SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

August 2019, Vol. 62 082102:1–082102:17
https://doi.org/10.1007/s11432-018-9749-8

An adaptive offloading framework for Android applications in mobile edge computing

Xing CHEN[1,2], Shihong CHEN[1,2], Yun MA[3], Bichun LIU[1,2], Ying ZHANG[4] & Gang HUANG[4*]

[1] *College of Mathematics and Computer Science, Fuzhou University, Fuzhou* 350116*, China;*
[2] *Fujian Key Laboratory of Network Computing and Intelligent Information Processing, Fuzhou University, Fuzhou* 350116*, China;*
[3] *School of Software, Tsinghua University, Beijing* 100084*, China;*
[4] *Key Laboratory of High Confidence Software Technologies, Ministry of Education, Beijing* 100871*, China*

Received 8 March 2018/Revised 18 July 2018/Accepted 19 December 2018/Published online 12 July 2019

Abstract Mobile edge computing (MEC) provides a fresh opportunity to significantly reduce the latency and battery energy consumption of mobile applications. It does so by enabling the offloading of parts of the applications on mobile edges, which are located in close proximity to the mobile devices. Owing to the geographical distribution of mobile edges and the mobility of mobile devices, the runtime environment of MEC is highly complex and dynamic. As a result, it is challenging for application developers to support computation offloading in MEC compared with the traditional approach in mobile cloud computing, where applications use only the cloud for offloading. On the one hand, developers have to make the offloading adaptive to the changing environment, where the offloading should dynamically occur among available computation nodes. On the other hand, developers have to effectively determine the offloading scheme each time the environment changes. To address these challenges, this paper proposes an adaptive framework that supports mobile applications with offloading capabilities in MEC. First, based on our previous study (DPartner), a new design pattern is proposed to enable an application to be dynamically offloaded among mobile devices, mobile edges, and the cloud. Second, an estimation model is designed to automatically determine the offloading scheme. In this model, different parts of the application may be executed on different computation nodes. Finally, an adaptive offloading framework is implemented to support the design pattern and the estimation model. We evaluate our framework on two real-world applications. The results demonstrate that our approach can aid in reducing the response time by 8%–50% and energy consumption by 9%–51% for computation-intensive applications.

Keywords computation offloading, software adaptation, mobile edge computing, application refactoring, Android

Citation Chen X, Chen S H, Ma Y, et al. An adaptive offloading framework for Android applications in mobile edge computing. Sci China Inf Sci, 2019, 62(8): 082102, https://doi.org/10.1007/s11432-018-9749-8

1 Introduction

With the increase in the application of artificial intelligence and big data, mobile applications are becoming both computation- and data-intensive. Meanwhile, with the rapid development of computing and communication technologies, the computation platform of mobile applications has expanded from smartphones and tablet computers to wearable devices [1] (smartwatches and smart glasses), vehicles [2], UAVs [3], etc. As a result, ensuring a satisfactory user experience with mobile applications is challenging [4]. On the one hand, the hardware configuration of different computation platforms is highly

*Corresponding author (email: hg@pku.edu.cn)

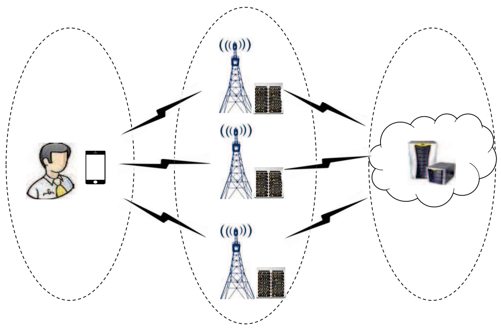

Figure 1 (Color online) Diagram of mobile edge computing.

heterogeneous so that the performance of an application varies significantly [5,6][1)][2)]. On the other hand, most of the computation platforms are powered by battery, wherein the battery capacity cannot satisfy the requirements of complex mobile applications [7–9].

Computation offloading, i.e., executing parts of an application on remote servers, is a commonly used technique to improve the performance and reduce the energy consumption of mobile applications [10–15]. Mobile cloud computing (MCC) has been introduced to extend the computing capability and battery capacity of mobile devices by offloading computation-intensive applications to the cloud. Nevertheless, the network communication between mobile devices and the cloud is likely to cause significant execution delay. To address this delay problem, mobile edge computing [16–20] (MEC) has also been introduced. Figure 1 illustrates this new diagram. The device can connect to nearby mobile edges, and each mobile edge connects to the cloud. The mobile edges provide computing capabilities in close proximity to mobile devices and enable the execution of highly demanding applications in mobile devices while offering significantly lower latencies.

Owing to the geographical distribution of mobile edges and mobility of mobile devices, the runtime environment of MEC is highly complex and dynamic. As the devices shift during the day, their contexts (e.g., locations, network conditions, and available mobile edges) keep changing. As a result, it is challenging for application developers to efficiently support computation offloading in MEC; this can be divided into two aspects.

First, developers need to implement the adaptation on offloading. The offloading scheme is determined at runtime to dynamically offload computation among available computation nodes. For example, if the mobile edge used for offloading becomes unavailable owing to an unstable network connection, the computation executed on the server should return to the device or go to another available server (mobile edge or cloud) in real time.

Secondly, each time the underlying environment changes, developers need to make trade-offs between the reduced execution time and the network delay in order to determine which parts are worth offloading and where to offload. Several factors need to be considered while deciding, including the processing power of the remote servers (mobile edge or cloud), quality of the network connections, computation complexity, and coupling degree of applications.

To address the challenges, in this paper, we present an adaptive framework; it supports Android applications with the offloading capability in MEC. Our study makes three major contributions:

- A new design pattern is proposed based on our previous study, DPartner [21]. The new design pattern enables an application to be offloaded among the mobile device, mobile edges and the cloud in a dynamic manner.
- An estimation model is designed to automatically determine the offloading scheme, in which different parts of the application may be executed on the device, mobile edges and cloud.
- An adaptive offloading framework is implemented to support the design pattern and the estimation

1) Android application requirements. https://developer.android.com/guide/components/fundamentals/.

2) Apps drain battery power. http://www.droidforums.net/threads/battery-drops-40-after-playing-game-for-hour.18301/.

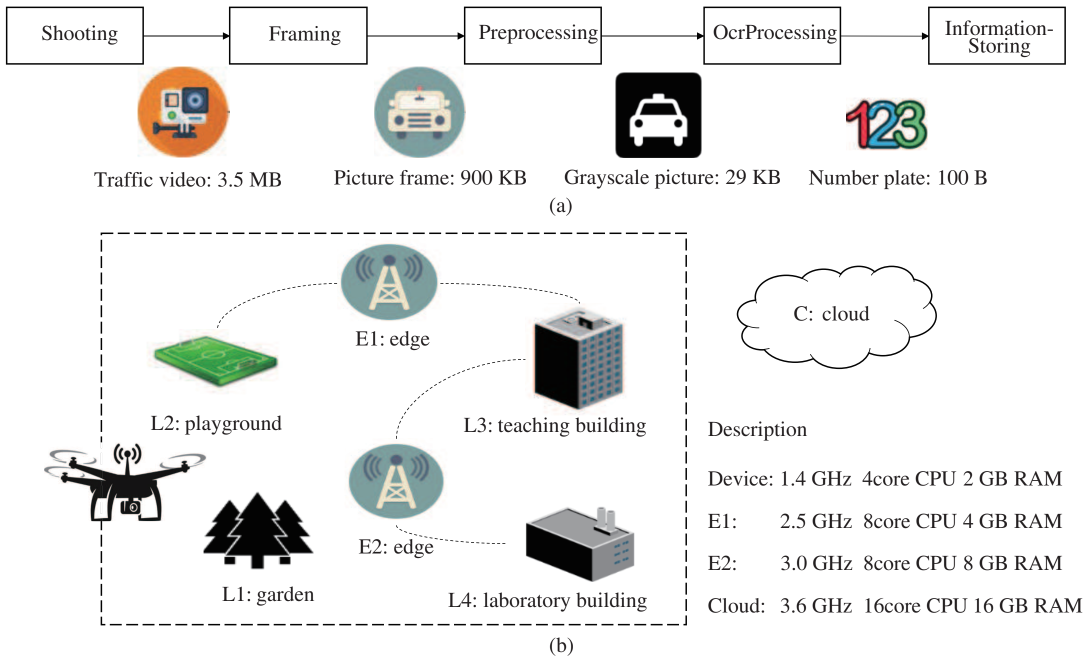

Figure 2 (Color online) Example of application in parking supervision. (a) Process of the license plate recognition application; (b) context of the drone in the campus.

model.

