

Santa Clara University

Scholar Commons

Computer Science and Engineering Master's
Theses

Engineering Master's Theses

12-4-2019

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

Colton Powell
Santa Clara University

Follow this and additional works at: https://scholarcommons.scu.edu/cseng_mstr



Part of the [Computer Engineering Commons](#)

Recommended Citation

Powell, Colton, "The Fog Development Kit: A Platform for the Development and Management of Fog Systems" (2019). *Computer Science and Engineering Master's Theses*. 14.
https://scholarcommons.scu.edu/cseng_mstr/14

This Dissertation is brought to you for free and open access by the Engineering Master's Theses at Scholar Commons. It has been accepted for inclusion in Computer Science and Engineering Master's Theses by an authorized administrator of Scholar Commons. For more information, please contact rscroggin@scu.edu.

Santa Clara University

Department of Computer Engineering

Date: December 4, 2019

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY
SUPERVISION BY

Colton Powell

ENTITLED

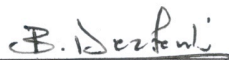
**The Fog Development Kit: A Platform for the
Development and Management of Fog Systems**

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR

THE DEGREE OF

**MASTER OF SCIENCE IN COMPUTER SCIENCE AND
ENGINEERING**



Thesis Advisor
Dr. Behnam Dezfouli



Chairman of Department
Dr. Silvia Figueira



Thesis Reader
Dr. Silvia Figueira

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

By

Colton Powell

Dissertation

Submitted in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in Computer Science and Engineering
in the School of Engineering at
Santa Clara University, 2019

Santa Clara, California

Acknowledgments

I would like to thank my advisor Dr. Behnam Dezfouli for all of his help throughout the past year on this project. His guidance was essential in making this project a reality, and the result was far more amazing than I would have ever expected. The work on this project was extremely challenging, but was also equally rewarding. We made a lot of accomplishments together throughout the past year, and they have equipped me with the skills and confidence to do great work anywhere I go in the future. Thank you for your guidance and your assistance!

In addition, I would like to thank all of my fellow researchers and colleagues in the SIOTLAB that I've met over the past year. In particular, I would like to thank Christopher Desiniotis for his hard work and significant contributions to this project. You put in some serious effort into this project, and I will always remember those long nights in the lab working on it with you!

I would also like to thank my family for all of their support. Specifically, I want to thank my Mom for her support throughout all of this, and for allowing me to focus on my schooling and making my dreams a reality. I also want to thank my Dad for his encouragement on pursuing a career that I love, and to thank the rest of my family for their support throughout my schooling.

Finally, I would like to thank my friends and professors for all of their support during my time spent at SCU. You have all helped me in ways that I will be forever thankful for.

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

Colton Powell

Department of Computer Engineering
Santa Clara University
Santa Clara, California
2019

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.

Table of Contents

1	Introduction	1
2	Related Work	6
2.1	Simulation Platforms	6
2.2	Resource Management and Allocation	7
2.3	Fog Architectures and Platforms	10
3	The Fog Development Kit	12
3.1	Resource Allocation	12
3.2	Agility	13
3.3	Portability	13
3.4	Application Independence	14
4	System Architecture	15
4.1	TopologyManager	17
4.2	FlowManager	20
4.3	ResourceManager	21
5	Evaluation	31
5.1	Verification and Evaluation using a Physical Testbed	31
5.1.1	Test 1: Resource Allocation and Deallocation	35
5.1.2	Test 2: Bandwidth Guarantee	38
5.1.3	Test 3: Multiple Bandwidth Guarantees	41
5.2	Scalability Analysis	43
6	Future Work	49

7 Conclusion	51
Bibliography	52

List of Figures

4.1	Overall system architecture.	16
4.2	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 towards the fog-device as well as those towards 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.	27
5.1	The physical testbed used to implement the topology depicted in Figure 5.2. End-devices, switches, and fog-devices are connected through physical links.	32
5.2	The network topology used for the development and testing of the FDK. C_i represents a container running on a fog-device.	33
5.3	Overall bandwidth of the data received by fog-devices corresponding to the images captured and sent by end-devices via <i>detection-app</i> . Blue (solid) and red (dashed) bars denote service requests and shutdown requests made by end-devices, respectively. Bars which 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.	35
5.4	Empirical Cumulative Distribution Function (ECDF) graphs for Test 1a. In this test, end-devices issue service requests <i>sequentially</i> . Groups closer to the controller (and therefore FDK) complete all of their operations slightly faster than those further from the controller.	36
5.5	Empirical Cumulative Distribution Function (ECDF) graphs for Test 1b. In this test, end-devices issue service requests <i>concurrently</i> . 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 <i>sequentially</i>	37

5.6	Bandwidth readings for end-devices 1, 4, and 6 throughout Test 2a (300 Mbps allocation). The bars show the sequential execution of <i>sleep-app</i> by 7 end-devices. These results show that there is no additional variation in bandwidth for running fog applications in the presence of <i>sequential</i> service requests made to the FDK by other end-devices.	39
5.7	Bandwidth readings for end-devices 1, 4, and 6 during Test 2b (300 Mbps allocation). The vertical lines show the instances the 7 end-devices start <i>sleep-app</i> concurrently. These results show that there is no additional variation in bandwidth in the presence of <i>concurrent</i> service requests made to the FDK by other end-devices.	40
5.8	Actual bandwidth readings for Tests 3a, 3b, and 3c for each Group. End-devices 1 through 7 run <i>iperf-app</i> , and end-device 8 performs 15 concurrent runs of <i>sleep-app</i> at 30 and 60 seconds (as indicated by the vertical lines) into the 90-second <i>iperf-app</i> transmissions. Even under network congestion and stress during these times, the results show that bandwidth allocations are enforced and no additional variation is observable.	42
5.9	The two topologies used for scalability evaluation. These two topologies are referred to as 'Topology (a)' and 'Topology (b)' in the text.	44
5.10	Execution time of the RAA (excluding switch and fog-device configuration delay) versus network size and number of fog-devices. Sub-figures (a) and (b) present the results for Topology (a) and (b) of Figure 5.9, 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.	45
5.11	Communication delay of resource allocation versus end-device to fog-device distance (hops) for Topology (a) presented in Figure 5.9. . .	46
5.12	Communication delay of resource allocation versus end-device to fog-device distance (hops) for Topology (b) presented in Figure 5.9. . .	47

List of Tables

4.1	TopologyManager APIs	18
4.2	FlowManager APIs	21
4.3	ResourceManager APIs	22

List of Abbreviations

ECMP Equal-Cost Multi-Path (routing)

FCT Flow Completion Time

FDK Fog Development Kit

IoT Internet of Things

MD-SAL Model-Driven Service Abstraction Layer

ODL OpenDaylight

OVS Open vSwitch

OVSDB Open vSwitch Database

QoS Quality of Service

RAA Resource Allocation Algorithm

RDA Resource Deallocation Algorithm

SDN Software-Defined Networking

SIOTLAB Santa Clara University Internet of Things Laboratory

VM Virtual Machine

YANG Yet Another Next-Generation (Data Modeling Language)

CHAPTER 1

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. Whether they be televisions, sensors, or wearable devices, these technologies often generate data and require computation and storage needs that cannot be met at the network edge. With the growing number of interconnected devices and IoT applications arises the challenge of handling 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, storage, and communication demands of billions of IoT devices [1–4]. Fog computing aims to solve this challenge by bringing additional computing, storage, and control capabilities to the network edge. The increased number of powerful computation and networking platforms has made the implementation of fog architectures a worthwhile undertaking [5]. Fog is intended to work alongside the cloud, forming a things-fog-cloud continuum where applications can be served promptly [1].

