Design, Realization, and Evaluation of DozyAP for Power-Efficient Wi-Fi Tethering

Hao Han, Yunxin Liu, *Member, IEEE*, Guobin Shen, *Senior Member, IEEE*, Yongguang Zhang, *Senior Member, IEEE*, Qun Li, *Senior Member, IEEE*, and Chiu C. Tan, *Member, IEEE*

***Abstract*—Wi-Fi tethering (i.e., sharing the Internet connection of a mobile phone via its Wi-Fi interface) is a useful functionality and is widely supported on commercial smartphones. Yet, existing Wi-Fi tethering schemes consume excessive power: They keep the Wi-Fi interface in a high power state regardless if there is ongoing traffic or not. In this paper, we propose DozyAP to improve the power efficiency of Wi-Fi tethering. Based on measurements in typical applications, we identify many opportunities that a tethering phone could sleep to save power. We design a simple yet reliable sleep protocol to coordinate the sleep schedule of the tethering phone with its clients without requiring tight time synchronization. Furthermore, we develop a two-stage, sleep interval adaptation algorithm to automatically adapt the sleep intervals to ongoing traffic patterns of various applications. DozyAP does not require any changes to the 802.11 protocol and is incrementally deployable through software updates. We have implemented DozyAP on commercial smartphones. Experimental results show that, while retaining comparable user experiences, our implementation can allow the Wi-Fi interface to sleep for up to 88% of the total time in several different applications and reduce the system power consumption by up to 33% under the restricted programmability of current Wi-Fi hardware.**

***Index Terms*—802.11, mobile hotspot, power-efficient, software access point, Wi-Fi tethering.**

I. Introduction

WI-FI tethering, also known as a "mobile hotspot," means sharing the Internet connection (e.g., a 3G connection) of an Internet-capable mobile phone with other devices over Wi-Fi. As shown in Fig. 1, a Wi-Fi tethering mobile phone acts as a mobile software access point (softAP). Other devices such as laptops, tablet PCs, and other mobile phones can connect to the mobile softAP through their Wi-Fi interfaces. The mobile softAP routes the data packets between its 3G interface and its Wi-Fi interface. Consequently, all the devices connected to the mobile softAP are able to access the Internet.

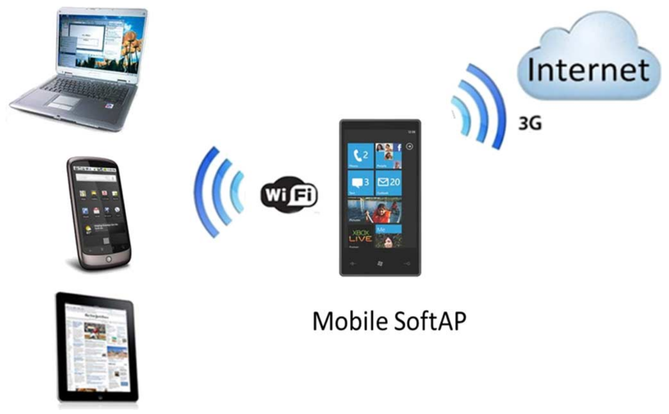


Fig. 1. Illustration of a typical setting of Wi-Fi tethering.

Wi-Fi tethering is highly desired. Main-streaming smartphones including iPhones (iOS 4.3+), Android phones (Android 2.2+), and Windows phones (Windows Phone 7.5+) all provide built-in support of Wi-Fi tethering. There were also many third-party Wi-Fi tethering tools with customized features in App markets. We believe there are two main reasons why Wi-Fi tethering is so desirable. First, cellular data networks provide ubiquitous Internet access over the world but the coverage of Wi-Fi networks is much limited. Second, it is common for people to own multiple mobile devices, but likely they do not have a dedicated cellular data plan for every device. As a result, it demands to share a data plan among multiple devices, e.g., sharing the 3G connection of an iPhone with a Wi-Fi only iPad. Wi-Fi tethering provides a convenient way to do this.

However, Wi-Fi tethering significantly burdens a smartphone's battery. When enabling tethering, the Wi-Fi interface always stays in high power state and never sleeps even when there is no data traffic going on. This increases the power consumption by one order of magnitude and reduces the battery life from days to hours (more details in Section II-A). To save power, a Windows phone automatically turns off Wi-Fi tethering if the network is inactive for a time threshold of several minutes. However, this method has two drawbacks. First, the Wi-Fi interface still operates in a high power state for the idle intervals less than the threshold, leading to waste of energy. Second, it harms usability. If a user does not generate any traffic for a time period longer than the threshold (e.g., while reading a long news article) and then starts to use the network again (e.g., by clicking another news link), the user will have

Manuscript received September 12, 2012; revised April 22, 2013 and August 27, 2013; accepted September 06, 2013; approved by IEEE/ACM Transactions on Networking Editor A. Capone. Date of publication October 09, 2013; date of current version October 13, 2014. This work was supported in part by the NSF under Grant CNS-1117412, CAREER Award CNS-0747108, and Grant CNS-1156574.