We apply our approach to support adaptive offloading on two real-world applications: a license plate recognition application and a voice recognition application. The results demonstrate that our approach can aid the reduction of execution time by 8%–50% and energy consumption by 9%–51% for computation-intensive applications.

The remaining paper is organized as follows: Section 2 overviews our approach with a supporting example. Section 3 presents the design pattern for computation offloading in MEC. Section 4 illustrates the estimation model for automatically determining the offloading scheme. Section 5 describes the implementation of the adaptive framework. Section 6 reports the evaluation of the two real-world applications. Section 7 introduces related work, and Section 8 concludes the paper.

2 Approach overview

This section briefly illustrates our approach through a motivating example.

2.1 A motivating example

At present, drones are used for supervision in a number of public areas. Here, we consider the parking supervision in a campus as an example. A drone cruises around the campus capturing videos of parked cars. When illegal parking is detected, a license plate recognition application in the drone will operate to identify the car's plate number. Figure 2(a) shows the process of the license plate recognition application. It applies image recognition technique to gather the plate number from the video stream. There are five main computation tasks in this application: shooting, framing, preprocessing, ocrProcessing, and informationStoring. Their computation complexities are different. For example, ocrProcessing is a computation-intensive task, and it is more effective to offload it to a remote server; meanwhile, framing exhibits low computation complexity. The data traffic between tasks is also different. Consider another example: the data traffic between shooting and framing is large so that it is more effective to execute these two tasks on the same device; meanwhile, the data traffic between preprocessing and ocrProcessing is marginal. Figure 2(b) shows that the context of the drone changes continuously as it cruises around the campus. There are several available remote servers (C, E1, and E2) that can be used for offloading in different locations. For example, E1 is available in the playground and the teaching building, whereas E2 can be accessed in the teaching building and lab building. For an improved application performance, when the drone is located in different places, it needs to determine where each computation task is executed and then offload each task to its corresponding server in real time.

2.2 Offloading in MEC

For effective offloading, it needs to automatically determine the offloading scheme according to the context of the mobile device and then offload the computation among the mobile devices, mobile edges, and cloud dynamically. It is challenging for our case; this challenge can be divided into two aspects.

On the one hand, the offloading scheme is determined at runtime so that it needs to dynamically offload the computation. Therefore, when the mobile device shifts from one location to another, the offloaded application needs to be reconfigured according to the new offloading scheme while remaining available for the end users.

On the other hand, it needs to calculate which parts are worth offloading and where to offload, according to the device context. In each location, apart from the cloud, there are several available mobile edges, which differ in the processing power and the quality of network conditions. It will consume a long time to collect runtime status and decide.

2.3 Our approach

To address the challenges, we propose an adaptive framework; it supports mobile applications with the offloading capability in MEC.

First, based on our previous study, DPartner, a new design pattern is proposed to enable an application to remain available when the device context changes and be offloaded among the device, mobile edges, and the cloud dynamically. Based on our previous study, DPartner, which can refactor Android applications to be offloaded between mobile devices and the cloud, we propose a new object-proxy mechanism to support adaptive offloading in the context of MEC. The offloaded computation is designed as remotely creating, migrating, and invoking objects performing the computation.

Secondly, an estimation model is designed to automatically determine the offloading scheme according to the device context, in which different parts of the application may be executed on the device, mobile edges, and cloud. We generate the control flow graph of the main activity of the application and retrieve the movable classes and their method invocations. Then, we introduce information models to collect history data on the average data traffic for each method invocation, and their execution time on the device, cloud, and each mobile edge. Based on this, we propose an algorithm to determine the optimal deployment location for each class.

The framework is implemented to support the design pattern and the estimation model, which is deployed on each computation node in the mobile edge environment. Applications that adhere to the design pattern are enabled by the framework to remain available and be offloaded among the device, mobile edges, and the cloud, when the mobile device shifts from one location to another. Moreover, the framework can automatically determine the offloading scheme for the application.

Next, we describe the details of the design pattern, estimation model, and framework in the following three sections, respectively.

3 Design pattern for computation offloading

For an Android application, it is organized as a set of classes associated with each other. Moreover, the class method represents the minimal granularity of computation that can be invoked. Thus, the offloading of computations can be implemented as remotely deploying and invoking a single class instance or a set of class instances performing computations. In this section, we first illustrate the design pattern that enables an application to be offloaded among the device, mobile edges, and the cloud dynamically. Then, we

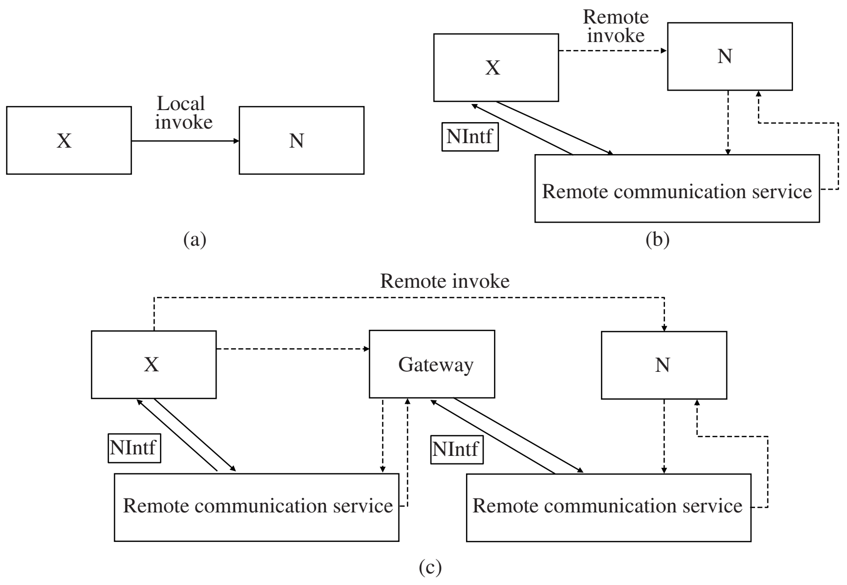
Local
invoke
X
N
(a)
Remote
invoke
X
N
NIntf
Remote communication service
(b)
Remote invoke
X
Gateway
N
NIntf
Remote communication service
NIntf
Remote communication service
(c)

Figure 3 Patterns of method invocation in MEC. (a) Local invocation (source structure); (b) direct remote invocation (source structure); (c) indirect remote invocation (source structure).

extend our previous study DPartner to refactor applications for adhering to the design pattern. Finally, we describe the mechanism to support the design pattern.

3.1 The design pattern

The execution of an Android application could be abstracted as invocations of methods from different classes. In MEC, class methods need to be executed on different computation nodes according to the device context. In this study, we assume a network environment of MEC, where (1) all the edges are always connected to the cloud, (2) mobile devices are always connecting to the cloud, and (3) mobile devices could be connected to nearby edges such that connections between the devices and the edges are likely to be broken when devices shift. Under such an assumption, we can derive three key patterns of method invocation in MEC.

