is on the order of a few milliseconds.

Test 1b: In this test, the end devices *concurrently* run *sleep-app*. In this case, all end devices issue a service request to the FDK at the same time, sleep for 3 s upon receiving a successful response, and then send a shutdown request. Fig. 7 presents the results of test 1b. The results presented in Fig. 7(a) and (b) are nearly identical to the corresponding Fig. 6(a) and (b) from test 1a, with 95% of these operations completing within 0.28 s across all groups. However, the results for service request fulfillment times in test 1b, shown in Fig. 7(c), look considerably different compared to the corresponding Fig. 6(c) from test 1a. Here, we can observe greater variations in the results, with group 1, group 2, and group 3 showing median service request fulfillment durations of 0.72, 1.06, and 1.71 s, respectively. Also, there is far less variation in the results for shutdown request fulfillment times, as Fig. 7(d) shows. In this regard, group 1, group 2, and group 3 show median shutdown request fulfillment durations of 0.23, 0.24, and 0.25 s, respectively.

Since the service fulfillment process accesses and modifies various shared data structures such as the topology object representing the current state of the network, the entire process is guarded by a mutex. This means that the FDK queues concurrent service requests and handles them sequentially. This effect can be seen in Fig. 7(c). Because the end devices in group 1 are closer to the controller than those in groups 2 and 3, service requests from these devices sit closer to the front of the queue than the requests arriving later from groups 2 and 3. Therefore, groups 2 and 3 experience slower service request fulfillment times compared to group 1. Similarly, the process of resource deallocation is also guarded by a mutex, meaning that concurrent shutdown requests are handled sequentially as well. However, because the service requests are fulfilled sequentially, the sleep durations and subsequent shutdown requests made by each *sleep-app* instance become desynchronized and happen sequentially. As a result, we see a much smaller impact on shutdown request fulfillment times in comparison to service request fulfillment times in test 1b.

2) Test 2 (Bandwidth Guarantee): In test 2, we evaluate the overhead of the FDK on the network. Specifically, we investigate if the FDK compromises bandwidth allocations (by reducing transmission speeds) for running fog applications. We chose one end device from each group to run *iperf-app* for 90 s with a 300 Mb/s bandwidth allocation. This is the maximum transmission rate of the Raspberry Pi Model 3 B+. Also, after subtracting transmission overheads, such as packet headers, the actual data transmission rate supported is around 280 Mb/s. Using *iperf-app*, an end device continuously streams data to a container for 90 s. Then, at 30 and 60 s into the 90-s transmission, all seven other end devices in the topology run *sleep-app* for 1 s. This results in a group of service requests, shutdown requests, OpenFlow messages, OVSDB messages, and Docker swarm container instantiation messages flowing through the network. We ran this experiment 100 times for the chosen end device and repeated it for the other two chosen end devices

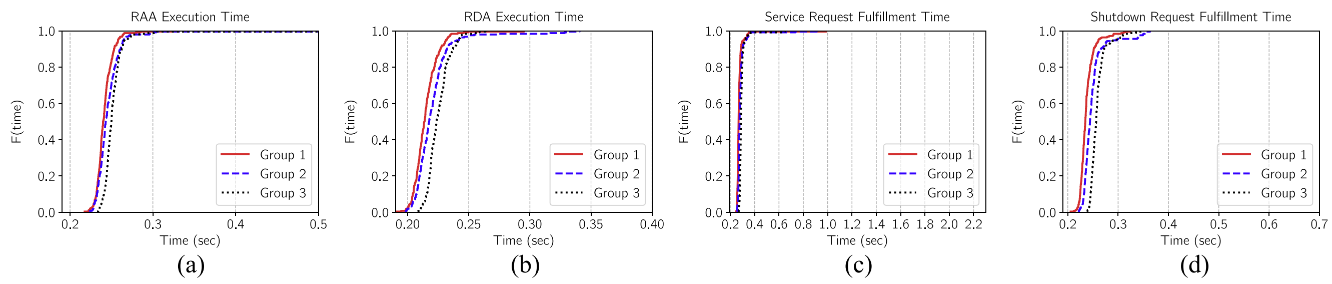

Fig. 6. ECDF graphs for test 1a. In this test, end devices issue service requests *sequentially*. Groups closer to the controller (and therefore FDK) complete all of their operations slightly faster than those further from the controller.