H. Han and Q. Li are with the Department of Computer Science, College of William and Mary, Williamsburg, VA 23185 USA (e-mail: hhan@cs.wm.edu; liqun@cs.wm.edu).

Y. Liu, G. Shen, and Y. Zhang are with Microsoft Research Asia, Beijing 100080, China (e-mail: yunxin.liu@microsoft.com; jackysh@microsoft.com; ygz@microsoft.com).

C. C. Tan is with the Department of Computer and Information Sciences, Temple University, Philadelphia, PA 19122 USA (e-mail: cctan@temple.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/TNET.2013.2283636

to go back to the smartphone and manually re-enable Wi-Fi tethering, which results in poor user experience.

The IEEE 802.11 standard defines the power saving mechanism for wireless stations in client mode, *ad hoc* mode, but not in AP mode. That is because traditional APs are externally powered by cables, so that power saving is not a crucial issue for those APs. However, old wisdom does not work for the battery-powered smartphones operating in Wi-Fi tethering. Hence, it is time to think about how to save power for smartphones working as softAPs.

Recently, Wi-Fi Direct specification introduces a power-saving protocol for Wi-Fi Direct devices acting as access points (APs). The protocol operates in the media access control (MAC) layer and allows APs to notify clients with newly defined messages when they are going to sleep. However, existing 802.11 devices including most smartphones cannot benefit from the new feature supplied by Wi-Fi Direct. Future mobile devices may have a chipset that can support Wi-Fi Direct. However, due to the lack of the programmability of Wi-Fi chipsets in Wi-Fi industry, a device vendor still may find it difficult to implement its own tethering solution independent of the one delivered by the chipset vendor. In this paper, we propose a new approach for device vendors and software developers to implement a power-efficient Wi-Fi tethering solution without underlying support. To demonstrate its efficacy, we design *DozyAP*, a system to reduce power consumption of Wi-Fi tethering on smartphones while still retaining a good user experience.

The key idea of DozyAP is to put the Wi-Fi interface of a softAP into sleep to save power. We measured the traffic pattern of various online applications used in Wi-Fi tethering. We find that the Wi-Fi network is idle for a large portion of total application time (more details in Section II-B), which means the AP could sleep during this idle time. Furthermore, we know that the cellular interface is typically slower than Wi-Fi interface. Thus, the Wi-Fi interface of a softAP could sleep while waiting for the data transmission through the cellular network. All of these indicate there are many opportunities to reduce softAP's power consumption. With DozyAP, a softAP can automatically sleep to save power when the network is idle and wake up on demand if the network becomes active again.

Putting a softAP into sleep imposes two challenges. First, without a careful design, it may cause packet loss. Existing Wi-Fi clients assume that APs are always available for receiving packets, so whenever a client receives an outgoing packet from applications, it will immediately send the packet to its AP. However, if the AP is in the sleep mode, this packet will be lost, even after the retries that occur at the low layers of the network stack. Second, putting an AP to sleep will introduce increased network latency and may impair user experience if the extra latency is user perceivable.

DozyAP addresses the first challenge with a sleep request-response protocol with which a softAP and its clients negotiate and agree on a valid sleep schedule. To avoid possible packet loss, a client will transmit packets only when the softAP is active, and buffer outgoing packets otherwise. To address the second challenge, we design an adaptive sleep scheme and limit the maximum sleep duration. Consequently, DozyAP is able to reduce power consumption of Wi-Fi tethering with negligible impact on the network performance. DozyAP does not require any changes to the 802.11 protocol and is incrementally deployable via software updates to mobile devices.

We have implemented the DozyAP system on existing commercial smartphones and evaluated its performance using various applications and the traces from real users. Evaluation results show that DozyAP can put the Wi-Fi interface of a softAP to sleep for up to 88% of the total time in several different applications. Due to the restricted programmability of current Wi-Fi hardware on smartphones, forcing a softAP to sleep or wake up consumes considerable overhead. Thus, DozyAP only saves power by up to 33% while increasing network latency by less than 5.1%.

It is noticed that tethering can be also enabled by USB, Bluetooth, and Wi-Fi in *ad hoc* mode. However, USB tethering has drawbacks that can only support one client. Also, connecting phones to devices such as tablets is not easy due to the constrained interface and complicated system configuration. Bluetooth suffers high energy consumption per bit transmission cost and low bandwidth [1], thus consuming more energy than Wi-Fi. *Ad hoc* mode of Wi-Fi is less used than the infrastructure mode in practice. The OS on many mobile devices including Windows phones, Android phones, and iPhones hides such a mode, preventing a device from connecting to an *ad hoc* network [2]. Due to the above reasons, those tethering methods are out of scope of this paper.

To the best of our knowledge, we are the first to study the power efficiency of a softAP in Wi-Fi tethering. The main contributions of this paper are the following.

