Peer-Assisted Computation Offloading in Wireless Networks

Yeli Geng, *Student Member, IEEE*, and Guohong Cao, *Fellow, IEEE*

***Abstract*— Computation offloading has been widely used to alleviate the performance and energy limitations of smartphones by sending computationally intensive applications to the cloud. However, mobile devices with poor cellular service quality may incur high communication latency and high energy consumption for offloading, which will reduce the benefits of computation offloading. In this paper, we propose a peer-assisted computation offloading (PACO) framework to address this problem. In PACO, a client experiencing poor service quality can choose a neighbor with better service quality to be the offloading proxy. Through peer to peer interface such as WiFi direct, the client can offload computation tasks to the proxy which further transmits them to the cloud server through cellular networks. We propose algorithms to decide which tasks should be offloaded to minimize the energy consumption. We have implemented PACO on Android and have implemented three computationally intensive applications to evaluate its performance. Experimental results and simulation results show that PACO makes it possible for users with poor cellular service quality to benefit from computation offloading and PACO significantly reduces the delay and energy consumption compared to existing schemes.**

***Index Terms*— Energy consumption, cellular phones, computation offloading, wireless communication.**

I. Introduction

As MOBILE devices are becoming increasingly powerful, computationally intensive mobile applications such as image or video processing, augmented reality, and speech recognition have experienced explosive growth. However, these computationally intensive applications may quickly drain the battery of mobile devices. One popular solution to conserve battery life is to offload these computation tasks from mobile devices to resource-rich servers, which is referred to as *computation offloading* [1].

Previous research on computation offloading has focused on building frameworks that enable mobile computation offloading to software clones of smartphones in the cloud [2]–[4]. However, all these studies assume good network connectivity while neglecting real-life challenges for offloading through cellular networks. In cellular networks such as 3G, 4G and LTE, some areas have good coverage while others may not because of practical deployment issues. As a result, the wireless signal strength of a mobile device varies based on its location. Mobile users experiencing weak signal strength usually have low data-rate connections. Moreover, the data throughput depends on the traffic load in the area [5]. When the network connectivity is slow, it may incur higher communication latency and consume more energy to offload computation to the cloud. Therefore, mobile devices experiencing poor service quality (in terms of signal strength and throughput) may not benefit from computation offloading.

Some existing research has identified similar challenges for data transmission in cellular networks. Many studies propose to offload cellular traffic to WiFi network to save energy [6], [7]. Schulman *et al*. [8] propose to defer data transmissions to coincide with periods of strong signal to save energy. In QATO [9], data traffic is offloaded from nodes with poor service quality to neighboring nodes with better service quality to save energy and reduce delay. However, all these works focus on traffic offloading rather than computation offloading, and computation offloading decisions have to consider the delay and energy consumption of both computation execution and data transmission.

In this paper, we propose a Peer-Assisted Computation Offloading (PACO) framework to enable computation offloading in wireless networks, which is especially helpful for mobile devices suffering from poor service quality. In PACO, clients with poor service quality can identify neighbors with better service quality and choose one of them as the offloading proxy. Through peer to peer interfaces such as WiFi direct, clients can offload computation tasks to the proxy which actually handles the computation offloading to the server.

Although leveraging nearby devices to relay traffic has been studied in prior work, using them for computation offloading in cellular networks raises new challenges which have not been addressed. One main challenge is how to make offloading decisions, i.e., determining which tasks should be offloaded to minimize the energy consumption of the mobile devices. Existing research [2], [4] considers the trade-off between the energy saved by moving computation to the cloud and the energy spent on offloading the computation. However, they did not consider the special characteristics of cellular networks when making offloading decisions. After a data transmission, the cellular interface has to stay in the high-power state for some time which could consume a significant amount of additional energy (referred to as the *long tail* problem) [10]. This long tail problem makes it hard to decide whether to offload the computation.

Manuscript received March 17, 2017; revised August 26, 2017 and January 19, 2018; accepted April 4, 2018. Date of publication April 24, 2018; date of current version July 10, 2018. This work was supported by the National Science Foundation under Grant CNS-1421578 and Grant CNS-1526425. The associate editor coordinating the review of this paper and approving it for publication was K. Huang. *(Corresponding author: Yeli Geng.)*

The authors are with the School of Electrical Engineering and Computer Science, Pennsylvania State University, University Park, PA 16802 USA (e-mail: yzg5086@cse.psu.edu; gcao@cse.psu.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TWC.2018.2827369

1536-1276 © 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

We have implemented PACO on the Android platform. To evaluate its performance, we have developed three computational intensive applications. Experimental and simulation results show that PACO can significantly reduce the energy and delay compared to other computation offloading approaches.

The main contributions of this paper are as follows.

- We introduce the idea of leveraging peers with better service quality to enable computation offloading in cellular networks.
- We design a peer-assisted computation offloading framework to detect and utilize neighbors with better service quality to save energy. We also propose algorithms to determine whether a task should be offloaded or not.
- We have implemented the framework on the Android platform and have implemented three applications to validate its effectiveness.

The remainder of this paper is organized as follows: In Section II, we present related work. Section III provides background and motivation for peer-assisted computation offloading. We present a high-level description of the PACO architecture in Section IV. We describe the design of PACO in more details; i.e., the proxy selection mechanism in Section V, and the offloading decision algorithms in Section VI. Section VII evaluates PACO's performance. Finally, Section VIII concludes the paper.

II. Related Work

In this section, we review three categories of research related to our work.

A. Power Saving in Cellular Networks

In cellular networks, the wireless interface will stay in high-power states for a long time (i.e., the long tail problem) after a data transmission. Existing research [10], [11] has shown that a large amount of energy can be wasted due to this problem. As a result, many researchers proposed to defer data transmission [8] or to aggregate the network traffic to amortize the tail energy [10], [12].

B. Computation Offloading

Computation offloading has received considerable attention recently. Some previous work has focused on building frameworks that enable computation offloading to the remote cloud, such as MAUI [2], CloneCloud [4] and ThinkAir [3]. Other work [13]–[15] has focused on the offloading decisions, i.e., which tasks of an application should be offloaded, to improve performance or save energy of the mobile devices.

There have been some studies on computation offloading which aim to reduce the cellular communication cost by solving the long tail problem. Xiang *et al.* [16] proposed the technique of coalesced offloading, which coordinates the offloading requests of multiple applications to amortize the tail energy. Tong and Gao [17] proposed application-aware traffic scheduling in computation offloading to minimize energy and satisfy the application delay constraint. Geng *et al*. [18] designed offloading algorithms to minimize the energy consumption considering the long tail problem. However, all of them assume that mobile devices have good cellular network connectivity.

Some existing research has exploited peer-to-peer offloading in different networks. In [19]–[21], authors proposed to offload computation to neighboring nodes in disruption tolerant networks, wireless sensor networks and small-cell networks, respectively. In [22], D2D communication was exploited to enable offloading from mobile devices to other mobile devices in cellular networks. Furthermore, Jo *et al.* [23] proposed a heterogeneous mobile computing architecture to exploit resources from D2D communication-based mobile devices. However, all of them offload computation to neighboring mobiles, instead of the faraway cloud. Different from them, our work exploits one-hop D2D communication to leverage a neighboring mobile device only as a relay to offload to the cloud through cellular network.

While solving the connectivity problem by exploiting the D2D communication, we also consider the long tail problem in our proposed solution to better utilize the cellular resources. To the best of our knowledge, none of the previous work attempted to solve the long tail problem when exploiting the cooperative computation offloading. Our work is the first to solve both problems in mobile cloud computing.

C. Cellular Traffic Offloading

To deal with the traffic overload problem in cellular networks, some researchers proposed to offload part of the cellular traffic through other wireless networks. Some research efforts have been focusing on offloading 3G traffic through opportunistic mobile networks or D2D networks [24], [25]. Others utilized public WiFi for 3G traffic offloading [6], [7]. Besides offloading through WiFi or opportunistic mobile networks, existing work also leveraged mobile nodes with good signal. For example, UCAN [26] enabled 3G base station to forward data to mobile clients with better channel quality, which then relay data to destination clients with poor channel quality through peer-to-peer links. Different from previous work which only considered traffic offloading, our work focuses on computation offloading.

III. Background and Motivation

In this section, we first introduce cellular networks and their energy model, and then give the motivation of our peer-assisted computation offloading.

The Universal Mobile Telecommunications System (UMTS) is a popular 3G standard developed by 3GPP. While GSM is based on TDMA, UMTS uses Wideband CDMA (WCDMA) radio access technology and provides a transfer rate of up to 384 Kbps for its first version Release 99. After that, High Speed Downlink Packet Access (HSDPA) has been introduced and provides a higher data rate up to 14 Mbps. In Release 6, the uplink is enhanced via High Speed Uplink Packet Access (HSUPA) to support a peak data rate up to 7Mbps. Later, HSDPA and HSUPA have been merged into one, High Speed Packet Access (HSPA), and its evolution

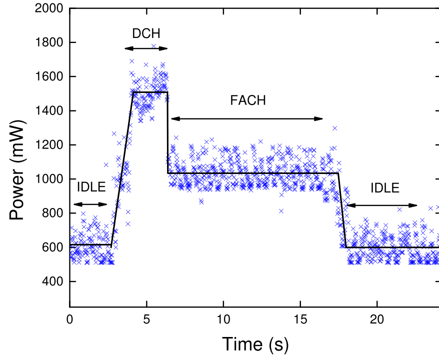


Fig. 1. The power level of the UMTS cellular interface at different states.