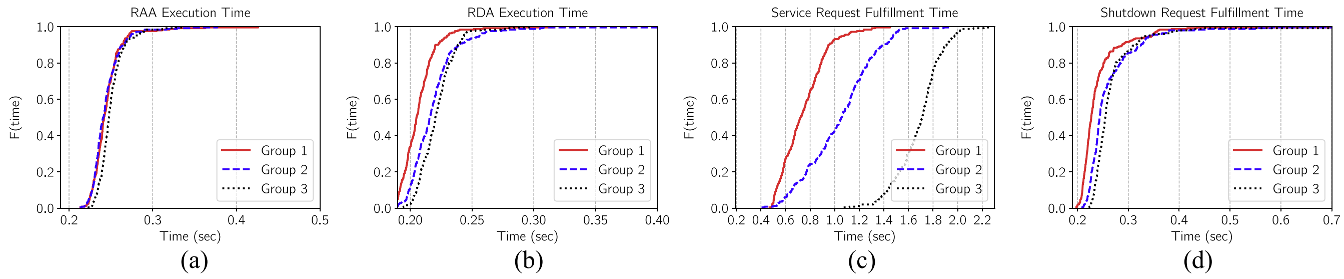

Fig. 7. ECDF graphs for test 1b. In this test, end devices issue service requests *concurrently*. Groups closer to the controller experience significantly faster service request fulfillment times compared to those further from the controller. This is because the FDK processes requests *sequentially*.

from the other two groups, for a total of 300 *iperf-app* runs and 4200 *sleep-app* runs. Finally, we used two separate versions of this test and analyzed their impact on network congestion and the transmission bandwidth of *iperf-app*. In the end, 600 *iperf-app* runs and 8400 *sleep-app* runs were performed. The two test cases, called test 2a and test 2b, are outlined in detail as follows.

Test 2a: Here, the *sleep-app* runs occur *sequentially* with a 2-s gap in between each run. Fig. 8 shows the median value, as well as the upper and lower quartile values, for all 90 bandwidth readings of end devices 1, 4, and 6. Although additional messages are flowing through the network at around 30–45 s and 60–75 s, the transmission speed of *iperf-app* is not affected, indicating that the bandwidth allocations are not compromised by the overhead incurred by the other *sleep-app* runs performed during this time.

Test 2b: This test is identical to test 2a, except that the *sleep-app* runs occur after 30 and 60 s into transmission are executed *concurrently*. Fig. 9 presents the results. Similar to the results of test 2a, we see that there is essentially no drop or variation in bandwidth.

3) Test 3 (Multiple Bandwidth Guarantees): In this test, we evaluate the effect of a large amount of concurrent requests on the service and transmission speeds of multiple fog applications running in parallel. We subject the hardware to a stress test to measure how the FDK operates under large volumes of requests and to see if bandwidth guarantees can be reliably fulfilled in a highly congested network.

For this test, we use end devices 1 through 7 to run *iperf-app* concurrently, and a bandwidth reading is collected per second for 90 s. Then, at 30 and 60 s into the 90-s transmission, end device 8 executes 15 concurrent runs of *sleep-app* at the same time. This process is repeated 100 times, meaning that 700 *iperf-app* runs and 3000 *sleep-app* runs are performed in

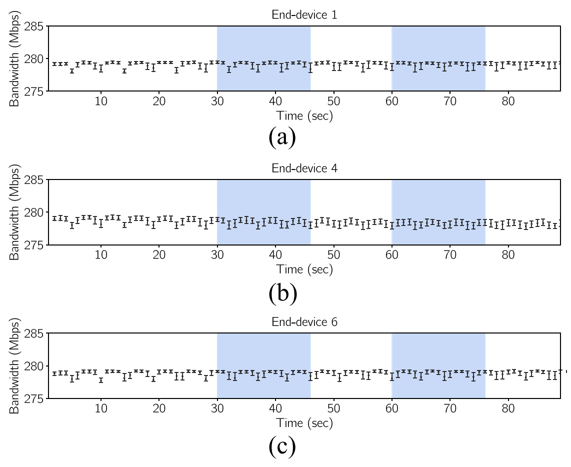

Fig. 8. Bandwidth readings for end devices 1, 4, and 6 throughout test 2a (300-Mb/s allocation). The bars show the sequential execution of *sleep-app* by seven end devices. These results show that there is no additional variation in bandwidth for running fog applications in the presence of *sequential* service requests made to the FDK by other end devices.

total. Finally, three different variations of test 3 are executed, where different bandwidth allocations of 100 Mb/s (test 3a), 200 Mb/s (test 3b), and 300 Mb/s (test 3c) are reserved for each *iperf-app* instance, bringing the total number of *iperf-app* and *sleep-app* runs to 2100 and 9000, respectively.