- We study the characteristics of existing Wi-Fi tethering and present our findings. We show that current Wi-Fi tethering is power-hungry, wasting energy unnecessarily. We analyze the traffic patterns of various applications and identify many opportunities to optimize the power consumption of Wi-Fi tethering.
- We propose DozyAP to improve power efficiency of Wi-Fi tethering. We design a simple yet reliable sleep protocol to schedule a mobile softAP to sleep without requiring tight time synchronization between the softAP and its clients. We develop a two-stage adaptive sleep algorithm to allow a mobile softAP to automatically adapt to the traffic load for the best sleep schedule.
- We implement DozyAP system on commercially available off-the-shelf (COTS) smartphones and evaluate its performance through experiments with real applications and simulations based on real user traces. Evaluation results show that DozyAP is able to significantly reduce power consumption of Wi-Fi tethering and retain comparable user experience at the same time.

The rest of the paper is organized as follows. In Section II, we report our findings on existing Wi-Fi tethering, with focus on the power consumption and the traffic patterns of various applications. Based on the findings, in Section III we design DozyAP that can schedule a mobile softAP to sleep and present the design details. We describe our implementation in

Section IV and evaluate it in Section V. We discuss limitations of DozyAP and future work in Section VI, survey the related work in Section VII, and conclude in Section VIII.

II. Understanding Wi-Fi Tethering

In this section, we report our findings on the characteristics of Wi-Fi tethering through real measurements on existing commercial smartphones. We focus on two characteristics: the power consumption and the traffic pattern of various online applications used in Wi-Fi tethering. Furthermore, we provide some background on Wi-Fi power management to set up the context of our DozyAP design.

A. Power Consumption

We first measured the power consumption of existing commercial smartphones regarding the Wi-Fi tethering. We imposed *no traffic* but simply turned on/off the Wi-Fi tethering, i.e., the Wi-Fi interface and the 3G interface were kept on but idle. We used a Nexus One phone (Android 2.3.6), an HTC HD7 Windows Phone (Windows Phone 7.5), and an iPhone 4 (iOS 4.3.5) for experiments. For the Nexus One and the HTC HD7, we measured the power consumption of the whole system using a Monsoon Power Monitor [3]. However, it is impossible to use the Monsoon Power Monitor to measure the power consumption of the iPhone without damaging the phone. Instead, we used MyWi 5.0 [4], a very popular third-party Wi-Fi tethering tool on iOS that is able to tell the draining current of the battery, to measure the power consumption of the iPhone 4 when Wi-Fi tethering is enabled. In all the experiments, the display was turned off.

With Wi-Fi tethering disabled, the power consumption of all smartphones was pretty low because the Wi-Fi and 3G interfaces were in sleep for most of the time. The average power consumption was only 20 mW for the Nexus One and 30 mW for the HTC HD7, respectively. For the iPhone 4, MyWi read a draining current of 6 mA, equivalently a power consumption of 22 mW.

With Wi-Fi tethering enabled, the power consumption of the smartphones increased significantly. Fig. 2 shows the results of the Nexus One and the HTC HD7 smartphones. We can see that both smartphones operated in a high power state constantly even though there was no traffic at all. There are periodic spikes in the plots caused by periodic Wi-Fi beacon transmissions. On average, the power consumption was 270 mW for the Nexus One and 302 mW for the HTC HD7. For the iPhone 4, MyWi read a draining current of 90 mA, equivalently a power consumption of 333 mW. While software reading may not be as accurate as the Monsoon Power Monitor, the result still clearly indicates that Wi-Fi tethering on the iPhone 4 has similar power consumption as on the Nexus One and the HTC HD7. Also, we tested phones with the latest OS versions such as Android v4.x. The power consumption was similar.

The above results demonstrate that existing Wi-Fi tethering schemes on all the three mobile platforms are power-hungry. They consume *an order of magnitude more power than necessary* when there is no ongoing traffic, i.e., in idle network state. In Section II-B, we will show that such idle states occur frequently in various typical Internet access scenarios.

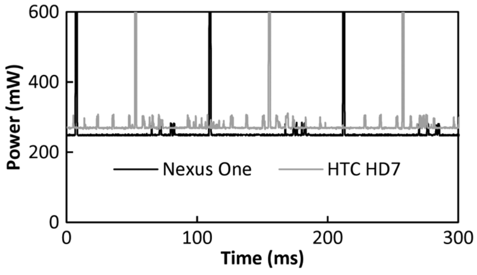


Fig. 2. Measured power consumption of Wi-Fi tethering on Nexus One and HTC HD7 without background traffic.

Intuitively, the Wi-Fi interface should be put to sleep when the Wi-Fi network is idle. As the battery is a very scarce resource on smartphones, this calls for a power-efficient Wi-Fi tethering solution and motivates us to conduct the work in this paper.

B. Traffic Pattern

Next, we study how frequently the Wi-Fi network is actually in an idle state and how long the idle state typically lasts. We enabled Wi-Fi tethering on a Nexus One smartphone with a China Unicom 3G connection. A Wi-Fi client is connected to such a mobile softAP. On the client side, we launched various applications to access the Internet, and they are used normally. In the meantime, we used a Lenovo T61 laptop running Linux 2.6.32 as a Wi-Fi sniffer to capture all the packets exchanged between the client and the softAP. We studied two different clients: a Nexus One smartphone and a Wi-Fi version iPad 2. Seven applications were measured including news reading, online book reading, video streaming, search, Map, e-mail, and RSS.