First, when both the classes are deployed on the device or the same server, they can interact with each other locally, as shown in Figure 3(a). Class X first obtains the local reference of class N and then invokes the methods of N.

Secondly, when one class is deployed on the device and the other on a connected server (mobile edge or cloud), they can interact with each other remotely, as shown in Figure 3(b). Class X obtains a remote reference to N from a remote communication service and then uses the reference to interact with N remotely. The remote communication service is responsible for associating N's reference with N across the network.

Thirdly, when the device shifts from one place to another, the two classes may be deployed on the device and an unconnected mobile edge separately. Thus, they need to interact with each other across a gateway in the cloud, as shown in Figure 3(c). Class X sends message to the gateway. The gateway obtains a remote reference to N from a remote communication service, forwards the request to N, and then returns the result to X.

In order to support such patterns of method invocation, we propose a design pattern for on-demand computation offloading, as shown in Figure 4. The core of the design pattern is composed of two elements: proxy and interceptor. A proxy, NProxy in Figure 4, functions similarly as the proxied class N except that it does not perform any computation. If the location of the proxied class N is changed, NProxy remains unchanged so that the caller, class X, will not be observed.

The interceptor is responsible for determining the current location of N and for forwarding the method invocation to N locally or to another NProxy remotely. When NProxy and N both run on the same

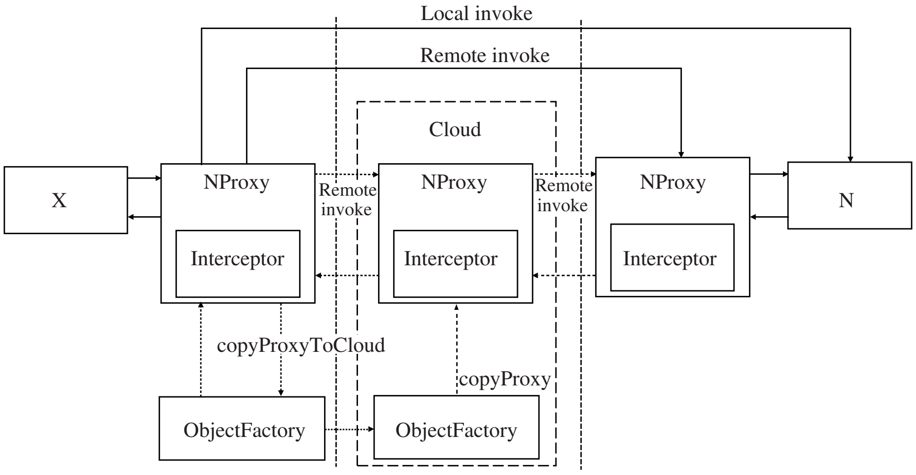

Figure 4 On-demand invocation (target structure).

```
//Source code of object creation
Person person = new Person();
Position position = person.
getPosition();
```

```
//Target code of object creation
PersonIntf person =
ObjectFactory.create(Person.class);
PositionIntf position = person. getPosition()
```

```
//Source code of array creation and array access
Person[] persons = new Person[10];
Person tempPerson = persons[0];
persons[1] = tempPerson;
```

```
//Target code of array creation and array access
Array<PersonIntf> persons =
ObjectFactory.create(Array.class, 10); PersonIntf
tempPerson = persons.get(0);
persons.set(1, tempPerson);
```

Figure 5 Treatment of object creation and array.

location, the interceptor obtains the local reference of N and invokes N directly. When NProxy and N are running on the device and a connected server, respectively, the interceptor will forward the method invocation to the other NProxy, which runs on the same location with N and invokes it directly. When NProxy and N are running on the device and an unconnected mobile edge, respectively, the NProxy will be copied to the cloud and the interceptor will forward the method invocation to the NProxy on the cloud; this in turn will forward the method invocation through the network stack.

The proxy and interceptor decouple the caller and callee. The object can be offloaded among the device, mobile edges, and cloud on demand. The interceptor automatically adapts to such changes so that the interacted classes are unaware of them.

3.2 Application refactoring

In order to enable a mobile application to be offloaded on demand, the program should follow certain coding guidelines.

(1) Movable and anchored classes. The classes of the application are divided into movable and anchored ones. The instances of the anchored ones need to use certain local resources; therefore, they cannot be offloaded to remote servers. On the contrary, the instances of the movable ones can be offloaded among the device, mobile edges, and the cloud.

(2) Class structure. A proxy (implemented as an interface class) and a serializable method are required for any class of the application. For those non-private fields, their modifier needs to be set private, and public getter/setter methods are required for them; thus, the caller classes that get/set the fields should use the corresponding getter/setter methods.

(3) Creation of objects and arrays. As shown in Figure 5, the ObjectFactory is responsible for generating class instances; this is detailed in Subsection 3.3. Moreover, the array interface handles creation and access of arrays.

In our previous study [21], we implemented a tool called DPartner[3] to automatically execute the refactoring steps; it first analyzes the bytecode of the application and then rewrites the bytecode to implement a special program structure. For a specified Android application, we use the DPartner tool to refactor the application into the one adhering to the above-mentioned coding guidelines.

3.3 Computation offloading at runtime

The offloading of computations is implemented as remotely deploying and invoking a single class instance or a set of class instances performing computations.

3.3.1 *Object creation and offloading*

We design the ObjectFactory to handle local and remote creation of an object or a set of objects. The "create" method of ObjectFactory is used to create the instance of class N. The parameters of the "create" method include (1) the class N's complete qualified class name, (2) the target location of the instance to be created, and (3) the constructor parameter values of the class instance.

When the target location is the local host, the ObjectFactory directly creates the instance and its ID and then uses the class name and instance ID to generate the proxy. In addition, the ObjectFactory maintains the hash table ⟨ID, Object Reference⟩ for each local instance.

When the target location is a remote server, the local ObjectFactory serializes the parameters and transfers them to the remote ObjectFactory. Then, the remote ObjectFactory deserializes the parameters and creates the instance as well as its ID and proxy on the remote server. Subsequently, the ID is returned, and the local ObjectFactory uses it to generate the proxy.

ObjectFactory is also used to handle the offloading of an object or a set of objects. The "offload" method of ObjectFactory is used to offload the instance of class N. The parameters of the "offload" method include (1) the class N's complete qualified class name, (2) the instance ID of class N, and (3) the target location of the instance to be offloaded.

The steps of object offloading include: (1) the instance and its ID are serialized on the original node and transferred to the target node; (2) the new instance and its proxy are created on the target node; and (3) the hash tables of ObjectFactory on both original and target nodes are updated.

3.3.2 *Method invocation*

Each proxy object contains an interceptor, which is used to handle the actual crossing network communication between objects, as shown in Figure 6. A method invocation from class X to N is passed first from X to NProxy locally, then to the interceptor. In a method body of NProxy, the "invokeMethod" method of the interceptor is used to forward the method invocation to the maybe-offloaded class N. The parameters of "invokeMethod" are: (1) the class N's full qualified class name; (2) the instance ID of class N; (3) the byte-code-level method signature; (4) the parameters required for the method of N being actually executed.

When X and N are both located in the same place, the interceptor will directly use the object reference to forward method invocation from X to N.

When X and N are respectively running on two connected nodes, the interceptor will serialize the parameters and transfer them to the NProxy on the connected node. And the remote NProxy deserializes the parameters and uses the object reference to forward method invocation to N locally.