Once the tests completed, we calculated the average of each 1-s bandwidth reading across the end devices in the three groups. For example, in the case of group 1, we initially had three bandwidth data sets consisting of 100 runs each (one for each of end devices 1, 2, and 3), where each run consists of 90 bandwidth readings. We then took the average of each bandwidth reading (per second) across every run to create a single data set of 100 runs. Similarly, the same idea applies to the devices and data for groups 2 and 3. Note that we did

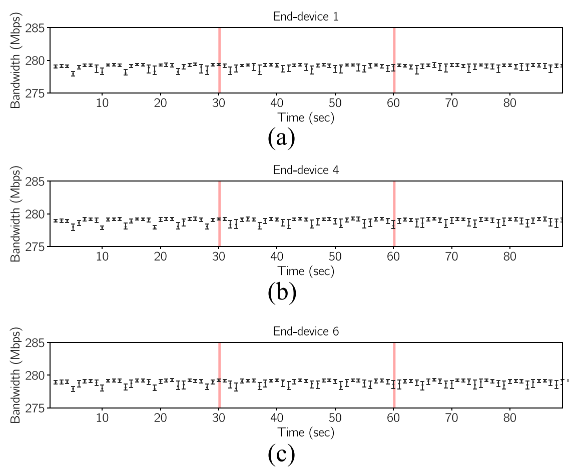

Fig. 9. Bandwidth readings for end devices 1, 4, and 6 during test 2b (300-Mb/s allocation). The vertical lines show the instances of the seven end devices start *sleep-app* concurrently. These results show that there is no additional variation in bandwidth in the presence of *concurrent* service requests made to the FDK by other end devices.

not include end device 8 in group 3 for this test because it was performing 15 concurrent *sleep-app* runs and would have experienced a degradation in performance if it were to run *iperf-app* as well. This is due to the limited networking and processing capabilities of the Raspberry Pi.

Fig. 10 shows the results for test 3. Here, we formatted the results similar to those of test 2, where markers for the median value, upper quartile value, and lower quartile value are displayed for each (averaged) bandwidth reading of every run. Each subfigure represents all of the data collected for an entire group. These results demonstrate less than 1-Mb/s variations for 100- and 200-Mb/s allocations, and less than 5-Mb/s variations for 300-Mb/s allocations. More specifically, Fig. 10 shows that the actual bandwidth readings are just below the allocated amounts at all times, regardless of traffic stress on the switches. As previously mentioned, this is because of transmission overheads (such as packet headers) and the limited processing power of the Raspberry Pi boards.

In the case of the 300-Mb/s *iperf-app* runs, there are more variations in the bandwidth readings than the 200- and 100-Mb/s runs. However, these variations do not correspond to the additional messages flowing throughout the network at 30 and 60 s into the 90-s *iperf-app* transmission. We believe that this is caused by the processing and queuing delays of the OVS kernel path. Similar observations have been made in [29], which confirms that enhancing the switching rate and reducing variations can be achieved by using OVS DPDK and certain GRUB configurations. We leave these enhancements as future work.

B. Scalability Analysis

A closer look into the operation of the FDK reveals that the five delay components of fulfilling a request are: i) sending a request from an end device to the controller; ii) execution of the RAA to identify a fog device and a communication path by the controller; iii) configuration-related communications between the controller and switches and fog devices; iv) execution of configuration commands on the switches and the fog device; and v) sending a reply back to the end device to confirm the reservation. Therefore, we can categorize these delays into three groups: *communication delay*: items i), iii), and v); *processing delay of controller*: item ii); and *processing delay of switches and fog devices*: item iv).

The communication delay and processing delay of the controller are affected by network size, which is defined by the number of nodes and the number of links connecting them. In addition, the communication delay is affected by other factors, such as queuing delay and link speed. The processing delay of configuring switches and fog devices depends on the hardware and software capabilities of these devices. In particular, the delay of path reservation on a switch depends on the delays of updating the forwarding table and queue allocation. Similarly, the delay of fog device configuration (container instantiation) depends on the processing capabilities of the fog device.

Since fog device and switch configuration delays depend on the hardware and software characteristics of these components, in this section, we neglect these delays and instead focus on the impact of controller processing delay and communication delay on resource allocation. To evaluate the performance of the FDK versus network size, we developed a simulation tool using the OMNet++ framework [41]. Fig. 11(a) and (b) present the topologies used, which are inspired by leaf-spine and fat-tree architectures [42], respectively. Topology (a) includes two levels of switches, where each level 1 switch is connected to 1/3 of the (nearest) level 2 switches. Note that, in order to increase the number of hops between end devices and fog devices, we did not connect each level 1 switch to all level 2 switches. Topology (b) is a tree-like topology that includes three levels of switches, where each level 1 switch is connected to one level 2 switch, and each level 2 switch is connected to one level 3 switch. The controller is connected to the middle switch in level 2 in topology (a) and level 3 in topology (b). In both topologies, the switches of the highest level are horizontally connected.