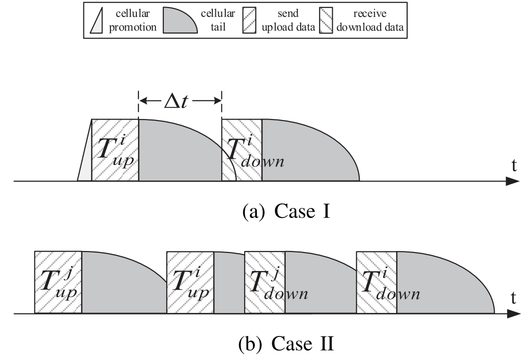


Fig. 2. Execution model for offloaded tasks.

HSPA+ has been introduced and standardized. HSPA+ offers a number of enhancements, supporting an increased data rate up to 84 Mbps [27]. The Long Term Evolution (LTE) is the latest extension of UMTS. LTE enhances both the radio access network and the core network. The targeted user throughput of LTE is 100Mbps for downlink and 50 Mbps for uplink, significantly higher than existing 3G networks [28].

A. Cellular Networks and Power Model

The power model of a typical data transmission in UMTS is shown in Fig. 1. Initially, the radio interface stays in the IDLE state, consuming very low power. When there is a data transmission request, it promotes to the DCH state. After the completion of data transmission, it stays in the DCH state for some time and then demotes to the FACH state before returning to the IDLE state. The extra time spent in the high-power DCH and FACH states is called the *tail time*. HSPA+ and LTE have similar power models. Thus, we generalize the power consumption of the cellular interface into three states: *promotion*, *data transmission* and *tail*. The power in the promotion and tail state are denoted as P_{pro}, and P_{tail}. We differentiate the power for uploads from downloads in data transmission, and denote them as P_{up} and P_{down}, respectively.

B. Task Execution Model

A task can be executed locally on the mobile device or executed on the server through offloading. If task T_i is executed on the mobile device, its energy consumption is denoted as E^i_{local}. If T_i is offloaded to the remote server with the help of a proxy, the energy consumption consists of two parts: the P2P part between the client and the proxy, and the cellular part between the proxy and the server. During offloading, the offloaded task may need some input data, denoted as s^i_{up}, which is sent from the client to the proxy and then uploaded from the proxy to the server. The offloaded task may also generate some output data, denoted as s^i_{down}, which is downloaded from the server to the proxy and then sent from the proxy to the client.

For the P2P part, the client and proxy use the WiFi direct interface to transmit the offloaded task. WiFi direct has much higher speed and larger transmission range than its Bluetooth counterpart. For WiFi direct, the promotion and tail energy are negligible. Thus, the energy consumption to offload task T_i over P2P link is calculated as

$$E^i_{p2p} = P_{p2p} \times (s^i_{up} + s^i_{down})/r_{p2p}, \tag{1}$$

where P_{p2p} is the data transmission power, and r_{p2p} is the transmission rate.

The cellular part consists of three steps: sending the upload data, executing the task on the server, and receiving the download data. Sending and receiving data of task T_i are denoted as two subtasks T^i_{up} and T^i_{down}, and the energy consumption of them are calculated as $E^i_{up} = P_{up} \times s^i_{up}/r_{up}$ and $E^i_{down} = P_{down} \times s^i_{down}/r_{down}$, where the upload and download data rate are denoted as r_{up} and r_{down}. There are two cases to calculate the energy of offloading task T_i over the cellular network. In the first case (Fig. 2(a)), after sending the upload data (T^i_{up}) to the server, the proxy is idle waiting for the download data (T^i_{down}) while T_i is executed on the server. Let Δt denote the interval between subtasks T^i_{up} and T^i_{down} (i.e., T_i's execution time on the server). Then, the energy consumption between these two subtasks (denoted as $E(T^i_{up}, T^i_{down})$) depends on Δt: 1) If Δt is larger than the tail time t_{tail}, the proxy will consume some extra promotion energy and tail energy. 2) If Δt is smaller than t_{tail}, there is a partial tail and no promotion energy. In summary,

$$E(T^i_{up}, T^i_{down}) = \begin{cases} P_{pro} \times t_{pro} + P_{tail} \times t_{tail}, & \text{if } \Delta t > t_{tail} \\ P_{tail} \times max\{\Delta t, 0\}, & \text{otherwise.} \end{cases} \tag{2}$$

In the second case (Fig. 2(b)), after sending the upload data (T^i_{up}) to the server, the proxy is busy offloading other tasks. Then there could be multiple subtasks between T^i_{up} and T^i_{down}. Let $\mathbb{S}_i$ denote the set of all offloaded tasks including T_i, and $\mathbb{S}'_i = \mathbb{S}_i \setminus \{T_i\}$. Each set can be considered as a sequence of subtasks ordered by their arrival times. We can use Eq. (2) to calculate the energy between adjacent subtasks and then get the overall energy consumptions of set $\mathbb{S}_i$ and set $\mathbb{S}'_i$ (denoted as $E(\mathbb{S}_i)$ and $E(\mathbb{S}'_i)$). Then the energy consumption of T_i in the cellular part is calculated as

$$E^i_{cell} = E(\mathbb{S}_i) - E(\mathbb{S}'_i). \tag{3}$$

TABLE I
MOBILE DEVICES AND NETWORK TYPES

Device	Provider	Network
Samsung Galaxy S3	Carrier 1	HSPA+
Samsung Galaxy S4	Carrier 2	LTE

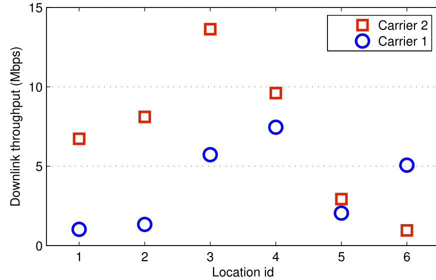


Fig. 3. Downlink throughput of different carriers at different locations.

C. Motivation for Peer-Assisted Computation Offloading

Existing study has shown that mobile devices within an area may have different service quality, especially when different service providers are used [9]. How will the service quality difference affect the efficiency of computation offloading? To answer this question, we have run some experiments.

Our testbed consists of two types of smartphones, served by two cellular carriers, as described in Table I. We picked 6 popular locations on our campus, and used these two phones to send data to our Linux server. We measured the data throughput of different carriers at each location, and the results are shown in Fig. 3. Then, we conduct computation offloading experiments at location 1, where Carrier 1 has extremely low data throughput but Carrier 2 has much better service quality.

We have implemented an Optical Character Recognition (OCR) application which automatically recognizes the characters in images and outputs the text on Android smartphones. The detailed setup and implementation of our testbed will be discussed in Section VII. We conduct experiments in three modes: no offloading and offloading with two different cellular networks. We run the application to recognize 10 images and repeat the test several times to measure the average energy consumption and the delay.

The results are shown in Fig. 4. As can be seen, offloading computation with poor service quality (Carrier1-offload) may consume more energy and increase the delay compared to executing the computation locally. On the other hand, computation offloading under good service quality (Carrier2-offload) can significantly reduce the energy consumption and delay. Based on these results, mobile devices with poor service quality should leverage the node with better service quality for computation offloading.

IV. PACO SYSTEM ARCHITECTURE

PACO considers the service quality difference among mobile devices, and leverages peers with better service quality for computation offloading. In this section, we present a high-level overview of the PACO architecture.

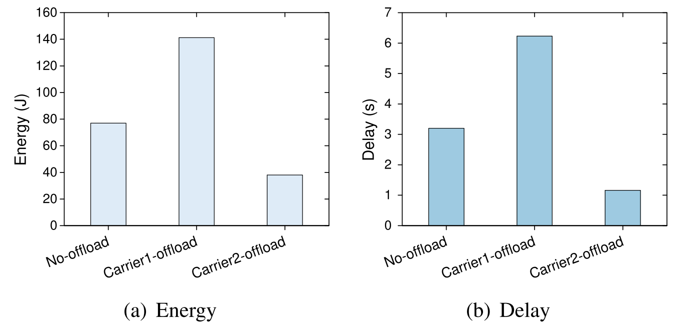


Fig. 4. Energy and delay with/without computation offloading.

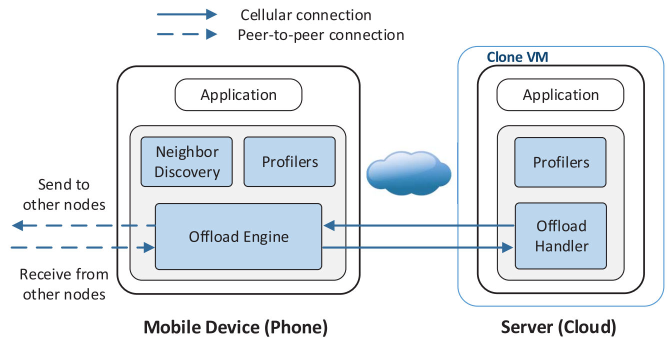


Fig. 5. Overview of the PACO architecture.

The architecture of PACO is shown in Fig. 5. PACO shares similar design with CloneCloud and ThinkAir by creating virtual machines (VMs) of a complete smartphone system in the cloud. In this way, PACO enables easy computation offloading between devices of diverging architectures, even different instruction set architectures (e.g., ARM-based smartphones and x86-based servers).

On the mobile device side, PACO consists of three components. (1) Profilers for device, application and network. The device profiler measures the mobile device's energy consumption characteristics and builds its energy model at the initialization time. The application profiler tracks a number of parameters related to program execution, such as the data size, the execution time and the resource requirements of individual tasks. The network profiler continuously monitors the network condition such as the data rate of the cellular network. (2) Neighbor discovery, which identifies neighboring nodes that support PACO service and collects a list of network quality profiles from them. (3) Offload engine, which determines whether to offload computation tasks and to which node (proxy) to offload. If a mobile device is chosen as a PACO proxy, its offload engine also handles the communication with the cloud server. It receives offloading requests from PACO clients, and sends the offloading requests to the server through the cellular interface. After the server finishes the execution of an offloaded task, the proxy receives the result and sends it back to the corresponding client.