When X and N are running on two unconnected nodes separately, the interceptor cannot directly forward method invocation through the network stack. In this case, the "copyProxy" method of Object-Factory is used to copy the proxy onto the cloud. The proxy and the instance location are serialized and transferred to the ObjectFactory on the cloud. The ObjectFactory on the cloud deserializes the proxy and records the instance location in its hash table. Then, the local ObjectFactory modifies the information of the instance location to the cloud. Thus, when a method is invoked, the interceptor first

3) DPartner. http://code.google.com/p/dpartner/.

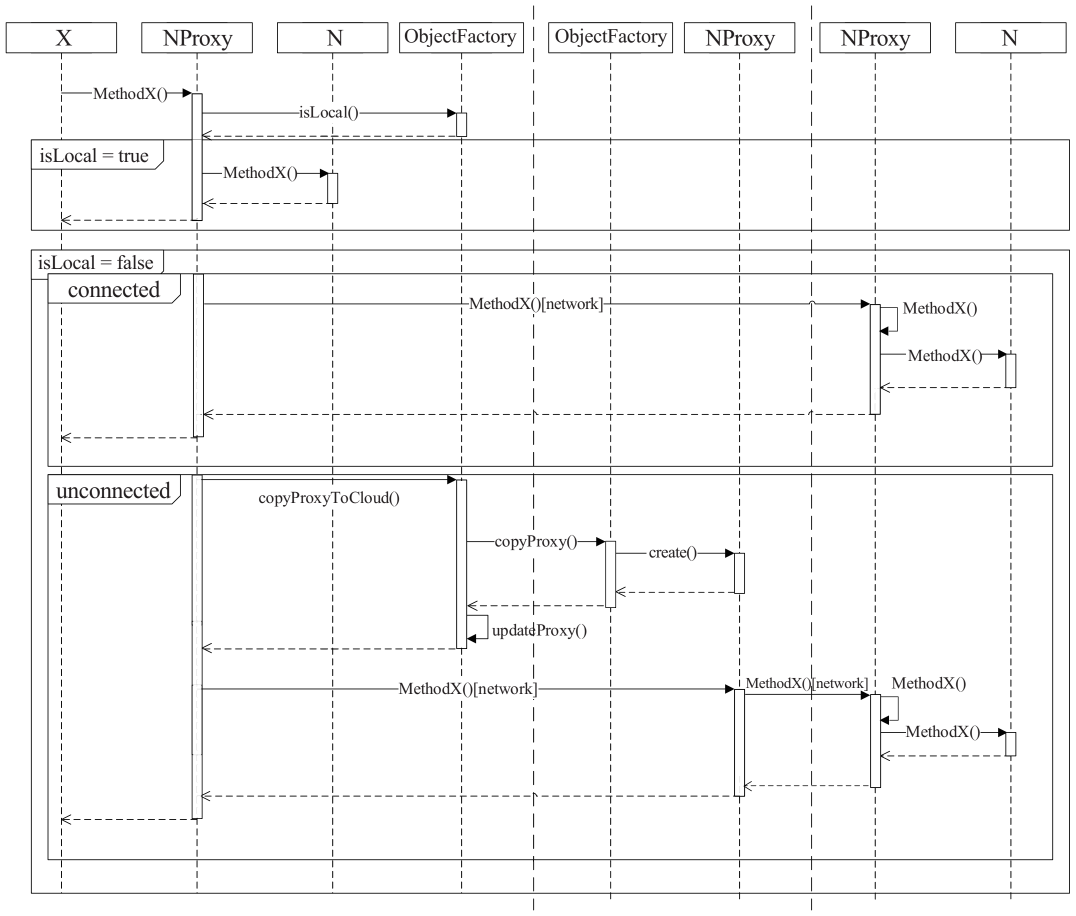

Figure 6 Handling method invocation.

forwards the method invocation to the proxy on the cloud, which then forwards the method invocation to the instance through the network stack.

4 Estimation model for offloading scheme

Android applications consist of activities, with a main activity functioning as an entry point for the application. For each execution of an application, the class methods in the main activity are invoked, which uses the data and methods internal to the same class or invokes certain methods of other classes. Such a process can be characterized by a control flow graph. In order to improve the offloading performance, it needs to cluster the frequently interacted objects and offload them as a whole. As a result, we consider the objects created in the main activity as the core objects; moreover, the other objects, which are created in the function bodies of core objects, should be offloaded with their related core objects as a whole. Thus, the problem of determining the computation offloading scheme could be reduced to solving the problem of an object deployment plan for each class in the main activity of the application.

There are likely to be several method invocations for each class. Furthermore, the response time for each method invocation consists of the execution time and the network delay, as shown in (1). Thus, there are numerous factors that can influence the object deployment plan, such as method invocations, remote servers, and network connections (Table 1).

$$T_{\text{response}} = T_e + T_d. \tag{1}$$

First, we use ASM[4)] to generate the control flow graph of the main activity and retrieve the core movable

4) http://asm.ow2.org/.

Table 1 List of factors that can influence object deployment plan

Symbol	Description
C	The set of method invocations for a movable class c_i
Task_i	The kth method invocation for the core movable class c_i
t_i^k	The kth method invocation for the core movable class c_i
N	The set of device, cloud and mobile edges
n_i	The server n_i
V	The data transmission rates between the device and remote servers
v_i	The data transmission rate between the device and the remote server n_i
RTT	The round trip times between the device and remote servers
rtt_i	The round trip time between the device and the remote server n_i

classes and their method invocations. Let $C = \{c_0, c_1, \ldots, c_n\}$ denote the core movable classes and $\mathrm{Task}_i = \{t_i^1, t_i^2, \ldots, t_i^m\}$ denote the method invocations for each core movable class c_i in the main activity.

Secondly, we maintain the information on the servers and network connections from the device context. Let $N = \{n_0, n_1, \ldots, n_z\}$ denote the device, cloud, and mobile edges, and let $V = \{v_0, v_1, \ldots, v_z\}$ and $\mathrm{RTT} = \{\mathrm{rtt}_0, \mathrm{rtt}_1, \ldots, \mathrm{rtt}_z\}$ denote the data transmission rates and round trip times between the device and remote servers.

Thirdly, we introduce information models [22] to collect history data regarding the average data traffic for each method invocation, as well as their execution time on the device, cloud, and each mobile edge. Let $T_e(t_i^k, n_t)$ denote the execution time of the method invocation t_i^k on the server n_t. Let $D(t_i^k)$ represent the average data traffic for the method invocation t_i^k. The network delay of the method invocation t_i^k on the server n_t can be calculated by

$$T_d(t_i^k, n_t) = \frac{D(t_i^k)}{V_{n_t}} + \mathrm{rtt}_{n_t}. \tag{2}$$

Thus, the response time for the method invocation t_i^k on the server n_t can be calculated by

$$T_{\mathrm{response}}(t_i^k, n_t) = T_e(t_i^k, n_t) + T_d(t_i^k, n_t). \tag{3}$$

Based on the models, we construct the fitness function to evaluate the offloading scheme, i.e., the object deployment plan for each core class, as shown in (4). Moreover, a more effective scheme obtains a smaller value of fitness function. Thus, we can determine the optimal deployment location for each class in the main activity of the application.

$$T_{\mathrm{response}}(c_i, n_t) = \sum_{k=1}^{m} (T_{\mathrm{response}}(t_i^k, n_t)). \tag{4}$$

5 Adaptive offloading framework

We have implemented the framework to support the design pattern and the estimation model, which is deployed on the device, mobile edges and cloud. Figure 7 shows the architecture of the framework.

For a specified Android application, the code is refactored to be capable of computation offloading; this adheres to the design pattern mentioned in Section 3. The ObjectFactory handles object creation and offloading, and the Proxy addresses object method invocation.

The code is also analyzed to obtain information on the method invocations of the movable classes in the main activity; this is used for modeling core movable classes. The tasks, servers, and network connections are also modeled. Based on the models, the decision module calculates the object deployment plan for each core movable class according to the device context.