Fig. 12 shows the RAA execution delay on a single core of a Xeon E5 3 GHz processor. The time required to evaluate the allocation of resources across all fog devices to a given end device is computed, and each point represents the median of these results for all the end devices. In other words, referring back to Algorithm 1, we assume that $\mathcal{F}' = \mathcal{F}$, meaning that all fog devices are eligible to run the service requested by the end device. Error bars show higher and lower quartiles. As discussed in Section IV-C, the time complexity of the RAA is $O(m \log_k n)$. Also, for a given topology, increasing the number of fog devices per switch increases the execution time because a higher number of paths must be evaluated whenever a service request arrives. Increasing the number of fog devices from 10 to 20 causes approximately 120% and 64% increase in execution time in Fig. 12(a) and (b), respectively. Comparing Fig. 12(a) and (b) shows that the execution time of topology (b) is about 22%, 39%, and 54% lower than that of topology (a) when the number of fog devices per highest level switch are 5, 10, and 20, respectively. This reduction is because of the fewer number of paths in topology (b). For example, although the number of nodes in topology (b)'s configuration (25, 12, 6) is higher than that of topology (a)'s configuration (25, 12),

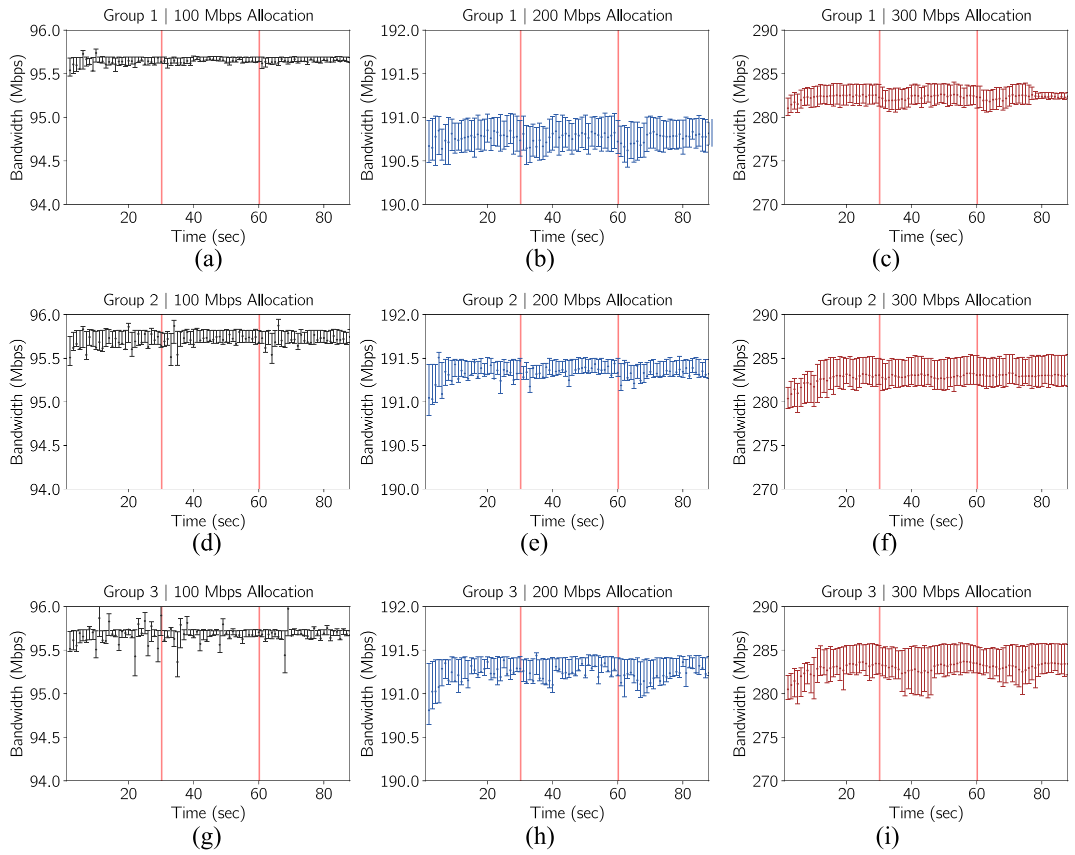

Fig. 10. Actual bandwidth readings for tests 3a, 3b, and 3c for each group. End devices 1 through 7 run *iperf-app*, and end device 8 performs 15 concurrent runs of *sleep-app* at 30 and 60 s (as indicated by the vertical lines) into the 90-s *iperf-app* transmissions. Even under network congestion and stress during these times, the results show that bandwidth allocations are enforced and no additional variation is observable.