The PACO server will execute the offloaded tasks. It consists of two parts: offload handler and profilers. The offload handler manages the connection with the proxy. After the initial

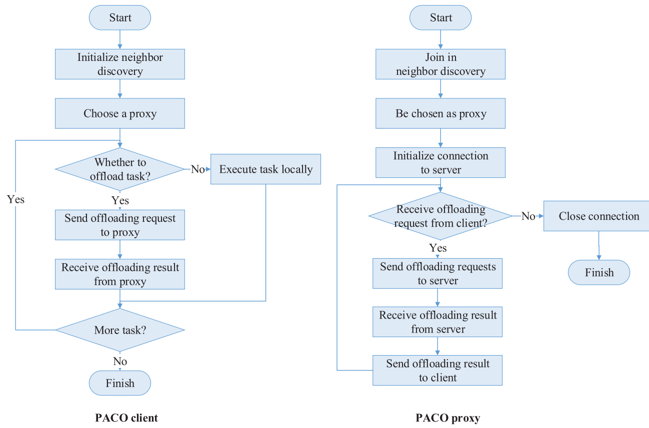


Fig. 6. The work flow of the PACO client and proxy.

connection setup, the server waits to receive the offloading requests, executes the offloaded code, and sends the results back to the proxy. Along with the results from executing the offloaded code, the server also sends the profiling data collected by its profilers, which can help future offloading decisions for the client.

To better understand how PACO works, Fig. 6 illustrates the work flow of the PACO client and proxy.

V. Neighbor Discovery and Proxy Selection

In PACO, the client first uses neighbor discovery to find a list of neighbors with better service quality, and then decides which neighbor is chosen to be the proxy. In this section, we first explain this proxy discovery process and the incentives of being a proxy. Then we discuss the overhead and other related issues in the process.

A. Proxy Discovery and Incentives

Each mobile device in PACO needs to detect nearby neighbors and identify which neighbors support PACO, i.e., are willing to offload tasks for others. We leverage the DNS (Name Domain System) based service discovery (DNS-SD) to find neighbors. It allows mobile devices to discover neighboring nodes supporting a specific service using the WiFi direct interface directly, without the support of central servers and access points. Android begins to support DNS-SD since Android 4.1 (Jelly Bean). On each node, we register "PACO" as a service, with "_http._tcp" as the service type. And a local port is assigned for this service. After registration, the node will be able to respond to the "PACO" requests from neighbors.

When a client joins PACO, it sends a "PACO" request using the WiFi direct interface. Each neighbor that supports PACO service will respond to the request and provide its IP address and port number. Based on these information, two nodes can connect with each other via the WiFi direct interface and exchange the network quality information.

Since the proxy will incur some cost in contributing its cellular bandwidth to assist other clients, there is a need to provide some incentive to offset this cost. The cost includes two parts. The first is the cost of transferring offloading requests through the cellular link for which the carrier charges a fee. This fee is determined by the user's service plan. For simplicity, we assume that this fee is known by the user and use DC_i to represent the data cost of node i. The second part is related to energy. We model this part as node i's residual energy (RE_i), which is the fraction of the remaining battery energy and it is within the range of [0,1]. Therefore, we model the cost of a node i as follows:

$$C_i = \frac{DC_i}{RE_i}.$$

As DC_i increases, using the cellular link of node i becomes more expensive. As RE_i decreases, the battery energy is more valuable for node i, and thus its cost will be higher. This cost model can promote fairness between nodes. In the beginning, a node with higher throughput and sufficient battery life is chosen to be the proxy. When serving clients, the proxy node contributes its own cellular bandwidth to transmit clients' offloading requests and its residual energy drops quickly, which will increase its cost to serve as a proxy. As a result, the node will less likely to be chosen as a proxy later. Based on this cost model, we can then apply some credit-based incentive mechanism like [9], [29]. Nodes are awarded credits for serving as proxies. These credits can be redeemed in the form of real money, or be used to pay its proxy when the node becomes a PACO client later.

In PACO, suppose a client, say k, initiates a neighbor discovery. Nodes receiving this request will estimate their cost of providing help and send such information and their network quality information to node k. After neighbor discovery, client k collects a list of network quality profiles and serving costs from neighbors. Then neighbors with higher data throughput than client k are candidates for being proxy. All proxy candidates are listed in descending order of the data throughput. Assume client k wants to offload its computation tasks and is willing to pay a cost of C. It will check the proxy candidate list, and select the first neighbor i which satisfies $C \geq C_i$ to be the proxy.

B. Discussions

1) Neighbor Discovery Cost: The neighbor discovery process consumes extra energy. To quantify this extra energy cost, we measured the power consumption during the neighbor discovery process. Fig. 7 shows the power consumption of discovering three neighbors. The power level of the initialization process is much higher since the mobile device needs to start its WiFi direct interface for neighbor discovery. After initialization, the power consumption becomes much lower. Frequently performing neighbor discovery consumes too much energy, but many neighbors may not be detected with low discovery frequency. To save energy, the neighbor discovery process in PACO will only be started when a node's proxy candidate list is empty.

2) Proxy Update: The service quality and the cost of mobile nodes may change with time due to mobility, channel interferences and congestion at the proxy. In PACO, a node periodically asks all nodes in its proxy candidate list for information such as the current service quality and the serving

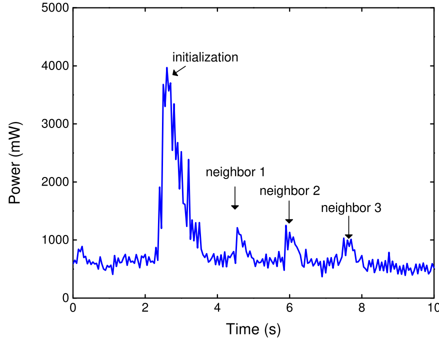


Fig. 7. The power consumption of the proxy discovery process.

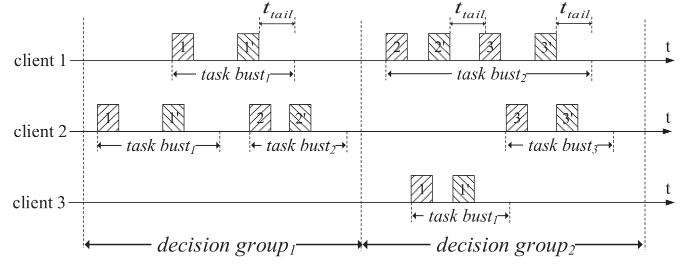


Fig. 8. Task burst and decision group. For simplicity, T^i_{up} and T^i_{down} (sending upload and receiving download data of T_i) are represented by box i and box i', respectively.

cost. A client will switch to a better proxy if the service quality of the current proxy becomes worse than itself.

3) Proxy Failures and Recovery: The connection between a client and its proxy will be broken when the proxy moves out of range. The client can detect such a failure if it receives no response from the periodical proxy updates, or if the delay of an offloaded task is too long. In such cases, the client will switch to another proxy in its proxy candidate list. If the proxy candidate list is empty, the client should start another round of neighbor discovery to look for a new proxy.

VI. Offloading Decision

For a PACO client, its offload engine decides whether a computation task should be offloaded. In this section, we present the offloading decision algorithms.

A. Problem Statement

Suppose a chosen proxy p is serving a group of clients (nodes) that run computationally intensive applications and generate N tasks in total. After a task T_i is generated, it is executed either locally on the client or on the remote server. The offloading decision problem is to find an offloading decision sequence $l_1, \ldots, l_i, \ldots, l_N$ which executes all tasks with the minimum amount of energy, where l_i indicates the offloading decision of task T_i.

When a task is offloaded via proxy p, the client can save some energy by moving the computation to the server, but the proxy will cost extra communication energy to offload the computation to the server. Moreover, the communication between proxy p and the server suffers from the long tail problem, which may impact the decisions of other tasks. Specifically, let us consider the following case. For a single task T_i at a client, suppose offloading it costs more by considering the communication cost including the tail energy. If the proxy p has just offloaded some other tasks, the tail energy for offloading T_i can be saved or reduced by those tasks. As a result, it is possible that offloading T_i costs less energy than executing it locally. Therefore, it is a challenge to calculate the actual offloading cost for a single task considering various decisions of other tasks.

Although Geng *et al.* [18] considers the effects of the long tail problem, their work focused on single mobile device which offloads tasks to the server sequentially. Due to the use of proxy, multiple clients offload their tasks to the proxy which accumulates multiple tasks, which makes existing solution unsuitable. In the following, we first propose an offline offloading algorithm, which can minimize the energy consumption. However, it requires the complete knowledge of all tasks including the task arrival times. Thus, we relax these assumption and present an online offloading algorithm.

B. Offline Offloading Algorithm

1) Independent Decision Group: For the offline offloading problem, as each task in N can be either executed locally or on the remote server, the solution space of this problem is $O(2^N)$. When N is large, it is hard to find the optimal solution. We find that the N tasks can be divided into several small groups with n tasks in each, where n is much smaller than N. The optimal decision of each group is independent with each other. Therefore, the original problem is divided into subproblems with a tractable solution space $O(2^n)$.

To identify such independent decision groups, we first introduce the concept of *task burst*. Suppose there is a set of tasks on a client. For each offloaded task T_i, we denote the arrival times of its two subtasks on proxy p as t^i_{up} and t^i_{down}. If two offloaded tasks from this client have an interval smaller than the tail time t_{tail}, the cellular interface of proxy p will stay in high power state during the whole period between these two tasks. Then the offloading cost of one task depends on the offloading decision of the other. Therefore, a task burst is defined as a group of tasks $(T_1, T_2, \ldots, T_j)$, where $t^i_{up} - t^{i-1}_{down} < t_{tail}, 1 < i \leq j$. For example, in Fig. 8, the first task of client 1 belongs to one task burst, while the second and the third tasks belong to another task burst.