Note that some Web sites detect the type of client devices and return different content for different device types. For example, when the Nexus One smartphone is used, Baidu News automatically redirects to its mobile version that returns less complex Web pages than the normal version. Similarly, Youku streams low-bit-rate video clips to the Nexus One smartphone, but high-bit-rate ones of the same videos to the iPad 2. As a result, the same application may behave differently on different devices. For each application, we study the traffic patterns by analyzing the packet interarrival time of all the captured packets.

Fig. 3 shows the results of the Nexus One. Due to the space limitation, we omit the iPad 2 case, which also has similar results. We first study the distribution of packet interarrival intervals in the total application time, which indicates the period from the first packet to the last one. The left figure in Fig. 3 shows the cumulative distribution function (CDF) for all the applications, where the y-axis depicts the percentage of packets with interpacket intervals less than or equal to a specific value in the x-axis to the total application time. To make the curves easy to read, we only show the data for the time intervals less than 1 s. We can see that the intervals under 200 ms only take less than 30% of the total application time for all the applications on the Nexus One. For the iPad 2, the corresponding number is 35%. For some applications, these intervals consume as low as 20% or even less than 10% on the Nexus One or the iPad 2. If we consider the network "idle" during the packet interarrival intervals larger than 200 ms, then we can say that the Wi-Fi network was idle for 70%–90% of the total application time. This

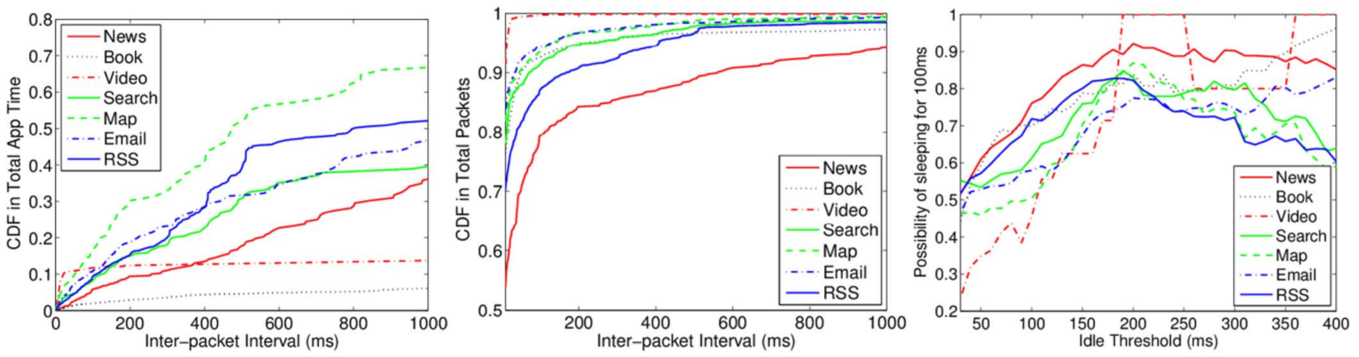


Fig. 3. Traffic pattern of seven applications. From left to right: CDF of packet interarrival intervals in total application time; CDF of packet interarrival intervals in total packets; probability of sleeping for 100 ms after an idle threshold.

shows that these applications only spent a small portion of time for the Internet access and their network traffic is very sparse and bursty.

There are two main reasons for the above findings. First, all the applications consist of two phases: a content fetching phase and a content consuming phase. Once users download some content from a remote server (e.g., a Web server), they need to spend time to consume the content (e.g., reading the text). The content consuming time may vary from seconds to tens of seconds to even minutes. During such a time, the network is mostly idle. In the e-mail case, replying to e-mails and composing new ones also result in significant network idle time. Secondly, the bandwidth of 3G is much lower than that of Wi-Fi. According to 3G Test [5], 3G typically offers 500 kb/s–1 Mb/s downlink throughput for US carriers, but the Wi-Fi offers much higher data rates (54 Mb/s for 802.11a/g and 300 Mb/s for 802.11n). Furthermore, 3G has much higher round-trip times (RTTs), ranging from 200 to 500 ms [5], than that of Wi-Fi. Consequently, the Wi-Fi interface of a softAP in Wi-Fi tethering often has to wait for data to be received from or transmitted over 3G. Such a waiting period will put the Wi-Fi interface in an “idle” state.

While the results are somehow as expected for those interactive applications, we are surprised to see that similar patterns were observed in the video streaming case. Even for the iPad 2 on which a high-bit-rate video clip was continuously played back, the packet interarrival intervals larger than 200 ms took more than 60% of the total streaming time. After carefully checking the captured trace, we found that it used a large video buffer when streaming video clips. It aggressively downloaded video content until the video buffer was full. Then, it stopped video downloading. The downloaded bits were constantly consumed and drained from the buffer. Once the buffer level became lower than a threshold, the aggressive downloading was resumed again.