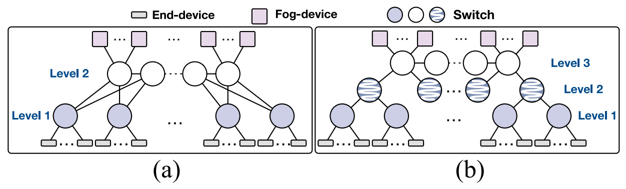

Fig. 11. Two topologies used for scalability evaluation. These two topologies are referred to as “topology (a)” and “topology (b)” in the text.

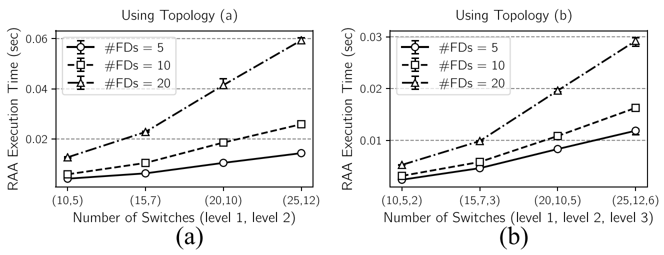

Fig. 12. Execution time of the RAA (excluding switch and fog device configuration delay) versus network size and number of fog devices. (a) and (b) Present the results for topology (a) and (b) of Fig. 11, respectively. #FDs refers to the number of fog devices per level 2 switch in topology (a) and level 3 switch in topology (b). The values in each parenthesis on the x-axis refer, from left to right, to the number of level 1, level 2, and level 3 switches.

there is a lower number of paths between each end device and fog device in the former topology.

The next set of results presents the communication delay of resource allocation. Figs. 13 and 14 show the median communication delay during resource allocation versus the number of hops between end devices and fog devices for all the possible allocations of fog devices to end devices. Each configuration is presented as a tuple (x, y), where x refers to the bandwidth *used* by all data flows (end device to/from fog device) and y refers to the bandwidth *allocated* to the exchange of control flows [items i), iii), and v)] between nodes and the controller.

Both Figs. 13 and 14 exhibit the impact of the number of hops and background traffic on allocation delay. The figures show that a higher number of hops increases the number of switches that must be configured along the reservation path. More specifically, they show that doubling the number of hops doubles the allocation time. A higher utilization level of links by data flows causes a higher queuing delay on the egress ports of switches. Queuing delay affects all communication delay components, including items i), iii), and v). For example, a 5× increase in the bandwidth allocated to data flows results in about 380% higher allocation delay. In cases of high bandwidth utilization by data flows, these results show that doubling the bandwidth allocated to control flows can cut the allocation delay by half. However, this introduces a tradeoff between resource allocation delay and the communication resources available for data flows.

Figs. 13 and 14 also reveal that increasing the number of end devices results in a higher allocation delay. Specifically,

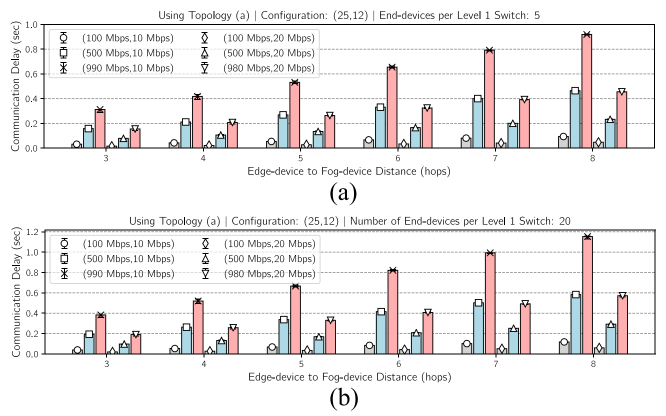

Fig. 13. Communication delay of resource allocation versus end device to fog device distance (hops) for topology (a) presented in Fig. 11.

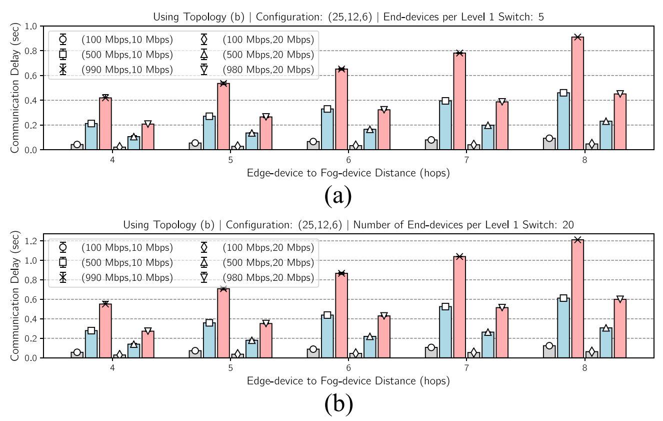

Fig. 14. Communication delay of resource allocation versus end device to fog device distance (hops) for topology (b) presented in Fig. 11.