For a group of clients that use the same proxy, as long as any of their task bursts overlap, the offloading decision of any task burst will affect some other task burst. Therefore, these task bursts are formed into one *decision group* as shown in Fig. 8. Offloading decisions in different decision groups are irrelevant since there is no overlap in calculating the tail energy. Thus, by combining the optimal decisions of each decision group, we can obtain the global optimal offloading decision.

2) Problem Formulation: Suppose there are n tasks in a decision group, and our problem is to find an optimal offloading decision sequence to minimize energy. This problem can be mapped to the shortest path problem as shown in Fig. 9. We introduce two dummy nodes: node V_{src} as the source

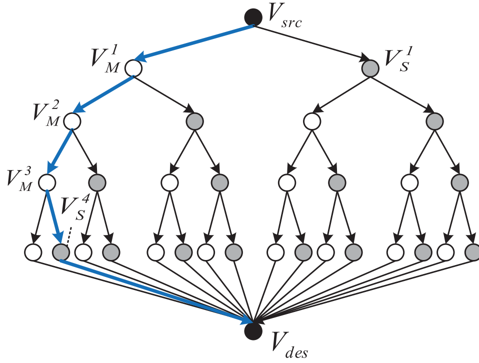


Fig. 9. Mapping the offloading decision problem to the shortest path problem. White nodes have $l_i = M$, while grey nodes have $l_i = S$.

node and node V_{des} as the destination node. First we create a directed graph as a full binary tree rooted from V_{src}. Formally, we denote a node at depth i as $V_{l_i}^i$ (or V_l^i for simplicity), where l_i indicates the offloading decision of T_i:

$$l_i = \begin{cases} M & T_i \text{ is executed on the mobile device;} \\ S & T_i \text{ is executed on the remote server.} \end{cases}$$

Specifically, each node at depth $h = i - 1$ has two children at depth $h = i$, where the left child V_M^i corresponds to task i being executed on local mobile device and the right child V_S^i corresponds to task i being offloaded to the remote server. We create a link from each node at depth n to the destination node V_{des}. In this graph, every path from V_{src} to V_{des} will map to one specific offloading decision sequence, and vice verse. Fig. 9 shows an example of such a graph for four tasks. The highlighted path corresponds to decision sequence M, M, M, S which means to execute tasks 1, 2, 3 on the mobile and to offload task 4 to the server.

The edge weight from a node V_l^{i-1} to its child V_l^i, denoted as W_{i-1}^i, is defined as the additional energy consumed by T_i. For edges that the endpoint has $l_i = M$, T_i is executed on the client. Then, the edge weight is the energy consumed to execute T_i on the mobile device, which is E_{local}^i. For edges that the endpoint has $l_i = S$, T_i is offloaded to the server. The computation of this weight is not straightforward, since the energy of an offloaded task is related with previous offloaded tasks, which is determined by the offloading decision history of all previous tasks. To solve this problem, we have each node V_l^i record the l_i values of all nodes along the path from V_{src} to V_l^i. Based on this information, we can get sets $\mathbb{S}_i$ and $\mathbb{S}'_i$ of task T_i and calculate the cellular energy cost of offloading T_i using Eq. (3). Since offloading T_i incurs additional P2P energy to transfer data between nodes, we also need to add this energy calculated by Eq. (1). In summary, we can compute the edge weight using Eq. (4) for all edges from $V_l^{i-1}(V_{src})$ to V_l^i $(1 \le i \le n)$. The edge weight from V_l^n to V_{des} is set to 0.

$$W_{i-1}^i = \begin{cases} E_{local}^i & \text{if } l_i = M; \\ E_{cell}^i + E_{p2p}^i & \text{if } l_i = S. \end{cases} \quad (4)$$

Under this framework, we transform the offloading decision problem to finding the shortest path in terms of energy consumption from V_{src} to V_{des} in the graph.

3) Offline Offloading Algorithm: We use the A^* search algorithm [30] to find the shortest path in the predefined directed graph corresponding to the optimal offloading decision.

In order to expand the fewest possible nodes in searching for an optimal path, a search algorithm must decide wisely which node to expand next. For example, Dijkstra's algorithm expands outwards from the source node and selects the node closest to the source. The greedy best-first search works in a similar way, except that it estimates the distance from any node to the destination. Instead of selecting the node closest to the source, it selects the node closest to the destination. It runs much quicker than Dijkstra's Algorithm because it uses the estimate to guide its way towards the goal very quickly. The A^* search algorithm combines the ideas of Dijkstra's algorithm and the greedy best-first search, and uses a specified rule to determine which one should be expanded next. For each node, A^* considers both its distance from the source node and its estimated distance to the destination.

Specifically, in the A^* search algorithm, each node V_l^i has a cost function $\hat{f}(V_l^i)$ to estimate the cost of a shortest path constrained to go through it, which contains two parts, as defined in Eq. (5).

$$\hat{f}(V_l^i) = \hat{g}(V_l^i) + \hat{h}(V_l^i), \quad (5)$$

where $\hat{g}(V_l^i)$ is the distance from the source node to the current node, i.e., the sum of the edge weight on the path P from V_{src} to V_l^i, which is

$$\hat{g}(V_l^i) = \sum_{(V_l^{j-1}, V_l^j) \in P} W_{j-1}^j, \quad (6)$$

and $\hat{h}(V_l^i)$ is an estimate of the cost from V_l^i to the destination. For A^* algorithm to find the optimal solution, $\hat{h}(V_l^i)$ should not be greater than the actual minimum cost from V_l^i to the destination. To satisfy this property, we define $\hat{h}(V_l^i)$ to be the minimal energy to execute all the remaining tasks, as shown in Eq. (7). When the remaining task T_j is executed on the local mobile, the energy consumption is E_{local}^j; when it is offloaded, we ignore the tail energy and only account for the energy to transmit the data through the cellular network and the P2P link. Therefore, $\hat{h}(V_l^i)$ will always be smaller than the actual minimum cost.

$$\hat{h}(V_l^i) = \sum_{j=i+1}^{n} min\{E_{local}^j, E_{up}^j + E_{down}^j + E_{p2p}^j\} \quad (7)$$

The key idea of A^* is to explore a graph by expanding the most promising node chosen according to node's $\hat{f}$ value. The detailed steps of the algorithm is shown in Algorithm 1. Specifically, it maintains an open set that keeps nodes that are about to be searched in the next round. The open set is initialized as V_{src}. In each round of the algorithm, A^* picks a node with minimum $\hat{f}$ value from the open set and expands its neighbor nodes into the open set. Then this node is removed from the open set. Finally, when V_{des} is picked from the open set, A^* can back track to V_{src} and find out the optimal solution.

Algorithm 1 Finding the Optimal Offloading Decision Using A^* Algorithm

```
   Data: Task graph
 1 Initialization:
 2 Decision sequence S ← ∅ ;
 3 OpenSet ← {V_src} ;
 4 f̂(V_src) ← ĥ(V_src) ;
 5 Procedure FindOptimalDecision(V_src, V_des)
 6   while OpenSet ≠ ∅ do
 7     V_l^i ← the node in OpenSet with min f̂(V_l^i) value ;
 8     if V_l^i == V_des then
 9       return ConstructOptimalDecision (V_l^i) ;
10     end
11     OpenSet ← OpenSet \ {V_l^i} ;
12     for each child V_l^{i+1} of V_l^i do
13       ĝ(V_l^{i+1}) ← ĝ(V_l^i) + W_i^{i+1} ;
14       f̂(V_l^{i+1}) ← Eq. (5) ;
15       V_l^{i+1}.parent ← V_l^i ;
16       OpenSet ← OpenSet ∪ {V_l^{i+1}} ;
17     end
18   end
19 Procedure ConstructOptimalDecision(V_l^i)
20   while V_l^i ≠ V_src do
21     if V_l^i == V_des then
22       V_l^i ← V_l^i.parent ;
23       continue ;
24     end
25     S[i] ← l_i ;
26     V_l^i ← V_l^i.parent ;
27   end
28   return S ;
```

4) Algorithm Analysis: In this section, we discuss the optimality of the offline algorithm to find the optimal offloading decision and its complexity.

We first introduce the notations used in the A^* algorithm. Let $f(V)$ be the actual cost of an optimal path constrained to go through node V, from V_{src} to V_{des}. Then $f(V)$ can be written as the sum of two parts:

$$f(V) = g(V) + h(V),$$

where $g(V)$ is the actual cost of an optimal path from V_{src} to V, and $h(V)$ is the actual cost of an optimal path from V to V_{des}. Note that $f(V_{src}) = h(V_{src})$ is the cost of an unconstrained optimal path from V_{src} to V_{des}. In fact, $f(V) = f(V_{src})$ for every node V on an optimal path, and $f(V) > f(V_{src})$ for every node V not on an optimal path.

Let $\hat{g}(V)$ be an estimate of $g(V)$, $\hat{h}(V)$ be an estimate of $h(V)$. Then we could add them to form an estimate of f:

$$\hat{f}(V) = \hat{g}(V) + \hat{h}(V).$$

In the offline offloading algorithm, we have $\hat{g}(V) = g(V)$ according to the definition of $\hat{g}$ in Eq. (6), and $\hat{h}(V) \leq h(V)$ according to the definition of $\hat{h}$ in Eq. (7).

To formally prove the optimality of the offline algorithm which uses A^* to search for the shortest path, we introduce the following lemma and theorem based on [30]:

Lemma 1: Suppose $\hat{h}(V) \leq h(V)$ for all V, and suppose A^* has not terminated. If a node is currently in the open set of A^*, it is an open node. Then, for any optimal path P from V_{src} to V_{des}, there exists an open node V' on P with $\hat{f}(V') \leq f(V_{src})$.