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 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 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 thesis, 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 all of 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 in 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

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

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 developing applications 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 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 virtual machines (VM) 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.

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 ms and 256 ms, respectively, for 95% of transactions. We also show the resiliency of the FDK by analyzing the impact of various network conditions and levels of congestion on already-running application transmission speeds and 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 rest of this thesis is organized as follows. We present the related work in Chapter 2. In Chapter 3, we summarize the goals and features of the FDK. Chapter 4 presents the system architecture and operation of the FDK. In Chapter 5, we present performance evaluation using a physical testbed and simulation. In Chapter 6, we highlight potential future work, and lastly in Chapter 7 we conclude the thesis.

CHAPTER 2

Related Work

In this chapter, 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.

2.1 Simulation Platforms

Due to the significant cost of creating fog and cloud network infrastructures, 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], Con-

tainerCloudSim [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).

2.2 Resource Management and Allocation

Resource management is key to the success of any fog system and consists of two main components: *networking resource management*, and *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 Multi-Path (ECMP) routing is used to manage network resources by distributing traffic throughout the network. However, ECMP is congestion oblivious and studies [20, 21] suggest that ECMP’s performance is far from optimal and that it

is known to result in unevenly distributed network flows and poor performance. 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 10%-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 *et al.* [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], which provides 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 run-

ning services, degrading performance and increasing completion delays. Ansari *et al.* [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.

2.3 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.

CHAPTER 3

The Fog Development Kit

The FDK addresses all of the problems mentioned in Chapter 2 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 chapter presents the main features offered by the FDK.

3.1 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.

3.2 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.

3.3 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, the FDK is designed to be highly portable. Fog-based applications written on top of the FDK are intended to be 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 wide-spread acceptance of OVS in large-scale environments [29], we used OVS as our switch software. OVS can also 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.

3.4 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 being run on top of it, 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 in the fog-devices to handle their processing needs. Conversely, the messaging protocol also provides methods to deallocate resources and terminate containerized applications. Therefore, 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 4.3 we show how various types of applications can be adapted to work with the FDK.

CHAPTER 4

System Architecture

Figure 4.1 shows the overall fog system architecture including four major components: *controller*, *end-devices*, *switches*, and *fog-devices*. End-devices are resource-constrained and cannot completely satisfy application requirements [1, 3, 4, 32]. Therefore, these devices communicate with more powerful machines—fog-devices—to offload their computing¹ 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

¹Here, *computing* is a general term that includes processing, storage, and communication.

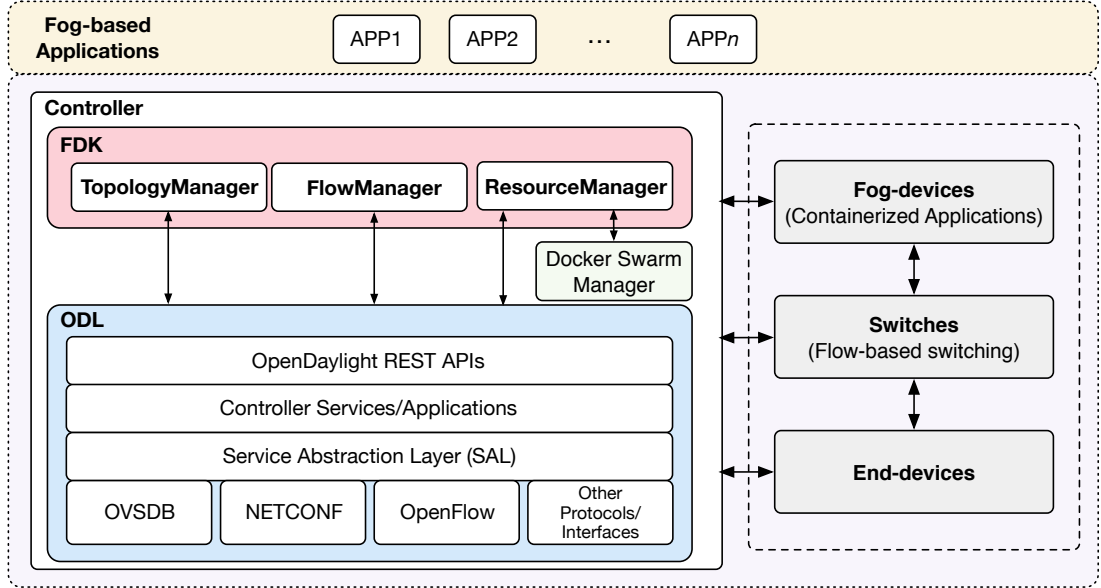


Fig. 4.1: Overall system architecture.

ODL’s northbound REST interfaces and 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 ultimately configure or modify devices on the network. 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 management using a variety of southbound protocols. For example, the FDK pushes OpenFlow flows to switches and then remotely configures 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 chapter.

4.1 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 4.1.

Table 4.1: TopologyManager APIs

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

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 network devices (i.e., end-devices, fog-devices, switches) and connecting 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.

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, in addition to 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 to 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 4.3 presents this mechanism in detail.

4.2 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 4.2. 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

Table 4.2: FlowManager APIs

API	Description
<code>create_flow()</code>	Push OpenFlow flow to switch
<code>delete_flow()</code>	Delete OpenFlow flow from switch
<code>track_flow()</code>	Track flow information
<code>untrack_flow()</code>	Untrack flow information

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.

4.3 ResourceManager

The FDK uses the *ResourceManager* component to manage and allocate all networking and computing resources. The core APIs of this component are described in Table 4.3. 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

Table 4.3: ResourceManager APIs

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

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 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 reserved, 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.