6 Evaluation

In this section, we set up a mobile edge environment and use two real-world applications to evaluate our adaptive offloading. First, we validate whether our framework is feasible for offloading real-world

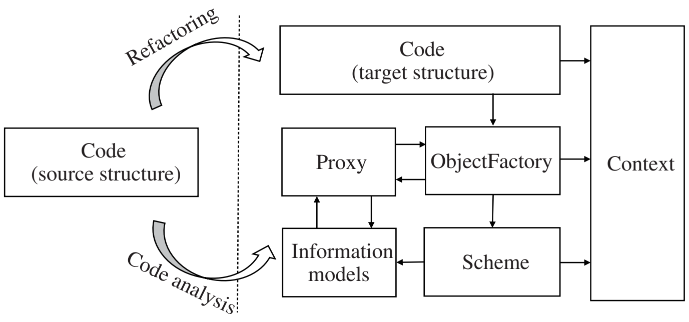

Figure 7 The core of adaptive offloading framework.

Table 2 The device contexts in different locations

	Garden	Playground	Teaching building	Laboratory building
E1	–	RTT=30 ms, $V = 1$ Mb/s	RTT = 30 ms, $V = 1$ Mb/s	–
E2	–	–	RTT = 60 ms, $V = 700$ Kb/s	RTT = 35 ms, $V = 950$ Kb/s
Cloud	RTT = 200 ms, $V = 150$ Kb/s	RTT = 200 ms, $V = 150$ Kb/s	RTT = 200 ms, $V = 150$ Kb/s	RTT = 55 ms, $V = 725$ Kb/s

applications in MEC. Secondly, we compare the performance of the adaptive offloaded applications with the original and the traditionally offloaded ones, in different scenarios. Finally, we demonstrate the energy saving enabled by adaptive offloading.

6.1 Experiment setup

Our experimental environment consists of five computation nodes: two mobile devices and three remote servers. We simulate four locations, which are named garden, playground, teaching building, and laboratory building, as shown in Figure 2. The network conditions between the mobile devices and the remote servers in each location are described in Table 2.

The two mobile devices are (1) Honor MYA-AL10[5] with a 1.4-GHz four-core CPU and 2-GB RAM, representing a low-end device and (2) Honor STF-AL00[6] with a 2.4-GHz four-core CPU and 4-GB RAM, representing a high-end device.

The three remote servers include two mobile edges (E1 and E2) and a cloud, which can be used for offloading in different locations. E1 is a server with a 2.5-GHz eight-core CPU and 4-GB RAM, whose network covers the playground and the teaching building. E2 is a server with a 3.0-GHz eight-core CPU and 8-GB RAM, whose network covers the teaching building and laboratory building. The cloud is a server with a 3.6-GHz 16-core CPU and 16-GB RAM, which can be publicly accessed from all the locations.

We use two real Android applications in the evaluation. One is a license plate recognition system (LRS), which uses an image recognition technique to gather car information from a video stream. The other is a voice recognition system (VRS), which uses a voice recognition technique to turn speech into text. We refactor the two applications based on our design pattern and gather information regarding the execution time on the computation nodes. Table 3 presents the movable classes in the main activities of these applications and their method invocations; it also presents the average data traffic and execution time for each method invocation on each computation node.

6.2 Offloading validation

We apply our framework to support adaptive computation offloading on the two real-world mobile applications. Then, we use the estimation model to determine the offloading scheme of each application

5) Honor MYA-AL10. http://detail.zol.com.cn/cell_phone/index1174655.shtml.
6) Honor STF-AL00. http://detail.zol.com.cn/1168/1167243/param.shtml.

Table 3 Details of the two applications for evaluation

Application	Movable class	Method	Number of method invocation	Data traffic	Response time (ms)				
					Honor MYA-AL10	Honor STF-AL00	E1	E2	Cloud
License plate recognition system	Framing	getFrame	1	3.5 MB	248	153	74	56	20
	Prepro-cessing	colorKMean	1	450 KB	1791	1447	835	464	210
		oritenation	1	450 KB	2667	2201	1286	618	178
	OCRPro-cessing	segInEachChar	3	10 KB	66	51	33	24	8
		clearSmall	3	1 KB	62	50	36	20	13
		getRegion	3	1 KB	48	37	29	21	6
		zoom	3	1KB	24	15	8	4	2
		recEachCharInMinDis	1	16 KB	1850	1074	708	443	222
Voice recognition system	SpeechRe-cognizer-setup	defaultSetup	1	5 KB	65	40	32	14	8
		setConfig	4	1 KB	257	184	93	64	25
		getRecognizer	1	1 KB	161	122	77	49	16
	SpeechRe-cognizer	getDecoder	1	2 KB	312	212	148	74	41
		startListening	1	415 B	71	50	37	20	8
		stopListening	1	372 B	53	31	16	9	5
	Decoder	startUtt	1	320 B	98	74	55	34	19
		processRaw	1	712 KB	1910	1333	712	423	141
		getInSpeech	2	20 B	45	22	14	11	8
		hyp	2	24 B	59	38	17	9	5
		endUtt	1	146 B	102	79	57	27	10

Table 4 Offloading schemes

Application	Device	Movable class	Location			
			Garden	Playground	Teaching building	Laboratory building
License plate recognition system	Honor MYA-AL10	Framing	Local	Local	Local	Local
		Preprocessing	Local	E1	E2	Cloud
		OCRProcessing	Local	E1	E1	E2
	Honor STF-AL00	Framing	Local	Local	Local	Local
		Preprocessing	Local	E1	E2	Cloud
		OCRProcessing	Local	Local	Local	E2
Voice recognition system	Honor MYA-AL10	SpeechRecognizerSetup	Local	E1	E1	Cloud
		SpeechRecognizer	Local	E1	E2	E2
		Decoder	Local	E1	E1	Cloud
	Honor STF-AL00	SpeechRecognizerSetup	Local	E1	E1	Cloud
		SpeechRecognizer	Local	Local	E2	E2
		Decoder	Local	Local	Local	Cloud

for each device at different locations. Table 4 presents the results. We observe numerous factors that impact the offloading schemes. First, the quality of network connection impacts the offloading schemes significantly. For example, while using the Honor MYA-AL10 to run the LRS, there are differences between the offloading schemes at different locations. Secondly, the computation complexity of the class methods can impact the offloading schemes. For example, while using the Honor MYA-AL10 to run the LRS in the Laboratory Building, the Framing, Preprocessing, and OCRProcessing class instances run separately on the device, Cloud, and E2. Thirdly, the processing power of the mobile devices also impacts the offloading schemes. For example, while running the LRS in the Playground and the Teaching Building, the OCRProcessing class instances are offloaded to E1 for the Honor MYA-AL10, whereas they run locally for the Honor STF-AL00.