when increasing the number of end devices from 5 to 20, communication delay is increased by 25% and 32% in topology (a) and (b), respectively. The cause behind this increase is a higher communication delay (caused by OVSDB) between the controller and switches that grows as the number of flows and queues on each of the switches increases. For example, the allocation of each queue on a switch inflates the number of bytes exchanged during the resource allocation process as follows: 55 extra bytes are sent from the controller to the switch, and 1000 extra bytes are sent from the switch to the controller.

When discussing Fig. 12, we highlighted that topology (b) reduces execution time by about 50%. In contrast, Figs. 13(b) and 14(b) demonstrate that there is a higher communication delay of path reservation in topology (b). This increase is about 5% when the number of end devices is 20, and it is further increased for a larger number of end devices (not shown in the results). This is because the lower number of communication paths between the controller and switches in topology (b) causes a higher queuing delay that intensifies delay component (iii). This is also the reason behind the large increase in communication delay versus the number of end devices in topology (b) [when comparing Figs. 13(b) and 14(b)]. In summary, by putting together the results of Figs. 12 and 14, when the number of level 1 and level 2 switches are 25 and 12, respectively, topology (b) results in 30 ms lower RAA execution delay and 22 ms higher communication delay. It should be noted that, if the number of end devices surpasses 20, the RAA execution delay would be the same but the communication delay would further increase.

VI. Future Work

In this section, we present potential future works to extend the FDK.

With regards to network resource allocation, we plan to include transmission delay guarantees by adopting approaches similar to [23]. Furthermore, the FDK does not support resource negotiation with end devices: This means that if the amount of resources requested by an end device exceeds the resources available in the system, the end device simply receives a failure response message from the FDK and cannot determine which resource demands should be reduced or by how much. We plan on including available resource information in responses to end devices to promote flexible and more efficient service requests. Moreover, it is not immediately apparent how an end device can calculate what amount of resources is appropriate to request. For example, determining the actual amount of processing and memory capabilities required to execute a fog application in a timely and efficient manner depends on various factors, such as data processing algorithms, data generation rate, and the sensitivity of an application to delays. As such, a proven, efficient solution to this problem is not immediately apparent and will be the key to enhancing interactions and establishing a greater synergy between end devices and fog devices.

In terms of scalability, it is essential to partition large-scale fog-systems with stringent resource allocation deadlines into separately controlled regions. Specifically, we propose to use a local controller in each region. These local controllers are provided with preallocated resources by the main controller, and these resources can be allocated to end devices immediately. Each local controller can also request (from the main controller) for more resources based on network dynamics. In addition, instead of using dedicated boxes, local controllers could be implemented in some switches. An alternative approach to reducing the execution delay of RAA is to create multiple logical overlay networks based on link delays and bandwidth. Then, for example, if an end device is requesting 100 Mb/s, only the overlay networks with links satisfying the requested amount of bandwidth will be considered.

As mentioned earlier, we assign a separate rate-limited queue to each egress port along the path identified for a reservation. For systems, including a large number of reservations between end devices and fog devices, the use of software switches, such as OVS allows the deployment of a higher number of flows and queues in comparison to hardware switches. In the case of software switches, OVS's mega-flow cache can be employed to aggregate flows. To this end, instead of flow matching on 5-tuples, multiple flows (sharing properties, such as destination fog device or egress port) could be aggregated [43]. However, to efficiently benefit from this feature, RAA (Algorithm 1) must be revised as well.

The FDK opens up vast possibilities for the research and development of fog systems in areas, such as image classification, medical monitoring, and industrial monitoring and process control [44], [45]. In addition to the enhancement and evaluation of the system's building blocks (e.g., RAAs and live container migration), further experimentation can be performed using the FDK to identify the shortcomings of existing solutions as well as developing production-ready solutions.

VII. Conclusion

In this article, we proposed the FDK: a platform for the development and management of fog systems. The FDK provides a comprehensive resource allocation scheme and stands ahead of other alternatives by enabling both computational and networking resource allocation. Also, the FDK is application-independent and offers a significantly shorter and simplified development cycle for fog-based applications. In addition to supporting physical, production-grade environments, the FDK significantly reduces development costs by supporting the use of emulation tools as well. Therefore, the FDK offers application portability between physical and emulated environments. These features make the FDK a valuable tool in prototyping and developing any fog system, as they can be created and tested virtually on personal computers and then be easily ported to a physical topology. These capabilities differentiate the FDK from existing simulation platforms. By allowing end devices to request an arbitrary amount of resources and services from fog devices, the FDK enables the development of large and complex fog systems at essentially no cost, while removing the complexity of resource allocation away from developers.