Algorithm 1: Resource Allocation Algorithm (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  $\mathcal{H} = \text{minHeap}(k)$  //K-ary min heap
21  $\text{init\_link} = (\text{src} : e_i, \text{dst} : e_i, \text{weight} : 0)$ 
22  $\mathcal{H}.\text{push}(\text{init\_link})$ 
23  $\mathcal{B}[e_i] = \text{init\_link}$ 
24 //find least-cost paths from  $e_i$  to fog-devices
25 while  $\text{size}(\mathcal{H}) > 0$  do
26      $u = \mathcal{H}.\text{pop\_min}()$ 
27     if  $u.\text{src} \neq u.\text{dst}$  then  $\mathcal{P}[u.\text{dst}] = u$ 
28     for  $v \in \{\text{outgoing links of } u.\text{dst}\}$  do
29         //v.src is equivalent to u.dst
30          $v.\text{weight} = \mathcal{B}[v.\text{src}].\text{weight} + 1/(T_B(v) - A_B(v))$ 
31         if  $(T_B(v) - A_B(v)) < R_B(e_i)$  then  $v.\text{weight} = \infty$ 
32         if  $v.\text{dst} \notin \mathcal{B}$  then
33              $\mathcal{H}.\text{push}(v)$ 
34              $\mathcal{B}[v.\text{dst}] = v$ 
35         else if  $v.\text{weight} < \mathcal{B}[v.\text{dst}].\text{weight}$  then
36             //Update link, shift based on weight
37              $\mathcal{H}.\text{decrease\_key}(\mathcal{B}[v.\text{dst}], v)$ 
38              $\mathcal{B}[v.\text{dst}] = v$ 
39 //find the best fog-device to fulfill the request
40  $\text{min} = \infty$ 
41 for  $f_j \in \mathcal{F}'$  do
42     if  $\mathcal{P}[f_j].\text{weight} < \text{min}$  then
43          $\text{min} = \mathcal{P}[f_j].\text{weight}$ 
44          $f_{\text{min}} = f_j$ 
45 if  $\text{min} == \infty$  then return FAILURE response
```