The large percentage of the Wi-Fi idle time in these applications demonstrates that there are many opportunities to reduce the power consumption of Wi-Fi tethering. During the large network idle intervals, the Wi-Fi interface of a mobile softAP should sleep to save power. More specifically, there are two kinds of network idle intervals that we exploit in this paper. The first one is the long network idle intervals resulting from the user content consuming behavior. The second one is the relatively shorter network idle intervals that occur during the content downloading. The latter case is mainly caused by the RTTs of 3G: After a client sends a request packet to a remote server, it has to wait for at least an RTT of 3G to get the first response packet from the server. For example, to access a Web server, we can typically see two such network idle intervals: one for the DNS name lookup for the server and the other for making a TCP connection to the server.

We further study how putting a softAP to sleep can affect the network performance. The middle figure in Fig. 3 shows the CDF of packet interarrival interval in total packets, where the y-axis depicts the percentage of the packets whose interpacket interval is less than or equal to a specific value in the x-axis to the total number of packets. We can see that the interpacket intervals under 150 ms cover more than 80% of all the packets for all the applications. For some applications, the number is as high as 90% or even 95%. This means that if we use an idle threshold of 150 ms to decide whether to put the softAP to sleep or not, most of the packets will not be affected. The right figure in Fig. 3 further shows the probability that the softAP could successfully sleep for extra 100 ms after waiting for different idle thresholds. We found that 150 ms was a good threshold to optimize the energy saving and minimize the incurred network latency in terms of sleeping probability and the number of involved packets.

All the above findings demonstrate that a mobile softAP *could* and *should* sleep to save power in Wi-Fi tethering, which provides the foundation for our DozyAP design.

C. Background: Wi-Fi Power Saving

The IEEE 802.11 standard defines a power saving mode (PSM) to save power for Wi-Fi clients [6]. In PSM, the Wi-Fi interface of a client always stays in a very low power state to save power and cannot receive or transmit any data. If an AP needs to send some packets to a client in PSM, the AP will first buffer the packet and set the Traffic Indication Map (TIM) in its beacons, which are broadcast typically every 100 ms. A PSM client periodically (i.e., every a certain number of beacon intervals) wakes up to listen to beacons. If the client detects a TIM for itself, it sends an individual PS-Poll frame to notify the AP of sending a buffered packet. Otherwise, it goes to sleep immediately. When the AP transmits a buffered packet to the client, an MORE flag in the header of the data frame is set if the AP has more packets for the client. This allows the client to decide when to stop sending PS-Poll frames.

On the Nexus One, the above static PSM scheme is called "PM_MAX." PM_MAX allows a client to sleep as long as the AP does not have any packet for it. However, this leads to long network latency and hence low network efficiency. Therefore, on the Nexus One, another power saving scheme called "PM_FAST" is used. In PM_FAST, a client stays in active unless its Wi-Fi interface is idle for a threshold of 200 ms. Then, it sends a Null-Data frame with power management flag set to 1 to tell its AP that it will sleep soon. If such a frame is acknowledged, the client is able to go to sleep since all packets destined for it will be buffered at AP. Otherwise, the client cannot go to sleep. Once the client detects a TIM for itself from beacons, it notifies the AP that it is active and ready to receive packets by a single Null-Data frame with power management flag set to 0. Many other Wi-Fi devices today also implement a similar scheme known as adaptive PSM [7]. PM_FAST is designed for fast system response, and PM_MAX is more suitable for background services. By default, Nexus One smartphones use PM_FAST if the screen is on, and switch to PM_MAX if the screen is turned off.

III. DozyAP Design

Guided by the findings in Section II, we design the DozyAP system that aims to reduce the power consumption of Wi-Fi tethering by putting the Wi-Fi interface of a mobile softAP into sleep mode whenever possible. We present the detail of DozyAP and the rationale of the design decisions. We start with a single client and describe the extension to support multiple clients later on.

A. Scheduling a SoftAP to Sleep

We design a simple *sleep request-response* protocol to enable a mobile softAP to safely sleep in Wi-Fi tethering according to its own best schedule. While the softAP can sleep at will, it can only do so when the client agrees, to avoid possible packet loss. Therefore, before entering sleep mode, a softAP sends a *sleep request* to its client. If the client sends back a *sleep response* to accept the sleep request, the softAP then enters the sleep mode. Otherwise, it will continue to stay in the active state. Fig. 4 shows a typical interaction procedure between a softAP and a client. At time t_1, the softAP decides to sleep and enters sleep mode at time t_2 after receiving the client's agreement. When the sleep times out at time t3, the softAP wakes up and continues to communicate with the client.

Packet Format: Both the sleep request and the sleep response are transmitted as a normal Wi-Fi unicast data packet. This design does not require any modification on existing Wi-Fi standard and is easy to implement. Fig. 5 shows the packet format. The sleep protocol is implemented directly on top of the underlying link layer without TCP/IP headers in the middle to reduce the overhead. The sleep protocol packets have three fields. The "Type" field indicates the packet type: "0×1" means *sleep request*, and "0×2" means *sleep response*. The "Sequence Number" field is a unique ID to identify a sleep request-response pair. It starts from zero and increases by one for every new sleep request. The "Time Duration" field specifies how long (in milliseconds) the softAP requests to sleep. All the sequence

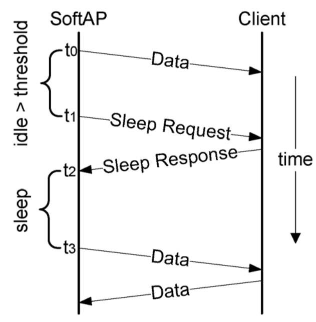


Fig. 4. Interaction of a softAP and a client using the sleep request-response protocol.