References

[1] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the Internet of Things," in *Proc. 1st ed. Workshop Mobile Cloud Comput. (MCC)*, 2012, pp. 13–16.

[2] Y. Cao, P. Hou, D. Brown, J. Wang, and S. Chen, "Distributed analytics and edge intelligence: Pervasive health monitoring at the era of fog computing," in *Proc. ACM Workshop Mobile Big Data*, 2015, pp. 43–48.

[3] M. Chiang and T. Zhang, "Fog and IoT: An overview of research opportunities," *IEEE Internet Things J.*, vol. 3, no. 6, pp. 854–864, Dec. 2016.

[4] W. Shi and S. Dustdar, "The promise of edge computing," *Computer*, vol. 49, no. 5, pp. 78–81, 2016.

[5] I. Amirtharaj, T. Groot, and B. Dezfouli, "Profiling and improving the duty-cycling performance of Linux-based IoT devices," *J. Ambient Intell. Humanized Comput.*, pp. 1–29, Jan. 2019. [Online]. Available: https://link.springer.com/article/10.1007/s12652-019-01197-2

[6] N. Wang, B. Varghese, M. Matthaiou, and D. S. Nikolopoulos, "ENORM: A framework for edge node resource management," *IEEE Trans. Services Comput.*, to be published.

[7] L. Yin, J. Luo, and H. Luo, "Tasks scheduling and resource allocation in fog computing based on containers for smart manufacture," *IEEE Trans. Ind. Informat.*, vol. 14, no. 10, pp. 4712–4721, Oct. 2018.

[8] O. Skarlat, S. Schulte, M. Borkowski, and P. Leitner, "Resource provisioning for IoT services in the fog," in *Proc. IEEE 9th Int. Conf. Service Orient. Comput. Appl. (SOCA)*, 2016, pp. 32–39.

[9] N. Ansari and X. Sun, "Mobile edge computing empowers Internet of Things," *IEICE Trans. Commun.*, vol. 101, no. 3, pp. 604–619, 2018.

[10] J. Son, A. V. Dastjerdi, R. N. Calheiros, X. Ji, Y. Yoon, and R. Buyya, "CloudSimSDN: Modeling and simulation of software-defined cloud data centers," in *Proc. 15th IEEE/ACM Int. Symp. Clust. Cloud Grid Comput.*, 2015, pp. 475–484.

[11] S. F. Piraghaj, A. V. Dastjerdi, R. N. Calheiros, and R. Buyya, "ContainerCloudSim: An environment for modeling and simulation of containers in cloud data centers," *Softw. Pract. Exp.*, vol. 47, no. 4, pp. 505–521, 2017.

[12] H. Gupta, A. V. Dastjerdi, S. K. Ghosh, and R. Buyya, "iFogSim: A toolkit for modeling and simulation of resource management techniques in the Internet of Things, edge and fog computing environments," *Softw. Pract. Exp.*, vol. 47, no. 9, pp. 1275–1296, 2017.

[13] The Linux Foundation. (2019). *OpenDaylight*. [Online]. Available: https://www.opendaylight.org/

[14] (2019). *Open vSwitch*. [Online]. Available: https://www.openvswitch.org/

[15] B. Pfaff and B. Davie. (Dec. 2013). *The Open vSwitch Database Management Protocol*. [Online]. Available: https://tools.ietf.org/html/rfc7047

[16] (2012). *OpenFlow Switch Specification*. [Online]. Available: https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.3.0.pdf

[17] (2019). Galaxy Technologies. *GNS3 Network Simulator*. [Online]. Available: https://gns3.com

[18] Mininet Team. (2019) *Mininet*. [Online]. Available: https://mininet.org

[19] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. D. Rose, and R. Buyya, "CloudSim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Softw. Pract. Exp.*, vol. 41, no. 1, pp. 23–50, 2011.

[20] N. Katta *et al.*, "Clove: Congestion-aware load balancing at the virtual edge," in *Proc. ACM 13th Int. Conf. Emerg. Netw. Exp. Technol. (CoNEXT)*, 2017, pp. 323–335.

[21] H. Zhang, J. Zhang, W. Bai, K. Chen, and M. Chowdhury, "Resilient datacenter load balancing in the wild," in *Proc. Conf. ACM Special Interest Group Data Commun. (SIGCOMM)*, 2017, pp. 253–266.