```

46 //configure switches along the path  $f_{min}$  to  $e_i$ 
47  $v = \mathcal{P}[f_{min}]$ 
48 while true do
49   if  $v.src == e_i$  then return SUCCESS response
50   Create rate-limited queues on  $v.src$ 
51   Place queues on appropriate QoS entry in  $v.src$ 
52   Create flows on  $v.src$  to redirect traffic to rate-limited queues
53    $v = \mathcal{P}[v.src]$ 

```

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 run-time 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 pre-processing 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 normal background) 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 30). 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 also results in the selection of paths throughout the network which 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 reached already), then the link used to reach n is pushed onto \mathcal{H} (line 33). Otherwise, n has been reached already 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 37). Then, the new link is shifted upwards 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 27).

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 44). 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 45). 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 .

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 through 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 50). 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 51). 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

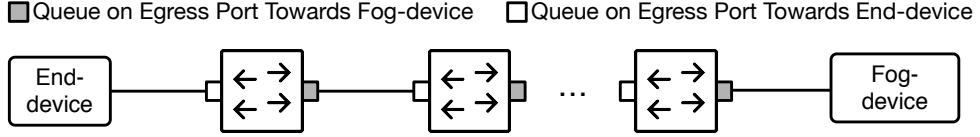


Fig. 4.2: Enforcing bandwidth reservation using rate-limited queues. For each path reservation, rate-limited packet queues are created and attached to QoS configurations located on the egress ports towards the fog-device as well as those towards the end-device. Then, flow table entries are pushed via OpenFlow to enqueue traffic traveling from the end-device to the fog-device, and vice versa, on these queues.

path by leveraging the FlowManager flow-creation APIs (line 52). 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. Figure 4.2 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 (flow classification) 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 49), 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. 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 to 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 work we particularly focused on the former due to its higher security and wide-spread 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, non-open-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 allocation features. 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. In short, 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.

CHAPTER 5

Evaluation

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

5.1 Verification and Evaluation using a Physical Testbed

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

Testbed. Figure 5.1 shows our testbed, which includes five OpenFlow switches, four fog-devices, and eight end-devices. This testbed implements the network presented in Figure 5.2. Each end-device is a Raspberry Pi Model 3 B+ (running Raspbian Linux) which is connected to a switch using a 1 Gbps 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 switches. The 2-port NIC is paired with one of the aforementioned 4-port NICs to build a 6-port switch which is connected using a 1 Gbps cable to the controller. Both machines include two 16-core Intel Xeon CPUs and 64 GB RAM. Each fog-device and

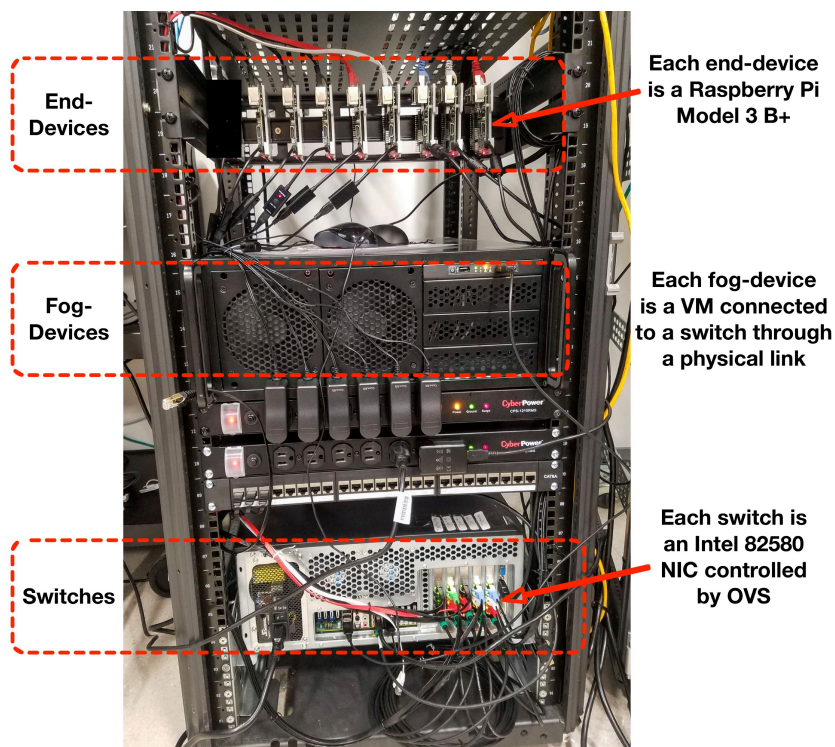


Fig. 5.1: The physical testbed used to implement the topology depicted in Figure 5.2. End-devices, switches, and fog-devices are connected through physical links.

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 Figure 5.2, 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 com-

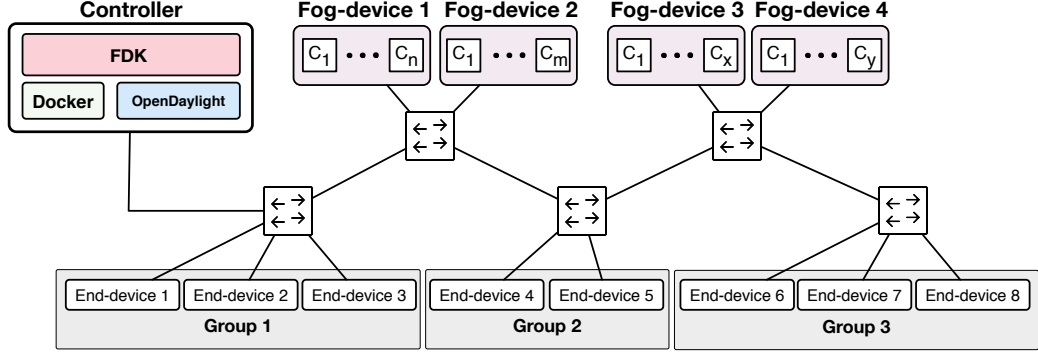


Fig. 5.2: The network topology used for the development and testing of the FDK. C_i represents a container running on a fog-device.

pared 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 passes 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. Finally, we 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. Before running any tests, and in order to confirm the functionality of the FDK, 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 we configured end-devices to randomly wake-up, issue a service request, and then capture and stream images to services on the fog-devices running object recognition algorithms. Each service request specifies a desired bandwidth allocation of 40 Mbps. Each end-device, after about 7 seconds into its streaming period, ceases its