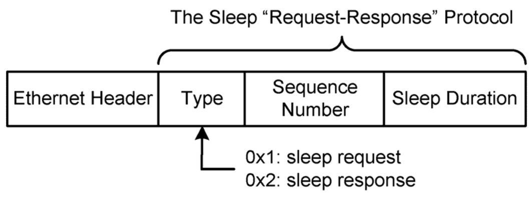
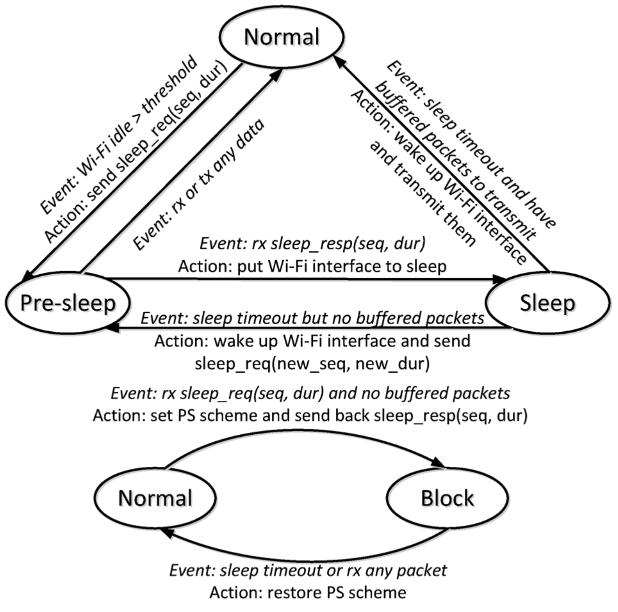


Fig. 5. Packet format of the sleep protocol.


Normal
Pre-sleep
Sleep
Event: Wi-Fi idle > threshold
Action: send sleep_req(seq, dur)
Event: rx or tx any data
Event: sleep timeout and have buffered packets to transmit
Action: wake up Wi-Fi interface and transmit them
Event: rx sleep_resp(seq, dur)
Action: put Wi-Fi interface to sleep
Event: sleep timeout but no buffered packets
Action: wake up Wi-Fi interface and send sleep_req(new_seq, new_dur)
Event: rx sleep_req(seq, dur) and no buffered packets
Action: set PS scheme and send back sleep_resp(seq, dur)
Normal
Block
Event: sleep timeout or rx any packet
Action: restore PS scheme
Fig. 6. State machine for a softAP (top) and a client (bottom).

numbers and time durations are decided by the softAP. When the client accepts a sleep request, it simply copies the sequence number and time duration from the sleep request packet into its sleep response packet. Sleep response packets are used only for accepting a sleep request. If the client does not agree the softAP to sleep, it simply chooses not to send out the sleep response. There is only one case that the client will decline the sleep request of the softAP: it has more data packets to transmit. In that case, the client will send a data packet, instead of the sleep response packet, to the softAP. The softAP then learns that the client has declined the sleep request and thus stays active. This design reduces the overhead of the sleep protocol because a sleep response packet is transmitted only when it is necessary.

State Machine: Fig. 6 shows the state machine of a softAP and a client. A softAP has three states: *Normal*, *Pre-sleep*, and

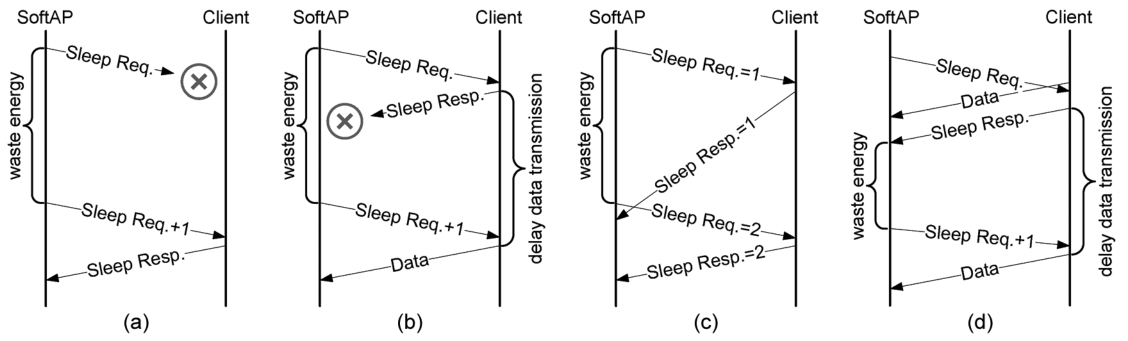


Fig. 7. Abnormal cases. (a) Sleep request is lost. (b) Sleep response is lost. (c) Sleep response is delayed. (d) Data packet is delayed.