[22] A. V. Akella and K. Xiong, "Quality of service (QoS)-guaranteed network resource allocation via software defined networking (SDN)," in *Proc. IEEE 12th Int. Conf. Depend. Auton. Secure Comput.*, Aug. 2014, pp. 7–13.

[23] R. Kumar *et al.*, "End-to-end network delay guarantees for real-time systems using SDN," in *Proc. IEEE Real Time Syst. Symp. (RTSS)*, 2017, pp. 231–242.

[24] Docker. (2019). *Enterprise Container Platform for High Velocity Innovation*. [Online]. Available: https://www.docker.com/

[25] Docker. (2019). *Swarm Mode Overview*. [Online]. Available: https://docs.docker.com/engine/swarm/

[26] (2019). *Production-Grade Container Orchestration*. [Online]. Available: https://kubernetes.io/

[27] K. Govindaraj and A. Artemenko, "Container live migration for latency critical industrial applications on edge computing," in *Proc. IEEE 23rd Int. Conf. Emerg. Technol. Factory Autom. (ETFA)*, vol. 1, 2018, pp. 83–90.

[28] (2019). EveryWare Lab. *User Trace Simulations Project*. [Online]. Available: http://everywarelab.di.unimi.it/lbs-datasim

[29] V. Fang, T. Lvai, S. Han, S. Ratnasamy, B. Raghavan, and J. Sherry, "Evaluating software switches: Hard or hopeless?" EECS Dept., Univ. California at Berkeley, Berkeley, CA, USA, Rep. UCB/EECS-2018-136, 2018.

[30] Cisco Systems. (Jun. 2018). *OVSDB Plugin Release Notes, Release 2.3.1*. [Online]. Available: https://www.cisco.com/c/en/us/products/collateral/switches/nexus-9000-series-switches/white-paper-c11-740091.html

[31] Juniper Networks. (Aug. 2018). *OVSDB Support on Juniper Networks Devices*. [Online]. Available: https://www.juniper.net/documentation/en_US/junos/topics/reference/general/sdn-ovsdb-supported-platforms.html

[32] S. Yi, C. Li, and Q. Li, "A survey of fog computing: Concepts, applications and issues," in *Proc. ACM Workshop Mobile Big Data*, 2015, pp. 37–42.

[33] R. Enns, M. Bjorklund, J. Schoenwaelder, and A. Bierman, "Network configuration protocol (NETCONF)," IETF, RFC 6241, 2011.

[34] M. Bjorklund, "YANG—A data modeling language for the network configuration protocol (NETCONF)," IETF, RFC 6020, 2010.

[35] B. V. Cherkassky, A. V. Goldberg, and T. Radzik, "Shortest paths algorithms: Theory and experimental evaluation," *Math. Program.*, vol. 73, no. 2, pp. 129–174, 1996.

[36] R. Sedgewick, *Algorithms in C++*, vol. 2, 3rd ed. Boston, MA, USA: Addison-Wesley, 2002, p. 299.

[37] The Chromium Projects. (2019). *QUIC, a Multiplexed Stream Transport Over UDP*. [Online]. Available: https://www.chromium.org/quic

[38] P. Kumar and B. Dezfouli, "Implementation and analysis of QUIC for MQTT," *Comput. Netw.*, vol. 150, pp. 28–45, Feb. 2019.

[39] Docker. (2019). *Docker Swarm Reference Architecture: Exploring Scalable, Portable Docker Container Networks*. [Online]. Available: https://success.docker.com/article/networking

[40] M. Mortimer. (2019). *iPerf-Python*. [Online]. Available: https://github.com/thiezn/iperf3-python

[41] OMNeT++. (2019). *OMNeT++ Discrete Event Simulator*. [Online]. Available: https://omnetpp.org.org

[42] S. A. Jyothi, M. Dong, and P. Godfrey, "Towards a flexible data center fabric with source routing," in *Proc. ACM SIGCOMM Symp. Softw. Defined Netw. Res.*, 2015, p. 10.

[43] B. Pfaff *et al.*, "The design and implementation of open vSwitch," in *Proc. NSDI*, 2015, pp. 117–130.

[44] B. Dezfouli, M. Radi, and O. Chipara, "REWIMO: A real-time and reliable low-power wireless mobile network," *ACM Trans. Sensor Netw.*, vol. 13, no. 3, p. 17, 2017.

[45] S. A. Magid, F. Petrini, and B. Dezfouli, "Image classification on IoT edge devices: Profiling and modeling," *Clust. Comput.*, pp. 1–19, Aug. 2019. [Online]. Available: https://link.springer.com/article/10.1007/s10586-019-02971-9