Proof: Since A^* has not terminated, there must be at least one open node on the optimal path P according to the algorithm. Hence, there exists an open node V' on P with $\hat{g}(V') = g(V')$ by definition of $\hat{g}$. Then by definition of $\hat{f}$, we have:

$$\begin{aligned}\hat{f}(V') &= \hat{g}(V') + \hat{h}(V') \\ &= g(V') + \hat{h}(V') \\ &\leq g(V') + h(V') = f(V').\end{aligned}$$

P is an optimal path, so $f(V') = f(V_{src})$ for all $V' \in P$, which completes the proof. ∎

Theorem 1: If $\hat{h}(V) \leq h(V)$ for all V, the A^* algorithm can find the minimum cost path from V_{src} to V_{des}.

Proof: According to the Algorithm 1, the A^* algorithm must terminate at V_{des}. We prove this theorem by assuming the contrary; that is, the offline algorithm terminates at V_{des} without achieving minimum cost. Then at V_{des}, $\hat{f}(V_{des}) = \hat{g}(V_{des}) > f(V_{src})$. But by Lemma 1, there exists just before termination an open node V' on an optimal path with $\hat{f}(V') \leq f(V_{src}) < \hat{f}(V_{des})$. Then V' would have been selected for expansion rather than V_{des}, contradicting the assumption that the algorithm terminated. ∎

The time complexity of the A^* algorithm depends on specific situation of the graph. For a node V, if $\hat{g}(V) + \hat{h}(V)$ is bigger than the cost of the shortest path, then V and all its children will not be searched by the A^* algorithm, and thus we can quickly find the shortest path. In the worst case, the offline offloading algorithm using A^* has to expand all nodes to find the optimal path. Then the complexity of the algorithm is equal to that of the Dijkstra shortest path algorithm. In practical, the offline offloading algorithm achieves better performance by using heuristics to prune away many nodes.

5) Delay Constraint: Until this stage, the only objective of the offline offloading algorithm is to minimize the energy consumption. However, the task completion time should also be considered for some applications with delay constraints. In this section, we solve the offloading decision problem considering both energy consumption and delay constraints.

Under the same framework used in Section VI-B.2, the new offloading decision problem is to find the shortest path in terms of energy consumption from V_{src} to V_{des} in the graph (Fig. 9), subject to the constraint that the total completion time of that path must be less than or equal to the delay constraint, $D_{deadline}$. In a formal description, we are looking for

$$\begin{aligned}&\min_{P \in \mathcal{P}} \ e(P) \\ &\text{s.t. } d(P) \leq D_{deadline},\end{aligned} \tag{8}$$

where $\mathcal{P}$ is the set of paths from V_{src} to V_{des}, and $e(P)$ and $d(P)$ are the total energy and delay of the path P, respectively.

In Section VI-B.2, the edge weight is defined as the energy consumption of executing a task, and $e(P)$ can be calculated by summing up the weight of all edges on the path P. Similarly, when the edge weight is defined as the completion time of a task, $d(P)$ can also be easily calculated. We omit the simple mathematical calculation of the edge weight using task completion time. $D_{deadline}$ is normally set by the user based on different applications. By default it is set to be the task completion time on the local mobile device.

This delay constrained least cost path problem has been proven to be NP-hard [31]. It can be approximately solved by the LARAC algorithm [32] using Lagrange relaxation. Specifically, the Lagrangian function is defined as follows:

$$L(\lambda) = \min_{P \in \mathcal{P}} e_\lambda(P) - \lambda D_{deadline},$$

where λ is the Lagrangian multiplier, and $e_\lambda(P) = e(P) + \lambda d(P)$.

Lemma 2: $L(\lambda)$ is a lower bound to the problem defined in Eq. (8) for any $\lambda \geq 0$.

Proof: Let P^* denote an optimal solution of Eq. (8). Then

$$\begin{aligned} L(\lambda) &= \min_{P \in \mathcal{P}} e_\lambda(P) - \lambda D_{deadline} \\ &\leq e_\lambda(P^*) - \lambda D_{deadline} \\ &= e(P^*) + \lambda\left(d(P^*) - D_{deadline}\right) \\ &\leq e(P^*) \end{aligned}$$

proves the lemma. ■

To obtain the best lower bound we need to maximize the function $L(\lambda)$ and find the maximizing λ^*. We apply the LARAC algorithm described in Algorithm 2 to find the optimal λ and the corresponding e_λ-minimal path for a given source and destination pair. In the algorithm, **ShortestPath**(V_{src}, V_{des}, c) returns a shortest path from V_{src} to V_{des} in terms of cost c.

The algorithm first finds the minimal-energy path. If it satisfies the delay constraint $D_{deadline}$, it will be the optimal path and the algorithm terminates. Otherwise, the algorithm stores the path as the best path that does not satisfy $D_{deadline}$ (denoted by P_e). Then it finds the shortest path on delay d. If the path satisfies the delay constraint (i.e., a feasible solution), the algorithm stores this path as the current best appropriate path (denoted by P_d). Otherwise there is no solution, so the algorithm terminates. After that, the algorithm repeatedly updates P_e and P_d with other paths to obtain the optimal λ. Although this algorithm cannot guarantee to find the optimal path, it is shown to have good performance and polynomial running time [32].

Algorithm 2 Finding e_λ-Minimal Path for Delay Constrained Least Energy Path Problem

```
Data: V_src, V_des, D_deadline
Result: e_λ-minimal path : P*_λ
1  P_e ← ShortestPath (V_src, V_des, e) ;
2  if d(P_e) ≤ D_deadline then
3  |  return P_e ;
4  end
5  P_d ← ShortestPath (V_src, V_des, d) ;
6  if d(P_d) > D_deadline then
7  |  return There is no solution ;
8  end
9  while true do
10 |  λ ← (e(P_e) − e(P_d)) / (d(P_d) − d(P_e)) ;
11 |  P_λ ← ShortestPath (V_src, V_des, e_λ) ;
12 |  if e_λ(P_λ) = e_λ(P_e) then
13 |  |  return P_d;
14 |  else if d(P_λ) ≤ D_deadline then
15 |  |  P_d ← P_λ ;
16 |  else
17 |  |  P_e ← P_λ ;
18 |  end
19 end
```

C. Online Offloading Algorithm

The offline offloading algorithm can minimize the energy consumption. However, it is a centralized algorithm which requires the complete knowledge of all tasks from all clients, and thus it can only be used as a performance bound. In practice, we need an online offloading algorithm, where each client makes offloading decisions by itself. Furthermore, it is impractical to accurately predict future task arrival time of different users with different applications. Therefore, the online offloading algorithm only uses information of the current tasks.

The basic idea of our online algorithm is to offload a task when the offloading can save energy while satisfying the delay constraint. We first describe how to estimate the energy and delay of task offloading. Then, we present the offloading decision and congestion avoidance scheme. Last, we discuss the overhead of the online offloading algorithm.

1) Delay Estimation: For task T_i, if it is executed locally, the execution time is denoted as D^i_{local}. When offloaded to the remote server via the proxy node p, the process is shown in Fig. 10. Since most output data are small and the cellular download bandwidth is high, we ignore the queuing delay on the download link similar to existing work [16], [33]. Therefore, the delay of offloading task T_i consists of four parts: the time to transmit data via the P2P interface, the queuing delay on cellular upload link, the time to transmit data via the cellular interface, and the execution time on the server. The P2P delay and the cellular network delay depend on the upload data size s^i_{up}, and the download data size s^i_{down}. The queuing delay can be estimated by the queue size at the proxy p (denoted as S^p_{queue}). The task execution time on the server (denoted as d^i_{server}) can be obtained by task profiling. Putting them together, the delay to offload T_i via the proxy node p is:

$$D^{i,p}_{remote} = \frac{s^i_{up} + s^i_{down}}{r_{p2p}} + \frac{S^p_{queue} + s^i_{up}}{r_{up}} + \frac{s^i_{down}}{r_{down}} + d^i_{server}. \tag{9}$$

2) Energy Consumption: For task T_i, if it is executed locally, the energy consumption is denoted as E^i_{local}. When

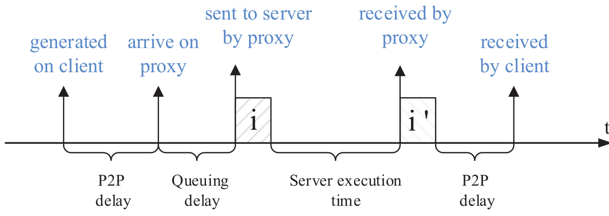


Fig. 10. The process of offloading task T_i. For simplicity, T^i_{up} and T^i_{down} (sending upload and receiving download data of T_i) are represented by box i and box i', respectively.

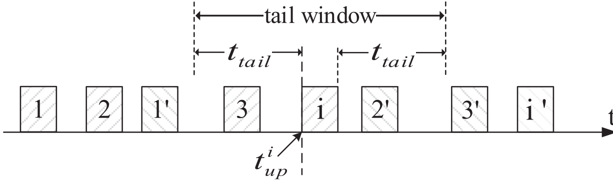


Fig. 11. Offloading subtasks at proxy p and T^i_{up}'s tail window. For simplicity, T^i_{up} and T^i_{down} (sending upload and receiving download data of T_i) are represented by box i and box i', respectively.