Sleep. In the *Normal* state, the softAP is active and can transmit and receive packets normally. When the Wi-Fi interface of the softAP is idle for a time period larger than a predefined threshold, the softAP sends a sleep request packet to its client with a sequence number *seq* and a time duration *dur*. Then, it enters the *Pre-sleep* state and waits for a sleep response. If it receives the right sleep response with the same sequence number *seq* and time duration *dur*, it will put its Wi-Fi interface into sleep mode and enter the *Sleep* state; if it receives any packet other than the expected sleep response, it will go back to the *Normal* state and invalidate the sleep request. In the *Sleep* state, the Wi-Fi interface of the softAP is turned to sleep to save power. Thus, the softAP cannot receive any packets over Wi-Fi. If it receives any data from its 3G interface, it will buffer them during the whole period of *Sleep* state. When the sleep timeout expires and some data are buffered, the softAP wakes up its Wi-Fi interface, switches to *Normal* state, and transmits the buffered data to the client. Otherwise, it moves back to the *Pre-sleep* state, sends out another sleep request with a new sequence number *new_seq* and a new time duration *new_dur*, and waits for the next sleep response.

The state machine of a client has only two states: *Normal* and *Block*. In the *Normal* state, the client communicates with the softAP as normal. It may use any Wi-Fi power saving schemes such as PM_MAX, PM_FAST, or none. If the client receives a sleep request from the softAP and agrees, i.e., it does not have any packets to transmit, it tentatively sets its Wi-Fi power saving scheme to PM_MAX, sends back a sleep response to the softAP, and enters the *Block* state. Note that by switching to PM_MAX, the firmware automatically sends a Null-Data frame to tell the softAP that the client is going to sleep. Done this way, the client can go to sleep as quickly as possible (i.e., immediately after the sleep response). In contrast, if a client that uses PM_FAST does not change to PM_MAX before sending the sleep response, it will wait for a 200-ms idle period to send out a Null-Data frame. However, as the softAP has already entered the *Sleep* state once receiving the sleep response, it cannot receive the Null-Data frame afterwards. As a result, the Null-Data frame is not acknowledged, and the client cannot go to sleep as supposed. Therefore, it is essential for clients to switch to PM_MAX to maximally save power. In the *Block* state, the Wi-Fi interface of the client is in power saving mode, and the client knows that the softAP is sleeping. Thus, it blocks all the packet transmissions by buffering all the packets from applications. If the sleep schedule times out or the client receives a data packet from the softAP, it restores the previous power saving scheme (e.g., back to PM_FAST) and moves back to the *Normal* state. At this time, both the softAP and the client can communicate normally. Otherwise, the softAP will send a new sleep request to try to sleep again.

B. Synchronization

One advantage of the sleep request-response protocol is that it does not require tight time synchronization between the softAP and the client. If the softAP and the client can synchronize their time perfectly, they can coordinate their sleep scheduling to avoid packet loss without transmitting any extra sleep request and response packets. However, this is hard to achieve in practice. Although very fine-grained hardware timestamps (e.g., at microsecond granularity) exists at link layer, such timestamps are segregated inside firmware and are not available to the Wi-Fi driver and applications. It is possible to do time synchronization by explicitly exchanging packets with timing information between the softAP and the client. Such time synchronization must be done periodically due to clock drift, which increases power consumption. Due to these considerations, we intentionally avoided the time synchronization approach.

Interestingly, the proposed sleep protocol can achieve loose synchronization between a softAP and a client, with a desirable property: *The client will never conclude that the softAP is awake while it is sleeping*. Therefore, our approach will not lead to packet loss that would arise from wrong attempts of sending packets while the softAP is actually in sleep mode. In normal case, this is obvious because the softAP will sleep only after it receives a sleep response from the client. However, due to the uncertainty and complexity of wireless communication, the sleep protocol may not work as smoothly as expected. We analyze several possible abnormal cases and their consequences, as illustrated in Fig. 7.

Packet Loss: First, sleep request or response packets may be lost during their transmissions. For example, a sleep request may be lost. In this case, the softAP will stay in active. If later on the client or the softAP has data to transmit, they start to communicate as normal. Conversely, the softAP will send out a new sleep request after the network remains idle for a period longer than the predefined idle threshold, as shown in Fig. 7(a). The worst effect of losing a sleep request is that the softAP would waste some energy for staying in unnecessary active state between two successive sleep requests. Similarly, if a sleep response is lost, the softAP also has to stay in active until the next sleep request. However, in this case, as the client has concluded that the softAP is in sleep, it will stay in the Block state and start to buffer packets. Thus, it may further incur extra delay up to the idle threshold to the client, as shown in Fig. 7(b).