Subsequently, we run the offloaded application considering two scenarios: (1) the mobile devices stay at a fixed location where computations are offloaded to the mobile edge and the cloud according to the offloading scheme and (2) the mobile devices shift from the garden to the playground, then to the teaching building, and finally the laboratory building, whereas the computation offloading changes from

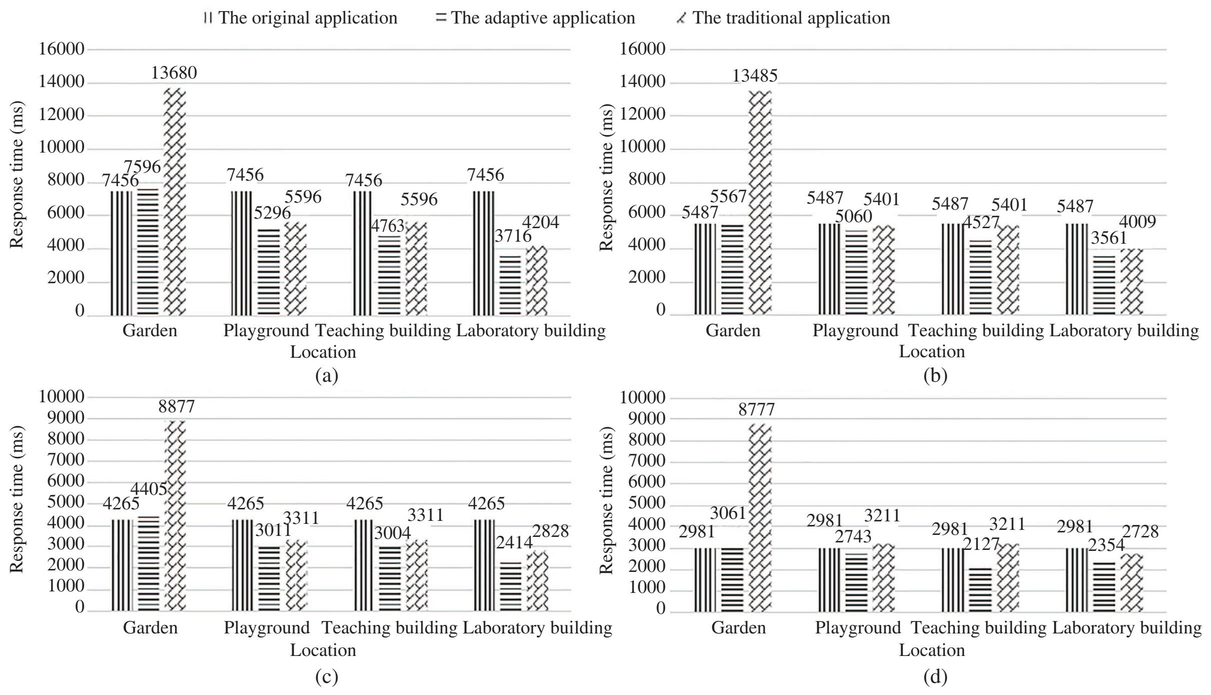

Figure 8 Performance comparison of running license plate recognition system and voice recognition system between four locations. (a) Honor MYA-AL10 -license plate recognition system; (b) Honor STF-AL00 -license plate recognition system; (c) Honor MYA-AL10 -video recognition system; (d) Honor MYA-AL10 -video recognition system.

the scheme in the source location to the one in the target location. In all the cases, both the applications can execute successfully and correctly as the original ones. This result demonstrates the feasibility of our offloaded framework.

6.3 Performance improvement

We evaluate the performance improvement achieved by our adaptive offloading framework based on a comparison with the original, the traditional offloaded, and the adaptive offloaded applications. The original application runs entirely on the device; the traditional offloaded application offloads movable objects with intensive computation to the remote server with the most advanced network connection; the adaptive offloaded application can use any remote server for on-demand offloading. We also consider scenarios: (1) when the devices stay at different locations and (2) when the devices shift between locations. Moreover, for the second scenario, the mobility model of mobile devices is a shift from the garden to the playground, then to the teaching building, and finally to the laboratory building. We use the response time as the metric of performance.

6.3.1 *Staying in different locations*

Figure 8 shows a performance comparison between the two applications when run on each phone in the four locations separately. We observe that the response time of the adaptive offloaded applications is smallest or close to the smallest in all the cases.

Compared with the original applications, our adaptive offloaded ones can reduce the response time by 8%–50% when the mobile device has a good network connection in the playground, teaching building, or laboratory building; this is because certain compute-intensive tasks are offloaded to mobile edges or the cloud. On the contrary, when the mobile device has an ineffective network connection in the garden, our adaptive offloaded applications run locally similar to the original ones; moreover, their performance is close to that of the original ones, albeit with an overhead of approximately 140 ms, which originates from the proxies.

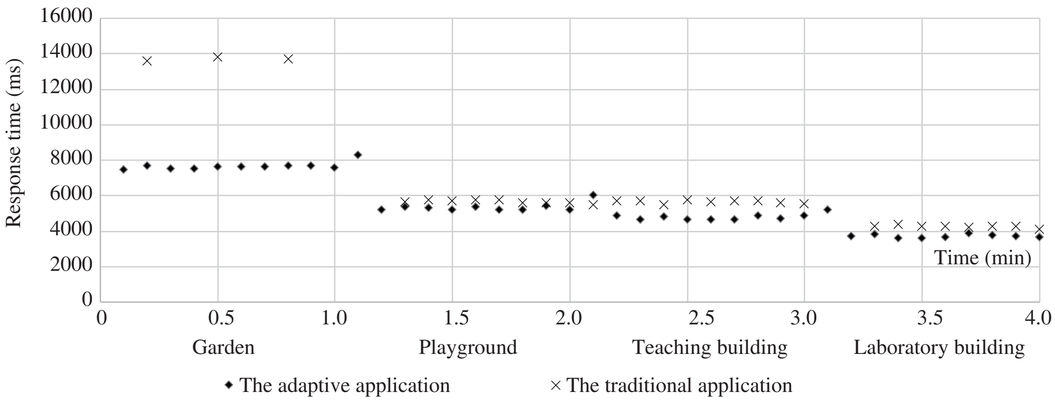

Figure 9 Performance comparison between running license plate recognition system on the Honor MYA-AL10 device shifting between four locations.

Compared with traditional offloaded applications, our adaptive offloaded ones can reduce the response time by 5%–33% when offloading computation to remote servers. The reason for the performance improvement is the large number of factors impacting the offloading effects, such as the quality of network connection, computation complexity of class methods, and processing power of device and remote servers; moreover, our adaptive offloaded applications determine the offloading scheme dynamically based on the trade-off between the execution time and network delay, rather than using the remote server with the best network connection for offloading.

6.3.2 *Moving between different locations*

Figure 9 shows the performance comparison of running the LRS on Honor MYA-AL10 when we shift between four locations, remaining in each location for 1 min.

The traditional offloaded application is likely to crash when the device enters a new location. For example, the application fails to run for 20 s when the device enters the playground. The reason for the application crash is that when the device context changes, the objects, which have been offloaded to the original mobile edge, cannot be accessed any longer from the device.

Our adaptive offloaded application remains available when the device context changes, although the response time is marginally longer when the device enters a new location. For example, the response time of the first invocation in the teaching building is 6039 ms, whereas the average time for all the invocations except the first is 4860 ms. On the one hand, our framework supports remote object access between the device and mobile edges via the cloud. On the other hand, when the device context changes, our framework can automatically determine the offloading scheme and then offload computation in a short time.

6.4 Energy saving

We measure the energy consumption by PowerTutor[7], which can provide the details of the energy consumption for each target application during a period of time. We executed the application ten times in each location, used PowerTutor to determine the battery's energy consumption, and calculated the average. Figure 10 shows the results. We observe that the energy consumption of the adaptive offloaded applications is the smallest or close to the smallest in all the cases.

When the mobile device has an ineffective network connection in the garden, the energy consumption of our adaptive is close to those of the original ones; however, it incurs an overhead of approximately 150 mJ because the proxies consume additional energy.

When the mobile device has a good network connection in the playground, teaching building, or laboratory building, our adaptive offloaded applications can reduce the energy consumption by 9%–

7) PowerTutor. http://powertutor.org/.