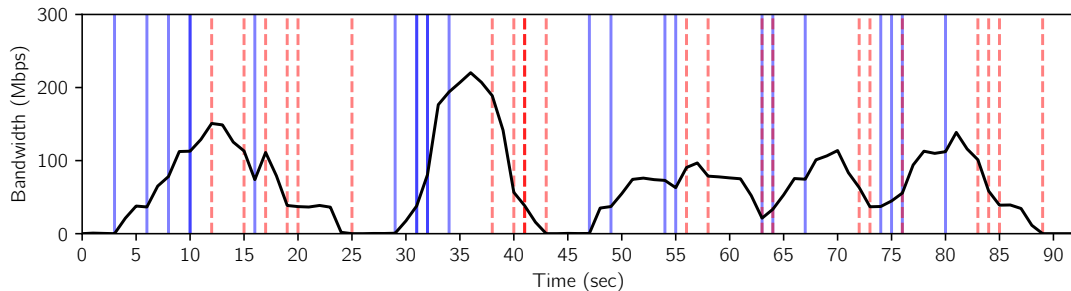


Fig. 5.3: Overall bandwidth of the data received by fog-devices corresponding to the images captured and sent by end-devices via *detection-app*. Blue (solid) and red (dashed) bars denote service requests and shutdown requests made by end-devices, respectively. Bars which are less transparent indicate 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.

image streaming and sends a shutdown request. We performed similar verification steps to ensure that resources were all allocated and deallocated properly. Figure 5.3 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 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 service and shutdown 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.

5.1.1 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

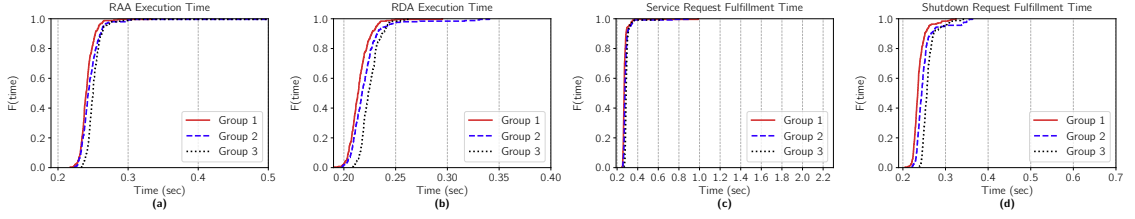


Fig. 5.4: Empirical Cumulative Distribution Function (ECDF) graphs for Test 1a. In this test, end-devices issue service requests *sequentially*. Groups closer to the controller (and therefore FDK) complete all of their operations slightly faster than those further from the controller.

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 seconds after receiving service, and then issues a shutdown request. After completion, the rest of the end-devices perform the same operation sequentially. Figure 5.4 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 ECDF graphs. As seen in Figure 5.4, more than 95% of all operations completed within 0.33 seconds 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

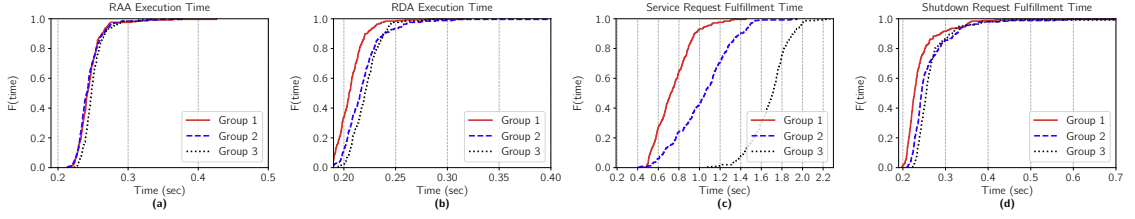


Fig. 5.5: Empirical Cumulative Distribution Function (ECDF) graphs for Test 1b. In this test, end-devices issue service requests *concurrently*. Groups closer to the controller experience significantly faster service request fulfillment times compared to those further from the controller. This is because the FDK processes requests *sequentially*.

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 seconds upon receiving a successful response, and then send a shutdown request. Figure 5.5 presents the results of Test 1b. The results presented in Figures 5.5(a) and 5.5(b) are nearly identical to the corresponding Figures 5.4(a) and 5.4(b) from Test 1a, with 95% of these operations completing within 0.28 seconds across all Groups. However, the results for service request fulfillment times in Test 1b, shown in Figure 5.5(c), look considerably different compared to the corresponding Figure 5.4(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 seconds, respectively. Also, there is far less variation in the results for shutdown request fulfillment times, as Figure 5.5(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 seconds, respectively.

Because 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 Figure 5.5(c). Here, 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.

5.1.2 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 seconds with a 300 Mbps 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 Mbps. Using *iperf-app*, an end-device continuously streams data to a container for 90 seconds. Then, at 30 and 60 seconds into the 90-second transmission, all 7 other end-devices in the topology run *sleep-app* for one

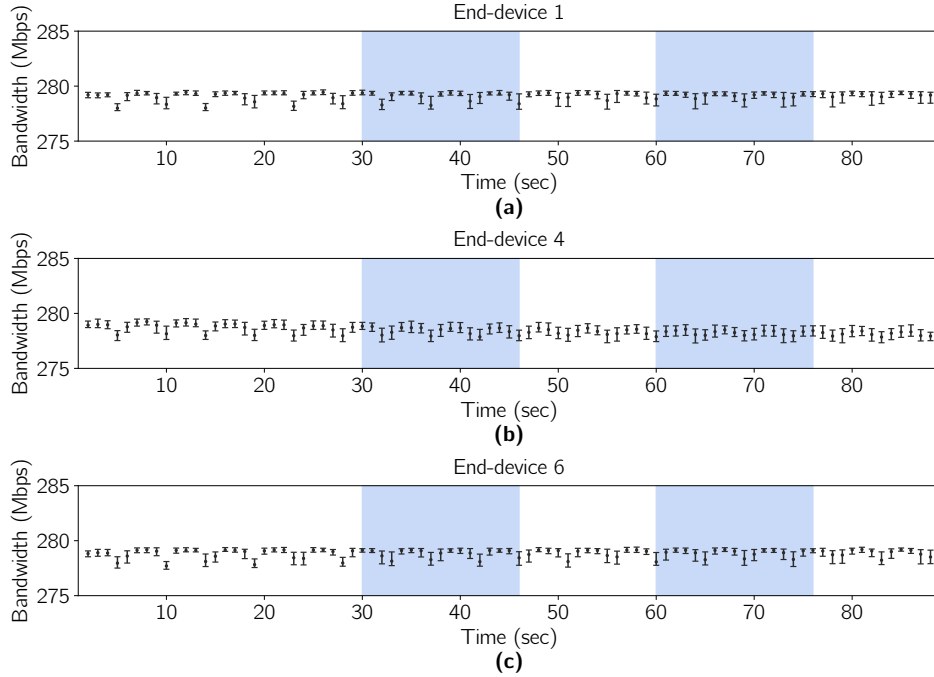


Fig. 5.6: Bandwidth readings for end-devices 1, 4, and 6 throughout Test 2a (300 Mbps allocation). The bars show the sequential execution of *sleep-app* by 7 end-devices. These results show that there is no additional variation in bandwidth for running fog applications in the presence of *sequential* service requests made to the FDK by other end-devices.

second. 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 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 modified 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-second gap in between each run. Figure 5.6 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

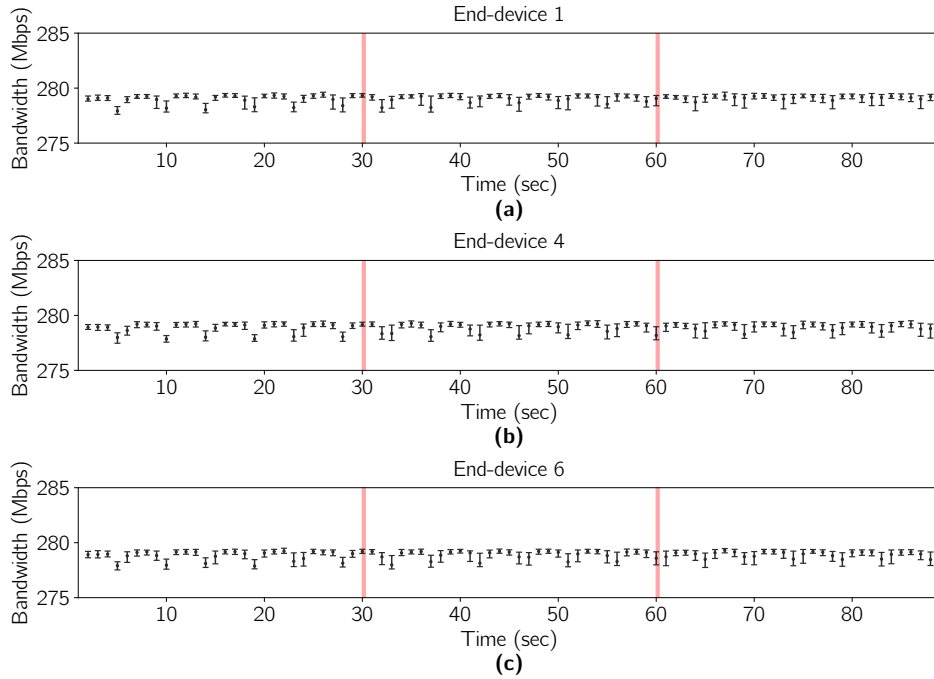


Fig. 5.7: Bandwidth readings for end-devices 1, 4, and 6 during Test 2b (300 Mbps allocation). The vertical lines show the instances the 7 end-devices start *sleep-app* concurrently. These results show that there is no additional variation in bandwidth in the presence of *concurrent* service requests made to the FDK by other end-devices.