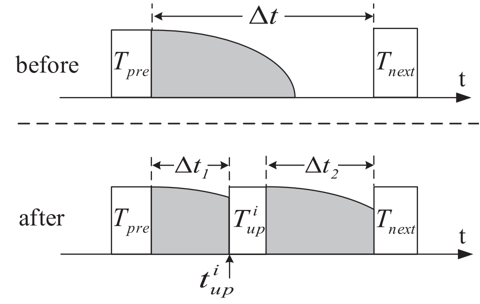


Fig. 12. Cellular energy consumption at proxy p before and after T^i_{up} is offloaded.

this task is offloaded to the server via the proxy node p, the energy consumption includes the energy consumed by the P2P interface, and the data transmission energy consumed at node p. The P2P energy (E^i_{p2p}) can be computed using Eq. 1. We show how to estimate the data transmission energy as follows.

First, we show the tasks at proxy node p assuming T_i has been offloaded via p in Fig. 11. Assume task T_i is generated at time t_i. We can estimate the time when p can start to send the upload data T^i_{up} (denoted as t^i_{up}) by adding the P2P delay and queuing delay to t_i. Before time t^i_{up}, proxy p has the information of all the past offloaded tasks from all clients it serves. After time t^i_{up}, proxy p is expecting the download data of some already offloaded tasks.

The data transmission energy of T_i consists of the energy of its subtasks T^i_{up} and T^i_{down}. Since the energy calculations of the two subtasks are the same, we use T^i_{up} as an example. We introduce the concept of tail window for estimating the cellular energy of offloading T^i_{up} as follows. After T^i_{up} is transmitted on the cellular uplink, the cellular interface enters the long tail state. The tail energy related to T^i_{up} can be affected in two cases (shown in Fig. 11). In the first case, when there is a subtask within time period $(t^i_{up} - t_{tail}, t^i_{up})$, offloading T^i_{up} can cut the long tail generated by that subtask. In the second case, when there is a subtask within time period $(t^i_{up}, t^i_{up} + s^i_{up}/r_{up} + t_{tail})$, the long tail generated by sending T^i_{up} can also be cut. Therefore, to calculate the actual tail energy to send T^i_{up}, we only need to consider the subtasks in the period $(t^i_{up} - t_{tail}, t^i_{up} + s^i_{up}/r_{up} + t_{tail})$, which is defined to be the *tail window* of T^i_{up}.

We denote the previous and the next tasks of T^i_{up} in its tail window as T_{pre} and T_{next}, respectively. Before T^i_{up} is offloaded (above in Fig. 12), the cellular energy consumed by proxy p between T_{pre} and T_{next} (denoted as $E(T_{pre}, T_{next})$) can be calculated based on interval Δt using Eq. (2). After T^i_{up} is offloaded, we use $E(T_{pre}, T^i_{up})$ to denote the tail energy between subtasks T_{pre} and T^i_{up}; $E(T^i_{up}, T_{next})$ denotes the tail energy between subtasks T^i_{up} and T_{next}. Each of them can be calculated based on interval Δt_1 and interval Δt_2 using Eq. (2). Then, we can get the cellular energy consumption of proxy p to offload T^i_{up} as the energy difference between before and after T^i_{up} is offloaded, which is

$$E^{i,up}_{cell} = P_{up} \times s^i_{up}/r_{up} + E(T_{pre}, T^i_{up}) + E(T^i_{up}, T_{next}) - E(T_{pre}, T_{next}). \quad (10)$$

Similarly, we can get the cellular energy consumption of proxy p to offload T^i_{down} as $E^{i,down}_{cell}$.

In summary, the energy consumption to offload T_i via the proxy node p is:

$$E^{i,p}_{remote} = E^i_{p2p} + E^{i,up}_{cell} + E^{i,down}_{cell}. \quad (11)$$

3) Offloading Decision and Congestion Avoidance: In a PACO group, proxy p serves several clients and handles their offloaded tasks. For task T_i generated by client k, to decide whether it should be offloaded, client k sends a query to proxy p for the task information within T_i's tail windows and the current queuing delay. Based on the query results, the client computes the energy and delay when offloading T_i via proxy p. The client will offload T_i when offloading can save energy (i.e., $E^{i,p}_{remote} < E^i_{local}$), and satisfy the delay constraint (i.e., $D^{i,p}_{remote} \leq D^i_{deadline}$) at the same time. The delay constraint $D^i_{deadline}$ is normally set by the user based on different applications. By default it is set to be the completion time of task T_i on the local mobile device (D^i_{local}). A complete description of the online algorithm is shown in Algorithm 3.

For a PACO proxy, serving too many offloaded tasks will increase the length of queue and thus increase the queuing delay for later tasks. Assume a client k makes offloading decision for task T_i given a chosen proxy p. If the delay to offload T_i via p is too long, e.g., longer than executing locally, client k will stop using proxy p and switch to a better proxy in its proxy candidate list. If no better proxy is found, the client will start another round of neighbor discovery to look for a new proxy as described in Section V-B.

4) Overhead Analysis: The signaling overhead of the online offloading algorithm is very small and can be neglected. Each client only needs to send queries about its own tasks and then make decisions locally. Since the query is very small and the

Algorithm 3 The Online Offloading Algorithm on Client k

Data: Task set on client k, proxy p
1 **Initialization:**
2 Decision sequence $\mathcal{S} \leftarrow \emptyset$;
3 **Procedure** `MakeDecision`(T_i)
4 $D^{i,p}_{remote} \leftarrow$ Eq. (9) ;
5 $E^{i}_{p2p} \leftarrow$ Eq. (1) ;
6 $T_{pre}, T_{next} \leftarrow$ get the tail windows of T^i_{up} and T^i_{down} ;
7 $E^{i,up}_{cell}, E^{i,down}_{cell} \leftarrow$ Eq. (10) ;
8 $E^{i,p}_{remote} \leftarrow$ Eq. (11) ;
9 **if** $E^{i,p}_{remote} < E^{i}_{local}$ **and** $D^{i,p}_{remote} \leq D^{i}_{deadline}$ **then**
10 offload T_i to proxy p ;
11 $\mathcal{S}[i] \leftarrow S$;
12 **else**
13 execute T_i on client k ;
14 $\mathcal{S}[i] \leftarrow M$;
15 **end**

transmission rate of the P2P link is high, the transmission delay of the query is negligible. The responses are also small, which only contains the start time and the execution time of several tasks. For a proxy, although it needs to maintain information of offloaded tasks and answer client queries, its processing overhead is small since the queries and the responses are very small.

The computation overhead of the online offloading algorithm is also very small. When a client generates a task, it sends query and receives response about the current task information at the proxy. Based on the response, the client can estimate the delay and energy to offload the task using Eq. (9) and Eq. (11). Then the client can quickly decide whether to offload the task considering both energy and delay constraints. This decision process only needs some simple mathematical computations.

VII. Performance Evaluations

In this section, we first use experiments to evaluate the benefits of peer-assisted computation offloading, and then use simulations to evaluate the performance of our system under various scenarios.

A. Experimental Setup

We have implemented PACO on four smartphones (3 GS3 and 1 GS4 As listed in Table I). we have measured the power level of the wireless interface at different states using a Monsoon Power Monitor [34], as shown in Table II. For the server, we create a clone VM by running the Android x86 virtual machine [35] on Oracle's VirtualBox. The clone executes on a dual-core desktop with a 2.3GHZ CPU and 6GB RAM running Linux.

All phones have pre-installed three computationally intensive applications: a face detection application, an optical character recognition (OCR) application and a speech-to-text application. The face detection application identifies all the faces in a picture and returns simple metrics for each detected face, such as the mid-point between the eyes, the distance in between, and the pose of detected faces. We have implemented the OCR application based on the Tesseract library [36]. The OCR application provides automatic conversion of photographed images of printed text into machine-readable text. The speech-to-text application takes an audio file and translates the speech into text using the Sphinx toolkit [37].

TABLE II
State Machine Parameters of Different Networks

Network	State	Power(mW)	Duration(s)
HSPA+	Promotion	1405.3±32.4	2.3±0.5
	Data	1984.8 ± 41.8	-
	Tail	1556.8±36.4	11.2±0.9
LTE	Promotion	1209.6±22.5	0.25±0.3
	Data	1853.4 ± 27.1	-
	Tail	1367.4±20.8	11.5±0.5
WiFi direct	Data	1316.5 ± 12.3	-

To evaluate the performance of PACO, we run each application on a different GS3 phone. Then we turn on PACO on all the phones and put them within WiFi direct communication range. Each client discovers three neighbors that support "PACO" service. As LTE has larger throughput than HSPA+, the GS4 phone is selected as the proxy among all candidates. We compared PACO with the following approaches:

- **All-Mobile**: all tasks are executed on the local mobile devices.
- **Self-Offload**: all tasks are offloaded to the remote server using the mobile devices' own cellular networks.
- **Proxy-All-Offload**: all tasks are offloaded to the remote server via the proxy's cellular network.
- **Proxy-ThinkAir-Offload**: the offloading decision is made using the approach in ThinkAir [3]. It compares the energy consumption of running the computation locally and that of offloading to the cloud, and only offloads the computation when energy can be saved via the proxy's cellular network. However, ThinkAir does not consider the long tail problem.

The metrics used for comparisons are energy consumption and delay. If a task is offloaded, the delay consists of communication delay (WiFi direct or Cellular delay) and computation delay. We use a Monsoon power monitor to measure the energy consumption. For "All-Mobile" and "Self-Offload", we only consider the energy consumption of GS3 phones (clients). For "Proxy-Offload" and "PACO", we also consider the energy consumed by the proxy node.

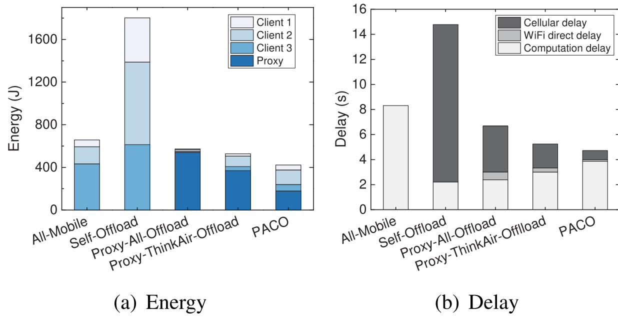
B. Experimental Results

In this section, we run some experiments to show the efficiency of PACO. The PACO prototype has one proxy node that serves three clients with different applications. In total, the clients generate 40 tasks, and the parameters of the tasks are shown in Table III.