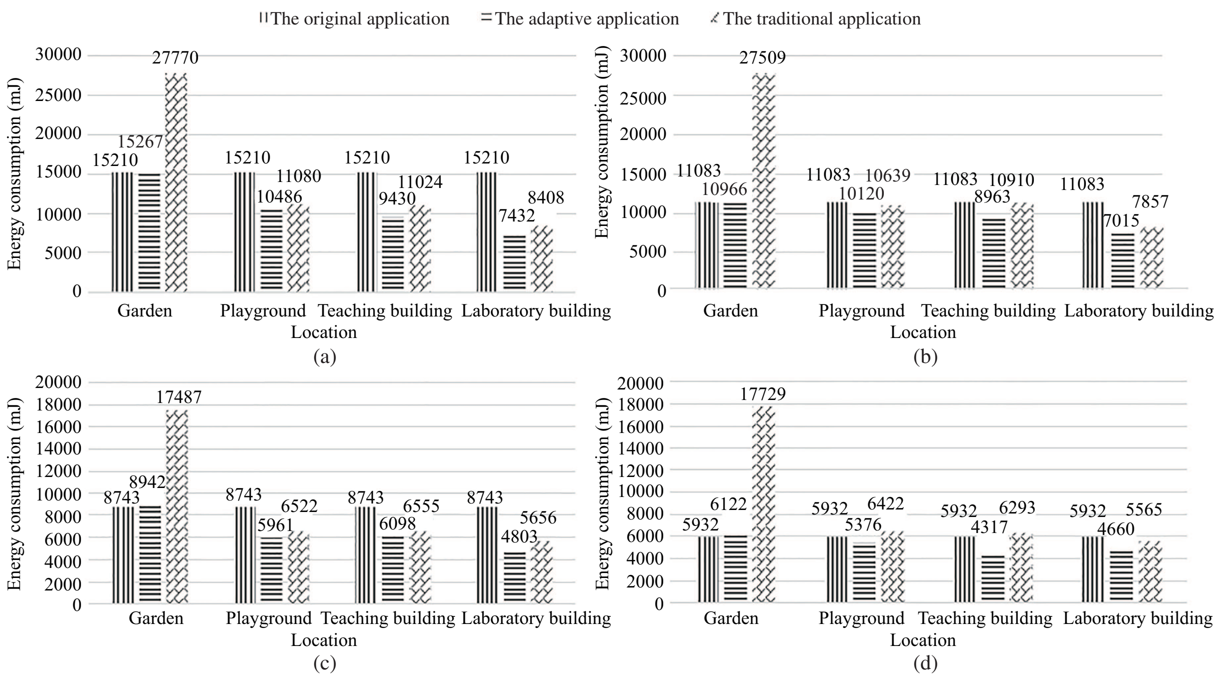

Figure 10 Energy consumption comparison of running license plate recognition system and voice recognition system at four locations. (a) Honor MYA-AL10 -license plate recognition system; (b) Honor STF-AL00 -license plate recognition system; (c) Honor MYA-AL10 -video recognition system; (d) Honor STF-AL00 -video recognition system.

51% compared with those of the original ones. The energy saving results from the fact that certain computation-intensive tasks are offloaded to mobile edges or the cloud.

7 Discussion

This study mainly focuses on supporting Android applications with the offloading capability in MEC; the experiment results demonstrate the effectiveness of our approach. However, some issues remain that is likely to influence the applicability of our proposed framework, including the application styles, network conditions, and real-world environment.

7.1 Application styles

Our framework focuses on Android applications, which are developed in the object-oriented language Java. Therefore, both the design pattern and estimation model are aimed at supporting the mobile applications in an object-oriented style. Thus, the adaptive framework can support only object-oriented mobile applications, with the offloading capability. However, non-OO applications such as DNN-based applications can also benefit from our approach. On the one hand, the three key patterns of invocation (Subsection 3.1) between two executed units such as objects and DNN layers are common for different application styles. In order to support similar patterns of invocation between two executed units, we propose the design pattern for on-demand computation offloading in MEC; it is composed of two elements: proxy and interceptor. Although the behaviors of an interceptor are dependent on the application style, the mechanism of proxy is independent of the application style and can be reused in other ones. On the other hand, the estimation model is designed to automatically determine the offloading scheme. We introduce information models to collect history data regarding the average data traffic for each invocation between two executed units as well as their execution time on the device, the cloud, and each mobile edge; based on this, we propose an algorithm to determine the optimal deployment location for each unit. The trade-off between the reduced execution time and the network delay is universal in different application styles, and the mechanism of the estimation model can be also reused in other ones.

7.2 Network conditions

In this study, we assume a network environment of MEC where mobile devices are continuously connected to the cloud. Under such an assumption, our framework supports remote object access between the device and mobile edges via the cloud. Therefore, our adaptive offloaded application can continue to operate when the device context changes. However, if the assumption cannot be guaranteed in certain cases (i.e., the mobile devices cannot connect to the cloud) and the mobile edges cannot be connected either, the offloaded application needs to be restarted to resume its processing logic; this is because the objects that have been offloaded to the mobile edge or the cloud are no more accessible from the device. In this case, the offloaded application is likely to fail to operate for a short time (several seconds) when the network is disconnected. We could leverage the object synchronization mechanism to address the network disconnections; we intend to address this in our future work.

7.3 Real-world environment

In our experiment, we set up a network environment to simulate the real-world environment to the maximum extent feasible. For example, the two mobile devices separately represent low-end and high-end devices, and the network conditions between the mobile devices and the remote servers are diff with the locations. The results reveal the effectiveness of our framework clearly. The differences between our simulated environment and the real-world environment are that (1) the application is running in a single-user environment, so that the execution time for each invocation of the same class method on the same computation node is generally close to their average and (2) the mobility model of the mobile devices is not complex, so that the network conditions between the mobile devices and the same remote server in the same location are generally close to their average. Therefore, our framework can still operate in the real-world environment, although the performance improvement may be marginally different. However, this study mainly focuses on supporting Android applications with the offloading capability in MEC; the two issues above are orthogonal to the problem in this study. In addition, a few related studies can be introduced to enhance our framework, such as supporting multi-user cases via game-theoretic model [23, 24] and supporting complex mobility models via other offloading decision algorithms [25, 26].

8 Related work

Computation offloading is established to be an effective method [27–30] for improving performance as well as for reducing the energy consumption of a mobile application, in MCC [16–20]. Cuckoo [27] is an offloading framework for Android applications. It requires developers to follow a specific programming model in order for certain parts of the application to be offloaded. MAUI [28] requires developers to annotate the can-be-offloaded methods of a .Net mobile application by using the “Remoteable” annotation. Then, it decides which method should be offloaded through runtime profiling. CloneCloud [30] provides a flexible application partitioner and execution runtime. Moreover, it enables unmodified mobile applications running in an application-level virtual machine to seamlessly offload part of their execution from mobile devices onto device clones operating in a computational cloud. ThinkAir [29] focuses on the elasticity and scalability of the cloud and enhances the power of MCC by parallelizing method execution using multiple virtual machine images. These studies mainly focus on offloading mechanisms, in which applications use only the cloud for offloading.

A few other studies focus on offloading strategies [26, 31–33]. Zhou et al. [31] proposed a prototype MCC offloading system that considers multiple cloud resources such as mobile ad-hoc network, cloudlet, and public clouds to provide an adaptive MCC service. Furthermore, they proposed a context-aware offloading decision algorithm aimed at providing a code offloading decision regarding the selection of the wireless medium and of the potential cloud resources as the offloading location based on the device context, during runtime. Cheng et al. [32] presented a novel three-layer architecture consisting of wearable devices, mobile devices, and remoted cloud for code offloading; they investigated an offloading strategy with a novel just-in-time objective and proposed an efficient algorithm based on genetic algorithm (GA) to solve it. Jin et al. [33] focused on decision making of multisite computation offloading in dynamic mobile cloud environments, considering environmental changes; they presented a runtime application repartitioning algorithm based on memory-based immigrants adaptive GA. Wang et al. [26] proposed combination optimization to facilitate mobile data traffic offloading in emerging vehicular cyber-physical systems to reduce the amount of mobile data traffic for QoS-aware service provision. These studies mainly focus on adaptive offloading decision algorithms, most of which are based on program high-level abstraction, rather than real-world applications.