additional messages are flowing through the network at around 30-45 seconds and 60-75 seconds, 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 seconds into transmission are executed *concurrently*. Figure 5.7 presents the results. Similar to the results of Test 2a, we see that there is essentially no drop or variation in bandwidth.

5.1.3 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 seconds. Then, at 30 seconds and 60 seconds into the 90-second 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 total. Finally, three different variations of Test 3 are executed, where different bandwidth allocations of 100 Mbps (Test 3a), 200 Mbps (Test 3b), and 300 Mbps (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 one-second bandwidth reading across the end-devices in the three groups. For example, in the case of Group 1, we initially had 3 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 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

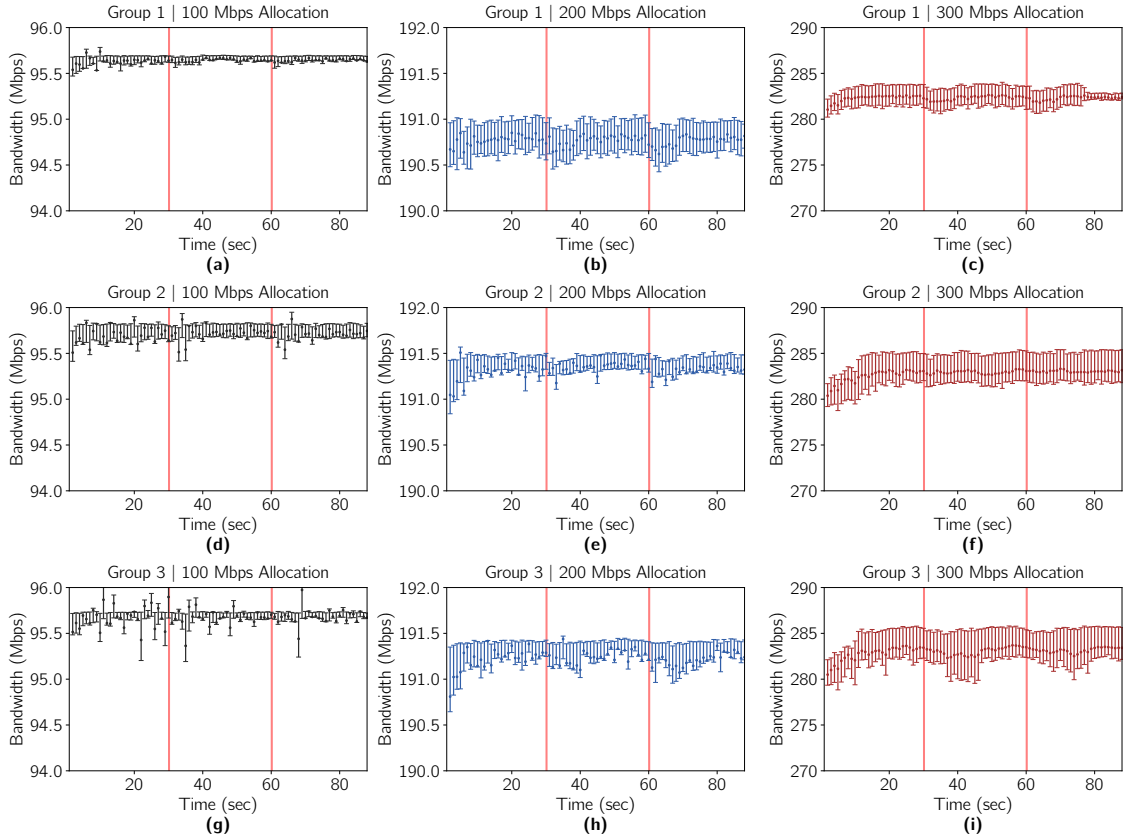


Fig. 5.8: Actual bandwidth readings for Tests 3a, 3b, and 3c for each Group. End-devices 1 through 7 run *iperf-app*, and end-device 8 performs 15 concurrent runs of *sleep-app* at 30 and 60 seconds (as indicated by the vertical lines) into the 90-second *iperf-app* transmissions. Even under network congestion and stress during these times, the results show that bandwidth allocations are enforced and no additional variation is observable.

Raspberry Pi.

Figure 5.8 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 sub-figure represents all of the data collected for an entire Group. These results demonstrate less than 1 Mbps variations for 100 Mbps and 200 Mbps allocations, and less than 5 Mbps variations for 300 Mbps allocations. More specifically, Figure 5.8 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 Mbps *iperf-app* runs, there are more variations in the bandwidth readings than the 200 Mbps and 100 Mbps runs. However, these variations do not correspond to the additional messages flowing throughout the network at 30 and 60 seconds into the 90-second *iperf-app* transmission. We believe that this is caused by the processing and queuing delays of 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.

5.2 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-device, (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

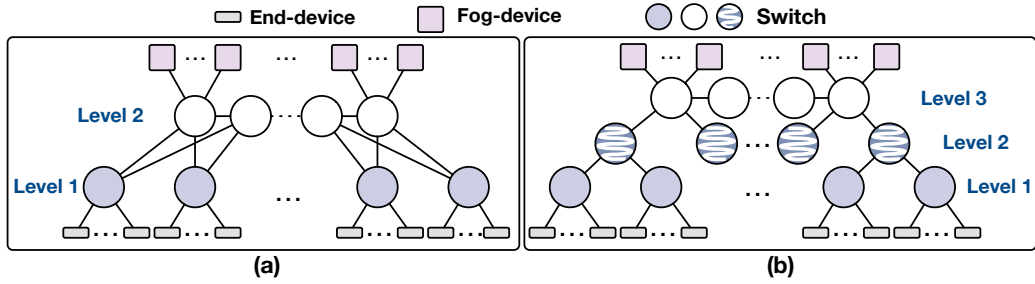


Fig. 5.9: The two topologies used for scalability evaluation. These two topologies are referred to as 'Topology (a)' and 'Topology (b)' in the text.

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]. Figures 5.9(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

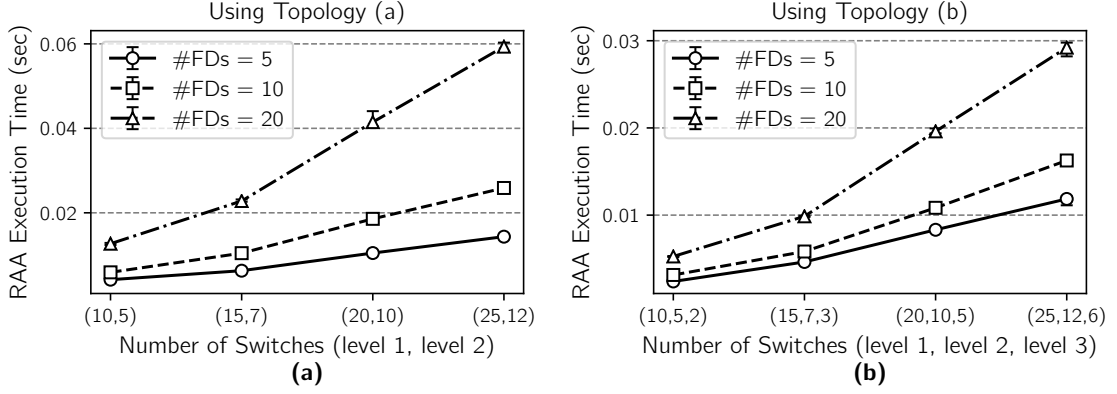


Fig. 5.10: Execution time of the RAA (excluding switch and fog-device configuration delay) versus network size and number of fog-devices. Sub-figures (a) and (b) present the results for Topology (a) and (b) of Figure 5.9, respectively. #FDs refers to the number of fog-devices per level 2 switch in Topology (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.

are horizontally connected.

Figure 5.10 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 presents 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 4.3, 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 a 120% and 64% increase in execution time in Figures 5.10(a) and (b), respectively. Comparing Figures 5.10(a) and (b) shows that the execution time on 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

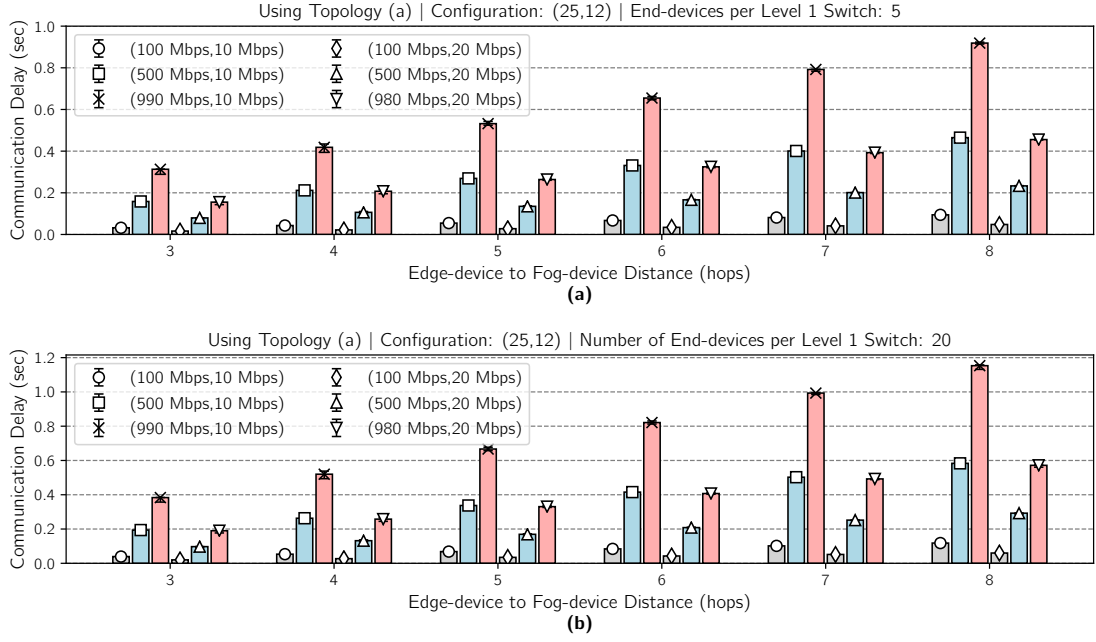


Fig. 5.11: Communication delay of resource allocation versus end-device to fog-device distance (hops) for Topology (a) presented in Figure 5.9.

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), the number of paths between each end-device and fog-device is lower in the former topology.

The next set of results presents the communication delay of resource allocation. Figures 5.11 and 5.12 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 Figures 5.11 and 5.12 exhibit the impact of the number of hops and background traffic on allocation delay. The figures show that a higher number of hops

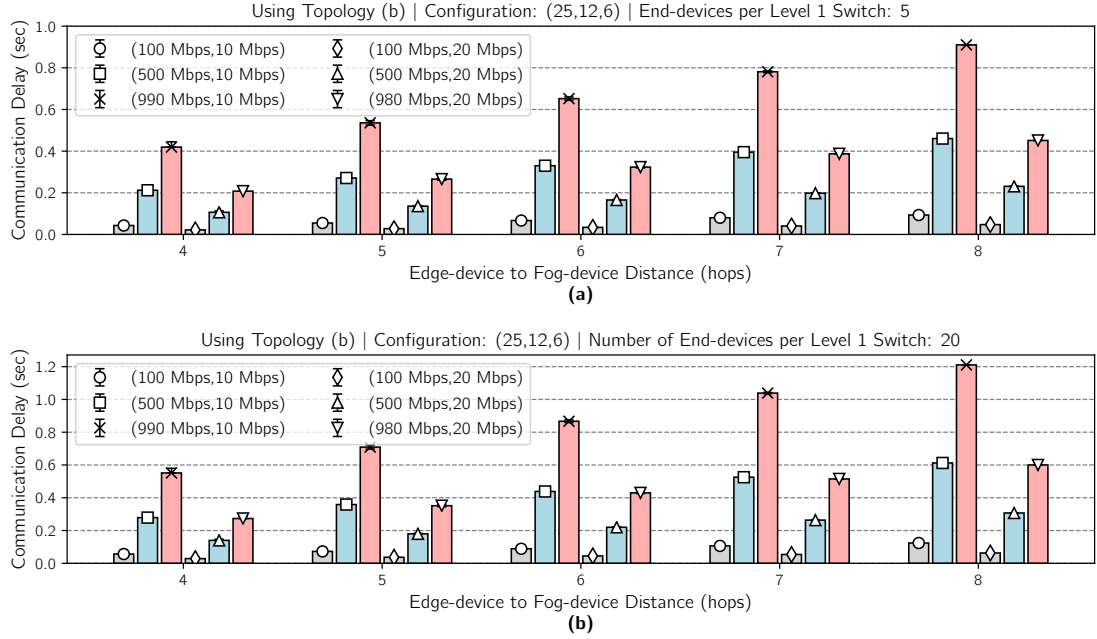


Fig. 5.12: Communication delay of resource allocation versus end-device to fog-device distance (hops) for Topology (b) presented in Figure 5.9.