We compare the performance of PACO with others and Fig. 13 shows the experimental results. As can be seen,

TABLE III
DATA SIZES AND EXECUTION TIMES OF DIFFERENT APPLICATIONS

Application	Avg data size (KB)	Avg execution time (s)	
		On phone	On server
Face Detection	923	2.9	0.2
OCR	2942	3.5	1.5
Speech-To-Text	692	25.0	7.3



(a) Energy (b) Delay

Fig. 13. Energy and delay of different offloading approaches in real experiments.

our offloading scheme (“PACO”) consumes less energy than other approaches. “Self-Offload” consumes more energy than the “All-Mobile” approach, which suggests that computation offloading is not helpful or even degrades performance when the client has poor service quality, “Proxy-All-Offload” only consumes 10% less energy than “All-Mobile”, which indicates that the energy saved by offloading the computation to the remote server is comparable to the extra energy wasted by transmitting more data to the remote server. Although the total energy consumption is less than “All-Mobile”, the energy consumption of the proxy is much higher compared to that of other clients. “PACO” consumes 36% less energy than “All-Mobile”. Compared to “Proxy-All-Offload”, “PACO” makes better offloading decisions to save more energy and utilize the proxy more properly. For “Proxy-ThinkAir-Offload”, since it does not consider the long tail problem, it may offload a task even though the actual energy consumption including the tail energy is more than local execution, and hence consumes more energy than “PACO”.

Fig. 13(b) compares the average delay of a task. For “All-Mobile”, the communication delay is 0 since its computation is not offloaded. For “Self-Offload”, although it manages to offload the task for faster execution on the server (i.e., less computation delay), it takes much longer time to transmit the offloading requests over the poor cellular connections. For “Proxy-ThinkAir-Offload”, since it does not consider the long tail problem, it may offload more tasks than “PACO” by underestimating the offload cost. Therefore, the communication delay consisted of cellular delay and WiFi direct delay is much longer than “PACO”. However, “Proxy-ThinkAir-Offload” has less computation delay by offloading more tasks. Overall, “PACO” has a smaller delay than “Proxy-ThinkAir-Offload”. Besides saving energy, “PACO” also reduces the delay. Generally, “PACO” decides to offload a task when energy can be saved. Since energy is closely related to

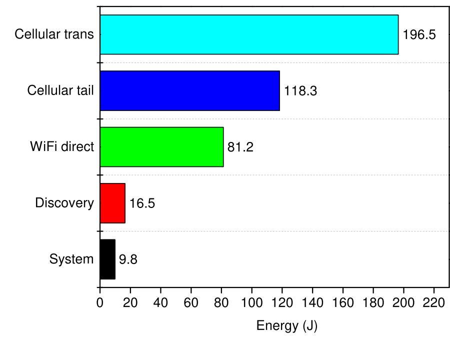
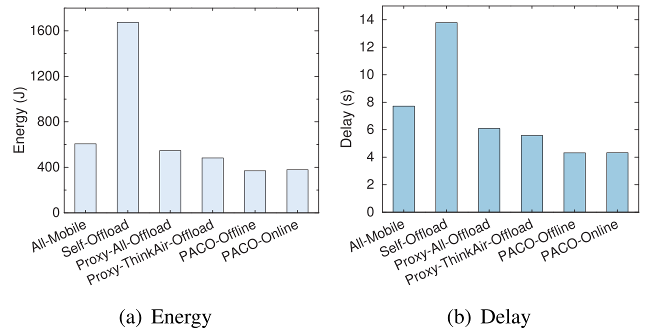


Fig. 14. Energy consumption of different components in PACO.



(a) Energy (b) Delay

Fig. 15. Energy and delay of different offloading approaches in simulation.

the communication delay and the computation delay, saving energy normally results in reduced delay. Also, it normally takes a client longer than the tail time to generate a new task, so there is always a promotion delay for the cellular interface to offload a new task. “PACO” aggregates the offloaded tasks from multiply clients to the proxy, and then such promotion delay can be avoided. On average, “PACO” can reduce the delay by about 43% compared to “All-Mobile”.

Fig. 14 shows the energy consumption of different components in PACO, which includes the cellular interface, WiFi direct interface, neighbor discovery and other system cost such as network profiling and offloading decision making. In general, data transmission consumes most of the energy, among which the cellular interface (cellular trans + cellular tail) consumes much more energy than the WiFi direct interface. The tail energy is very high because there are still idle time intervals between data transmissions on the proxy, thus introducing some tail energy. Other components only consume limited amount of energy. For example, neighbor discovery consumes less than 5% of the total energy, and system consumes less than 3% of the total energy.

C. Simulation Results

We have collected the task traces from the real experiments, and then use them to evaluate the performance of the offloading algorithms. Such simulations will give us more flexibility to study the effectiveness and scalability of our algorithms. In the simulations, we compare All-Mobile, Self-Offload, Proxy-All-Offload, Proxy-ThinkAir-Offload and two variants (Offline and Online) of our PACO offloading scheme.

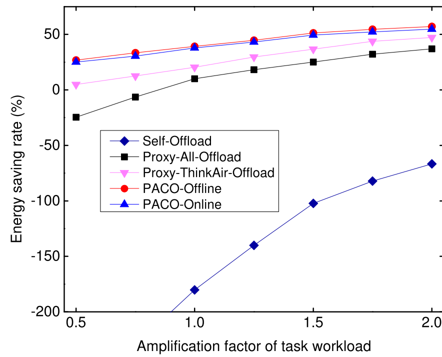


Fig. 16. Energy saving rate as a function of computation loads.

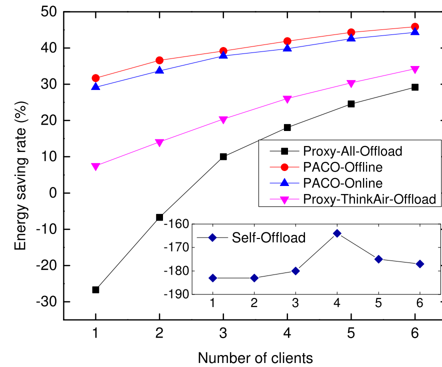


Fig. 17. Variation of the energy saving rate of different approaches with varying group size.

1) Performance Comparison: The energy consumption and delay of different approaches are shown in Fig. 15. Overall, our PACO offloading scheme can save 39% of energy and reduce the delay by 45% compared to that without computation offloading. By comparing Fig. 15 and Fig. 13, we can see that the simulation results are consistent with the experimental results. The energy consumptions of all approaches are a little higher than the simulation results since mobile devices consume some extra energy in idle state. Fig. 15 also shows that the online offloading algorithm can achieve similar energy and delay saving rate as the offline solution, although not as good as the offline algorithm.

2) Impact of Computation Load: The computation workload affects the computation time at the mobile device and the server, and hence it affects the overall delay and energy consumption of offloading algorithms. For a task with larger computation load, the advantage of offloading to a fast server over executing locally will be more obvious. In this section, we change the computation load by multiplying an amplification factor (i.e.[0.5,2.0]) to the original value, and evaluate its effects on performance.

Fig. 16 shows the energy saving rates of different approaches compared to "All-Mobile". When the computation load increases, offloading the task will reduce more computation time, resulting in higher energy saving rates. However, the advantage of our algorithms over "Proxy-ThinkAir-Offload" drops as the task workload increases. The reason is that the energy saved for computation gradually dominates the overall energy saving as the task workload increases, and then our algorithm's efforts on energy saving of data transmission including the tail energy become less obvious. When the task workload decreases (i.e., the amplification factor decreases from 1.0 to 0.5), "Proxy-All-Offload" even consumes more energy than "AllMobile". This is due to the fact that tasks at this time are too simple to be offloaded, and the energy saved in computation is not as much as the energy spent on data transmission. For "Self-Offload", its energy saving rate is always negative, which means it consumes more energy than "All-Mobile" in all cases. The energy saving rate decreases and becomes lower than -200 when the amplification factor decreases from 1.0 to 0.5, where "-200%" means that it consumes 200% more energy than "All-Mobile". This validates that a mobile device with poor service quality cannot benefit from computation offloading in most cases.

3) Impact of Group Size: We define the *group size* as the number of clients served by a proxy. Intuitively, as the size of the group increases, more offloaded tasks can be aggregated at the proxy and more likely to save the tail energy. In this section, we change the group size and evaluate its impact on the performance.

Fig. 17 shows the energy saving rates of different approaches compared to "All-Mobile". For "Self-Offload", each client in the group uses its own cellular connection to offload tasks to the remote server. Thus, the group size has no direct effect on its performance. The result shows that "Self-Offload" always consumes much more energy than the one without offloading, where "-180%" means that it consumes 180% more energy than "All-Mobile". For "Proxy-ThinkAir-Offload", it offloads more tasks than our PACO offloading scheme since it ignores the long tail problem and underestimates the offloading cost. Thus, the energy saving rate of PACO is always higher than "Proxy-ThinkAir-Offload". As expected, the energy saving rates of other approaches increase as the size of the group is increased. However, there is an upper-bound of this performance improvement. Consider an extreme scenario where all offloaded tasks are sent together as one bundle, then adding more clients (their tasks) has no advantage in further amortizing the tail energy. For "Proxy-All-Offload", the negative energy saving rate when the group size is small means that it consumes more energy than the one without offloading. On the other hand, our PACO offloading scheme can achieve better performance even when the group size is small. With more clients joining the group, the performance of PACO becomes even better.