Our previous study, DPartner [21], can automatically refactor any available Android application into the one supporting on-demand offloading in MCC; based on this, DelayDroid [34] can automatically refactor Android applications and expose the opportunity for batching network requests to developers. This study focuses on supporting Android applications with the offloading capability in MEC. It differs from previous study in two aspects. First, our framework enables an application to remain available when the mobile device shifts from one location to another and the mobile edge used for offloading gets disconnected. However, few available studies can support such a feature. Secondly, our framework analyzes static code and collects runtime data in order to determine the offloading scheme; meanwhile, most of the available study only establishes adaptive offloading decision algorithms based on program high-level abstraction.

9 Conclusion

This paper proposed an adaptive offloading framework for Android applications in MEC. For a specified Android application, the framework first refactors the application to implement a special design pattern and then analyzes static code of the application to obtain information on movable classes. Thus, it can determine the offloading scheme during runtime according to the device context and enable applications to be offloaded between the device, mobile edges, and the cloud dynamically. We evaluate our framework for two real-world applications, and the results demonstrate that our approach can significantly improve the performance and reduce energy consumption.

Acknowledgements This work was supported by National Key R&D Program of China (Grant No. 2017YFB1002000), National Natural Science Foundation of China (Grant No. 61725201), and Talent Program of Fujian Province for Distinguished Young Scholars in Higher Education. Yun MA's work was supported by China Postdoctoral Science Foundation.

References

1 Berglund M E, Duvall J, Dunne L E. A survey of the historical scope and current trends of wearable technology applications. In: Proceedings of ACM International Symposium on Wearable Computers, Heidelberg, 2016. 40–43

2 Yang F C, Li J L, Lei T, et al. Architecture and key technologies for Internet of vehicles: a survey. J Commun Inf Netw, 2017, 2: 1–17

3 Li P, Yu X, Peng X Y, et al. Fault-tolerant cooperative control for multiple UAVs based on sliding mode techniques. Sci China Inf Sci, 2017, 60: 070204

4 Mei H, Liu X Z. Software techniques for Internet computing: current situation and future trend. Chin Sci Bull, 2010, 55: 3510–3516

5 Yang K, Ou S M, Chen H H. On effective offloading services for resource-constrained mobile devices running heavier mobile Internet applications. IEEE Commun Mag, 2008, 46: 56–63

6 Paradiso J A, Starner T. Energy scavenging for mobile and wireless electronics. IEEE Pervasive Comput, 2005, 4: 18–27

7 Kumar K, Lu Y H. Cloud computing for mobile users. Computer, 2011, 43: 51–56

8 Goyal S, Carter J. A lightweight secure cyber foraging infrastructure for resource-constrained devices. In: Proceedings of Mobile Computing Systems and Applications, Windermere, 2004. 186–195

9 Balan R, Flinn J, Satyanarayanan M, et al. The case for cyber foraging. In: Proceedings of ACM Sigops European Workshop, Saint-Emilion, 2002. 87–92

10 Balan R K, Gergle D, Satyanarayanan M, et al. Simplifying cyber foraging for mobile devices. In: Proceedings of International Conference on Mobile Systems, Applications and Services, San Juan, 2007. 272–285

11 Balan R K, Satyanarayanan M, Park S Y, et al. Tactics-based remote execution for mobile computing. In: Proceedings of International Conference on Mobile Systems, Applications, and Services, San Francisco, 2003. 273–286

12 Gu X H, Nahrstedt K, Messer A, et al. Adaptive offloading inference for delivering applications in pervasive computing environments. In: Proceedings of IEEE International Conference on Pervasive Computing and Communications, Fort Worth, 2003. 107

13 Kumar K, Liu J B, Lu Y H, et al. A survey of computation offloading for mobile systems. Mobile Netw Appl, 2013, 18: 129–140

14 Philippsen M, Zenger M. JavaParty - transparent remote objects in Java. Concurrency-Pract Exper, 1997, 9: 1225–1242

15 Hunt G C, Scott M L. The coign automatic distributed partitioning system. In: Proceedings of Enterprise Distributed Object Computing Workshop, La Jolla, 1999. 252–262

16 Shi W S, Cao J, Zhang Q, et al. Edge computing: vision and challenges. IEEE Internet Things J, 2016, 3: 637–646

17 Chiang M, Zhang T. Fog and IoT: an overview of research opportunities. IEEE Internet Things J, 2016, 3: 854–864

18 Tran T X, Hajisami A, Pandey P, et al. Collaborative mobile edge computing in 5G networks: new paradigms, scenarios, and challenges. IEEE Commun Mag, 2017, 55: 54–61

19 Abbas N, Zhang Y, Taherkordi A, et al. Mobile edge computing: a survey. IEEE Internet Things J, 2018, 5: 450–465

20 Wang S G, Xu J L, Zhang N, et al. A survey on service migration in mobile edge computing. IEEE Access, 2018, 6: 23511–23528

21 Zhang Y, Huang G, Liu X Z, et al. Refactoring Android Java code for on-demand computation offloading. SIGPLAN Not, 2012, 47: 233

22 Chen X, Chen S H, Zeng X, et al. Framework for context-aware computation offloading in mobile cloud computing. J Cloud Comp, 2017, 6: 1

23 Chen X, Jiao L, Li W Z, et al. Efficient multi-user computation offloading for mobile-edge cloud computing. IEEE/ACM Trans Netw, 2016, 24: 2795–2808

24 Chen X. Decentralized computation offloading game for mobile cloud computing. IEEE Trans Parallel Distrib Syst, 2015, 26: 974–983

25 Lei T, Wang S G, Li J L, et al. AOM: adaptive mobile data traffic offloading for M2M networks. Pers Ubiquit Comput, 2016, 20: 863–873

26 Wang S G, Lei T, Zhang L Y, et al. Offloading mobile data traffic for QoS-aware service provision in vehicular cyber-physical systems. Future Gener Comput Syst, 2016, 61: 118–127

27 Kemp R, Palmer N, Kielmann T, et al. Cuckoo: a computation offloading framework for smartphones. In: Proceedings of International Conference on Mobile Computing, Applications, and Services, Santa Clara, 2010. 59–79

28 Cuervo E, Balasubramanian A, Cho D K, et al. Maui: making smartphones last longer with code offload. In: Proceedings of International Conference on Mobile Systems, Applications, and Services, San Francisco, 2010. 49–62

29 Kosta S, Aucinas A, Hui P, et al. Thinkair: dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In: Proceedings IEEE INFOCOM, Orlando, 2012. 945–953

30 Chun B G, Ihm S, Maniatis P, et al. Clonecloud: elastic execution between mobile device and cloud. In: Proceedings of Conference on Computer Systems, Salzburg, 2011. 301–314

31 Zhou B W, Dastjerdi A V, Calheiros R N, et al. A context sensitive offloading scheme for mobile cloud computing service. In: Proceedings of IEEE International Conference on Cloud Computing, Washington, 2015. 869–876

32 Cheng Z X, Li P, Wang J B, et al. Just-in-time code offloading for wearable computing. IEEE Trans Emerg Top Comput, 2015, 3: 74–83

33 Jin X M, Liu Y N, Fan W H, et al. Multisite computation offloading in dynamic mobile cloud environments. Sci China Inf Sci, 2017, 60: 089301

34 Huang G, Cai H Q, Swiech M, et al. DelayDroid: an instrumented approach to reducing tail-time energy of Android apps. Sci China Inf Sci, 2017, 60: 012106