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 5x 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 trade-off between resource allocation delay and the communication resources available for data flows.

Figures 5.11 and 5.12 also reveal that increasing the number of end-devices results in a higher allocation delay. Specifically, 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 Figure 5.10 we highlighted that Topology (b) reduces execution time by about 50%. In contrast, Figures 5.11(b) and 5.12(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 Figures 5.11(b) and 5.12(b)). In summary, by putting together the results of Figure 5.10 and 5.12, 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.

CHAPTER 6

Future Work

In this chapter 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, for large-scale fog systems with stringent resource allocation deadlines, it is essential to partition the network into regions controlled separately. Specifically, we propose to use a local controller in each region. These local controllers are provided with pre-allocated 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 Mbps, 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 a 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., resource allocation algorithms 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.

CHAPTER 7

Conclusion

In this thesis, we proposed the *Fog Development Kit (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 applications 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. Moreover, 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 at the same time abstracting and eliminating the complexity of resource allocation away from developers.

Bibliography

- [1] M. Chang and T. Zhang, “Fog and IoT: An Overview of Research Opportunities,” *IEEE Internet of Things Journal*, vol. 3, no. 6, pp. 854–864, 2016. 1, 15
- [2] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, “Fog Computing and Its Role in the Internet of Things,” in *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, 2012, pp. 13–16. 1
- [3] W. Shi and S. Dustdar, “The Promise of Edge Computing,” *Computer*, vol. 49, no. 5, pp. 78–81, 2016. 1, 15
- [4] 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 *Proceedings of the Workshop on Mobile Big Data*. ACM, 2015, pp. 43–48. 1, 15
- [5] I. Amirtharaj, T. Groot, and B. Dezfouli, “Profiling and Improving the Duty-Cycling Performance of Linux-based IoT Devices,” *Journal of Ambient Intelligence and Humanized Computing*, pp. 1–29, 2018. 1
- [6] N. Wang, B. Varghese, M. Matthaiou, and D. S. Nikolopoulos, “ENORM: A Framework For Edge NOde Resource Management,” *IEEE Transactions on Services Computing*, 2017. 2, 11

- [7] L. Yin, J. Luo, and H. Luo, “Tasks Scheduling and Resource Allocation in Fog Computing Based on Containers for Smart Manufacture,” *IEEE Transactions on Industrial Informatics*, vol. 14, no. 10, pp. 4712–4721, 2018. 2, 6, 10
- [8] O. Skarlat, S. Schulte, M. Borkowski, and P. Leitner, “Resource Provisioning for IoT Services in the Fog,” in *IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*, 2016, pp. 32–39. 2, 6, 10
- [9] N. Ansari and X. Sun, “Mobile Edge Computing Empowers Internet of Things,” *IEICE Transactions on Communications*, vol. 101, no. 3, pp. 604–619, 2018. 2, 6, 10
- [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 *15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2015, pp. 475–484. 2, 6
- [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,” *Software: Practice and Experience*, vol. 47, no. 4, pp. 505–521, 2017. 2, 7
- [12] H. Gupta, A. Vahid 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,” *Software: Practice and Experience*, vol. 47, no. 9, pp. 1275–1296, 2017. 2, 7
- [13] The Linux Foundation, “OpenDaylight.” [Online]. Available: <https://www.opendaylight.org/> 4
- [14] “Open vSwitch.” [Online]. Available: <https://www.openvswitch.org/> 4

- [15] B. Pfaff and B. Davie, “The Open vSwitch Database Management Protocol,” Dec 2013. [Online]. Available: <https://tools.ietf.org/html/rfc7047> 4
- [16] “OpenFlow Switch Specification,” 2012. [Online]. Available: <https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.3.0.pdf> 4
- [17] Galaxy Technologies,. GNS3 Network Simulator. [Online]. Available: <https://gns3.com> 4, 13
- [18] Mininet Team. (2019) Mininet. [Online]. Available: <https://mininet.org> 4, 13