VIII. Conclusions

In this paper we proposed PACO, a system that enables computation offloading in wireless networks, which is especially helpful for mobile devices suffering from poor service quality. PACO can identify neighbors with better service quality and choose a proper proxy. With PACO, clients send offloading requests to the proxy via WiFi direct interfaces, and the proxy offloads the computation to the remote server. This paper addressed research challenges such as how to discover the proxy for a client, and how to make offloading decisions to minimize the energy consumption. To validate our design,

we have implemented PACO on Android. Experimental and simulation results show that PACO can significantly reduce the energy and delay for mobile devices when running computationally intensive applications.

REFERENCES

[1] K. Kumar, J. Liu, Y.-H. Lu, and B. Bhargava, "A survey of computation offloading for mobile systems," *Mobile Netw. Appl.*, vol. 18, no. 1, pp. 129–140, 2013.

[2] E. Cuervo *et al.*, "MAUI: Making smartphones last longer with code offload," in *Proc. ACM MobiSys*, Jun. 2010, pp. 49–62.

[3] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, "Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading," in *Proc. IEEE INFOCOM*, Mar. 2012, pp. 945–953.

[4] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: Elastic execution between mobile device and cloud," in *Proc. ACM EuroSys*, Apr. 2011, pp. 301–314.

[5] A. Chakraborty, V. Navda, V. N. Padmanabhan, and R. Ramjee, "Coordinating cellular background transfers using loadsense," in *Proc. ACM MobiCom*, Oct. 2013, pp. 63–74.

[6] S. Dimatteo, P. Hui, B. Han, and V. O. K. Li, "Cellular traffic offloading through wifi networks," in *Proc. IEEE MASS*, Oct. 2011, pp. 192–201.

[7] K. Lee, J. Lee, Y. Yi, I. Rhee, and S. Chong, "Mobile data offloading: How much can wifi deliver?" *IEEE/ACM Trans. Netw.*, vol. 21, no. 2, pp. 536–550, Apr. 2013.

[8] A. Schulman *et al.*, "Bartendr: A practical approach to energy-aware cellular data scheduling," in *Proc. ACM MobiCom*, Sep. 2010, pp. 85–96.

[9] W. Hu and G. Cao, "Quality-aware traffic offloading in wireless networks," *IEEE Trans. Mobile Comput.*, vol. 16, no. 11, pp. 3182–3195, Nov. 2017.

[10] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani, "Energy consumption in mobile phones: A measurement study and implications for network applications," in *Proc. ACM SIGCOMM*, Nov. 2009, pp. 280–293.

[11] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck, "A close examination of performance and power characteristics of 4G LTE networks," in *Proc. ACM MobiSys*, Jun. 2012, pp. 225–238.

[12] W. Hu and G. Cao, "Energy optimization through traffic aggregation in wireless networks," in *Proc. IEEE INFOCOM*, May 2014, pp. 916–924.

[13] L. Yang, J. Cao, Y. Yuan, T. Li, A. Han, and A. Chan, "A framework for partitioning and execution of data stream applications in mobile cloud computing," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 40, no. 4, pp. 23–32, 2013.

[14] W. Zhang, Y. Wen, and D. O. Wu, "Energy-efficient scheduling policy for collaborative execution in mobile cloud computing," in *Proc. IEEE INFOCOM*, Apr. 2013, pp. 190–194.

[15] D. Huang, P. Wang, and D. Niyato, "A dynamic offloading algorithm for mobile computing," *IEEE Trans. Wireless Commun.*, vol. 11, no. 6, pp. 1991–1995, Jun. 2012.

[16] L. Xiang, S. Ye, Y. Feng, B. Li, and B. Li, "Ready, set, go: Coalesced offloading from mobile devices to the cloud," in *Proc. IEEE INFOCOM*, May 2014, pp. 2373–2381.

[17] L. Tong and W. Gao, "Application-aware traffic scheduling for workload offloading in mobile clouds," in *Proc. IEEE INFOCOM*, Apr. 2016, pp. 1–9.

[18] Y. Geng, W. Hu, Y. Yang, W. Gao, and G. Cao, "Energy-efficient computation offloading in cellular networks," in *Proc. IEEE ICNP*, Nov. 2015, pp. 145–155.

[19] C. Shi, V. Lakafosis, M. H. Ammar, and E. W. Zegura, "Serendipity: Enabling remote computing among intermittently connected mobile devices," in *Proc. ACM MobiHoc*, Jun. 2012, pp. 145–154.

[20] Z. Sheng, C. Mahapatra, V. C. M. Leung, M. Chen, and P. K. Sahu, "Energy efficient cooperative computing in mobile wireless sensor networks," *IEEE Trans. Cloud Comput.*, vol. 6, no. 1, pp. 114–126, Jan./Mar. 2018.

[21] L. Chen, S. Zhou, and J. Xu. (Mar. 2017). "Computation peer offloading for energy-constrained mobile edge computing in small-cell networks." [Online]. Available: https://arxiv.org/abs/1703.06058

[22] Y. Li, L. Sun, and W. Wang, "Exploring device-to-device communication for mobile cloud computing," in *Proc. IEEE ICC*, Jun. 2014, pp. 2239–2244.

[23] M. Jo, T. Maksymyuk, B. Strykhalyuk, and C.-H. Cho, "Device-to-device-based heterogeneous radio access network architecture for mobile cloud computing," *IEEE Wireless Commun.*, vol. 22, no. 3, pp. 50–58, Jun. 2015.

[24] B. Han, P. Hui, V. S. A. Kumar, M. V. Marathe, J. Shao, and A. Srinivasan, "Mobile data offloading through opportunistic communications and social participation," *IEEE Trans. Mobile Comput.*, vol. 11, no. 5, pp. 821–834, May 2012.

[25] L. Xu, C. Jiang, Y. Shen, T. Q. S. Quek, Z. Han, and Y. Ren, "Energy efficient D2D communications: A perspective of mechanism design," *IEEE Trans. Wireless Commun.*, vol. 15, no. 11, pp. 7272–7285, Nov. 2016.

[26] H. Luo, R. Ramjee, P. Sinha, L. Li, and S. Lu, "UCAN: A unified cellular and ad-hoc network architecture," in *Proc. ACM MobiCom*, Sep. 2003, pp. 353–367.

[27] H. Holma and A. Toskala, *WCDMA for UMTS: HSPA Evolution and LTE*. Haboken, NJ, USA: Wiley, 2007.

[28] S. Yi, S. Chun, Y. Lee, S. Park, and S. Jung, *Radio Protocols for LTE LTE-Advanced*. Hoboken, NJ, USA: Wiley, 2012.

[29] X. Zhuo, W. Gao, G. Cao, and S. Hua, "An incentive framework for cellular traffic offloading," *IEEE Trans. Mobile Comput.*, vol. 13, no. 3, pp. 541–555, Mar. 2014.

[30] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Trans. Syst., Man, Cybern.*, vol. SMC-4, no. 2, pp. 100–107, Jul. 1968.

[31] Z. Wang and J. Crowcroft, "Quality-of-service routing for supporting multimedia applications," *IEEE J. Sel. Areas Commun.*, vol. 14, no. 7, pp. 1228–1234, Sep. 1996.

[32] A. Juttner, B. Szviatovski, I. Mecs, and Z. Rajko, "Lagrange relaxation based method for the QoS routing problem," in *Proc. IEEE INFOCOM*, Apr. 2001, pp. 859–868.

[33] S. Guo, B. Xiao, Y. Yang, and Y. Yang, "Energy-efficient dynamic offloading and resource scheduling in mobile cloud computing," in *Proc. IEEE INFOCOM*, Apr. 2016, pp. 1–9.

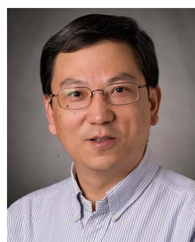
[34] *Monsoon Power Monitor*. Accessed: Mar. 20, 2015. [Online]. Available: http://www.msoon.com/

[35] *Android-x86 Project*. Accessed: Dec. 15, 2015. [Online]. Available: http://www.android-x86.org/

[36] *Tesseract Ocr*. Accessed: Feb. 2, 2016. [Online] Available: https://code.google.com/p/tesseract-ocr/

[37] *Cmu Sphinx*. Accessed: Feb. 17, 2016. [Online] Available: http://cmusphinx.sourceforge.net/

Yeli Geng (S'15) received the B.S. degree in computer science and technology from the Huazhong University of Science and Technology in 2008 and the M.S. degree in computer architecture from the Chinese Academy of Sciences in 2011. She is currently pursuing the Ph.D. degree with the Department of Computer Science and Engineering, Pennsylvania State University. Her research interests include energy management for smartphones, mobile cloud, and mobile networks.



Guohong Cao (S'98–M'03–SM'06–F'11) received the Ph.D. degree in computer science from The Ohio State University in 1999. Since 1999, he has been with the Department of Computer Science and Engineering, Pennsylvania State University, where he is currently a Distinguished Professor. His research interests include wireless networks, mobile systems, wireless security and privacy, and Internet of Things. He has published over 200 papers which have been cited over 18000 times, with an h-index of 70. He was a recipient of several best paper awards, the IEEE INFOCOM Test of Time Award, and the NSF CAREER Award. He has served on the Editorial Board of the IEEE TRANSACTIONS ON MOBILE COMPUTING, the IEEE TRANSACTIONS ON WIRELESS COMMUNICATIONS, and the IEEE TRANSACTIONS ON VEHICULAR TECHNOLOGY, and has served on the organizing and technical program committees of many conferences, including the TPC Chair/Co-Chair of the IEEE SRDS, MASS, and INFOCOM.