- [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,” *Software: Practice and Experience*, vol. 41, no. 1, pp. 23–50, 2011. 6
- [20] N. Katta, A. Ghag, M. Hira, I. Keslassy, A. Bergman, C. Kim, and J. Rexford, “Clove: Congestion-Aware Load Balancing at the Virtual Edge,” in *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*. ACM, 2017, pp. 323–335. 7, 8
- [21] H. Zhang, J. Zhang, W. Bai, K. Chen, and M. Chowdhury, “Resilient Data-center Load Balancing in the Wild,” in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2017, pp. 253–266. 7, 8
- [22] A. V. Akella and K. Xiong, “Quality of Service (QoS)-Guaranteed Network Resource Allocation via Software Defined Networking (SDN),” in *IEEE 12th International Conference on Dependable, Autonomic and Secure Computing*, Aug 2014, pp. 7–13. 8

- [23] R. Kumar, M. Hasan, S. Padhy, K. Evchenko, L. Piramanayagam, S. Mohan, and R. B. Bobba, “End-to-End Network Delay Guarantees for Real-Time Systems Using SDN,” in *IEEE Real-Time Systems Symposium (RTSS)*, 2017, pp. 231–242. 8, 49
- [24] Docker, “Enterprise Container Platform for High Velocity Innovation.” [Online]. Available: <https://www.docker.com/> 9, 17
- [25] Docker, “Swarm Mode Overview.” [Online]. Available: <https://docs.docker.com/engine/swarm/> 9
- [26] “Production-Grade Container Orchestration.” [Online]. Available: <https://kubernetes.io/> 9
- [27] K. Govindaraj and A. Artemenko, “Container Live Migration for Latency Critical Industrial Applications on Edge Computing,” in *IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, vol. 1, 2018, pp. 83–90. 9
- [28] EveryWare Lab. User Trace Simulations Project. [Online]. Available: <http://everywarelab.di.unimi.it/lbs-datasim> 10
- [29] V. Fang, T. Lvai, S. Han, S. Ratnasamy, B. Raghavan, and J. Sherry, “Evaluating Software Switches: Hard or Hopeless?” *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2018-136*, 2018. 13, 43
- [30] Cisco Systems, “OVSDB Plugin Release Notes, Release 2.3.1,” Jun 2018. [Online]. Available: <https://www.cisco.com/c/en/us/products/collateral/switches/nexus-9000-series-switches/white-paper-c11-740091.html> 14

- [31] Juniper Networks, “OVSDB Support on Juniper Networks Devices,” Aug 2018. [Online]. Available: https://www.juniper.net/documentation/en_US/junos/topics/reference/general/sdn-ovsdb-supported-platforms.html 14
- [32] S. Yi, C. Li, and Q. Li, “A Survey of Fog Computing: Concepts, Applications and Issues,” in *Proceedings of the 2015 workshop on mobile big data*. ACM, 2015, pp. 37–42. 15
- [33] R. Enns, M. Bjorklund, J. Schoenwaelder, and A. Bierman, “Network Configuration Protocol (NETCONF),” RFC 6241, 2011. 16
- [34] M. Bjorklund, “YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF),” RFC 6020, 2010. 16
- [35] B. V. Cherkassky, A. V. Goldberg, and T. Radzik, “Shortest Paths Algorithms: Theory and Experimental Evaluation,” *Mathematical Programming*, vol. 73, no. 2, pp. 129–174, 1996. 24
- [36] R. Sedgewick, *Algorithms in C++*, 3rd ed. Addison-Wesley, 2002, vol. 2, p. 299. 24
- [37] The Chromium Projects, “QUIC, a multiplexed stream transport over UDP.” [Online]. Available: <https://www.chromium.org/quic> 28
- [38] P. Kumar and B. Dezfouli, “Implementation and analysis of QUIC for MQTT,” *Computer Networks*, vol. 150, pp. 28–45, 2019. 28
- [39] Docker, “Docker Swarm Reference Architecture: Exploring Scalable, Portable Docker Container Networks.” [Online]. Available: <https://success.docker.com/article/networking> 29

- [40] M. Mortimer, “iperf-python,” <https://github.com/thiezn/iperf3-python>, 2019. 33
- [41] OMNeT++. (2019) OMNeT++ Discrete Event Simulator. [Online]. Available: <https://omnetpp.org.org> 44
- [42] S. A. Jyothi, M. Dong, and P. Godfrey, “Towards a Flexible Data Center Fabric with Source Routing,” in *Proceedings of ACM SIGCOMM Symposium on Software Defined Networking Research*, 2015, p. 10. 44
- [43] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar *et al.*, “The Design and Implementation of Open vSwitch,” in *NSDI*, 2015, pp. 117–130. 50
- [44] B. Dezfouli, M. Radi, and O. Chipara, “Rewimo: A real-time and reliable low-power wireless mobile network,” *ACM Transactions on Sensor Networks (TOSN)*, vol. 13, no. 3, p. 17, 2017. 50
- [45] S. A. Magid, F. Petrini, and B. Dezfouli, “Image Classification on IoT Edge Devices: Profiling and Modeling,” *Cluster Computing*, <https://doi.org/10.1007/s10586-019-02971-9>, 2019. 50