Packet Out-of-Order: Second, packet transmission may be delayed due to the hardware queuing and wireless contention. As a result, there is a slight chance that packets may not arrive at their destinations in the expected order. For example, Fig. 7(c) shows the case that a sleep response is delayed by the client's hardware. The softAP receives sleep response 1 after sleep request 2 is sent out. In this case, the softAP just ignores sleep response 1, but it has to stay in active between the two sleep requests. Fig. 7(d) shows a more complex case. The client has already passed a packet to the firmware, and the packet is waiting for transmission in the hardware queue. At this moment, the client receives a sleep request from the softAP. As the client does not have more data to transmit, it replies a sleep response. However, once the softAP receives the data packet, it resets its idle timer, stays in active, and ignores the sleep response. Consequently, the softAP and the client are out of sync: The softAP stays in active, wasting energy, but the client assumes the softAP is in sleep and delays its packets transmission.

Based on the above analysis, we can see that those abnormal cases would at most cause some overhead in energy and transmission latency, but they would not break the desired synchronization property of our sleep protocol. A client will never try to send packets when the softAP is actually in sleep. The softAP and the client may run out of sync temporally, but will always resume sync after the subsequent sync response. This demonstrates the robustness of our sleep protocol. In addition, in the Wi-Fi tethering, the bottleneck is usually the 3G connection as its bandwidth is much lower than that of Wi-Fi. The necessity airtime of Wi-Fi is usually light, and thus, the above abnormal cases can rarely happen.

It is noticed that the sleep request-response protocol operates above the MAC layer, thus all the MAC-layer frames such as beacons and Null-Data cannot be seen by the protocol. That is why DozyAP has to explicitly exchange sleep requests and responses to negotiate a sleeping schedule. If the protocol is applied to the MAC layer, the negotiation can be performed through existing MAC-layer frames. For example, if a client does not have data to transmit, it sends a null-data frames with power management flag set to 1. Once observing that all clients have been in sleep mode, the AP automatically turns off. Owing to the tight time synchronization existing at MAC layer, both AP and clients could wake up almost at the same time when the predetermined sleep timeout or next beacon is due. Then, AP could send buffered packets and clients could upload buffered packets.

C. Adaptive Sleeping

Our sleep protocol allows a softAP to sleep. The next natural question is: How long should it sleep? The simplest solution is certainly to sleep for a fixed interval. However, it is difficult to determine such an interval because the RTT of 3G connection varies and the packet arrival time is irregular. In our design, we come up with an adaptive sleep scheduling algorithm to adapt to the traffic pattern and also the 3G network property. Our adaptive sleep algorithm consists of two stages, namely a *short sleep* stage and a *long sleep* stage, that are designed to exploit the two distinctive phases (i.e., the content downloading phase and the content consumption phase) of interactive applications, respectively.

```
1: // Parameters:
2: thresh, min, max, init, step, cur, pre, thresh_l, long;
3: ACTIVE:
4: measure the Wi-Fi network idle time;
5: SHORT_SLEEP:
6: if (Wi-Fi network idle time > thresh)
7:     first = true;
8:     sleep for a time period of init;
9: while (1)
10:     cur = first? init : (cur + step); first = false;
11:     if receive or transmit a packet
12:         if (cur <= init)
13:             init = max(init – step, min);
14:         pre = cur; goto ACTIVE;
15:     if ((cur > init + step) && (pre > init + step))
16:         init = min(init + step, max);
17:     if (cur >= thresh_l)
18:         pre = cur; goto LONG_SLEEP;
19:     sleep for a time period of step;
20: LONG_SLEEP:
21: while (1)
22:     sleep for a time period of long;
23:     if receive or transmit a packet;
24:         goto ACTIVE;
```

Fig. 8. Two-stage adaptive sleep algorithm.

Sleep Algorithm: Fig. 8 shows how the two-stage adaptive sleep algorithm works. The basic idea is to probe the optimal sleep interval such that the softAP can wake up shortly before a packet arrives. Starting with an initial conservative sleep interval, the sleep interval is gradually increased, at a conservative pace, until a packet has arrived during the last sleeping. Then, the initial sleep interval is updated dynamically. For sake of easier expression, all the successive sleep slots are collectively called a sleep cycle.

More concretely, when the Wi-Fi interface remains idle for a time period of *thresh*, the softAP will enter the short sleep stage. It first sleeps for a time period of *init*, which equals to *min* initially. When the softAP wakes up, it either goes back to the ACTIVE mode if there are pending outgoing or incoming packets, or continues to sleep for a fixed interval of *step*. Depending on the real packet arrival pattern, the length of the sleep cycle may become longer and longer between two subsequent wakeups. The sleep period can be expressed as $init + N * step$, where N is the number of continuous sleep slots after the first sleep slot of *init*.

As waking the Wi-Fi hardware up introduces certain energy overhead [8], it is desirable to reduce the number of unnecessary wakeups. This calls for a good *init* value that can let the softAP sleep as long as possible while still being able to wake up in time, i.e., to avoid or shorten the probing process. We determine the *init* value by exploiting the sleep history. We use parameters *cur* and *pre* to track the gross length of all successful sleep slots in the current sleep cycle and that in the previous sleep cycle, respectively. That is, we have *cur* equal to $init + (N-1) * step$ because a sleep cycle is always ended up by a false sleep slot during which a packet has arrived and been buffered. Parameter *pre* is simply a running record of the previous *cur*. Based on the

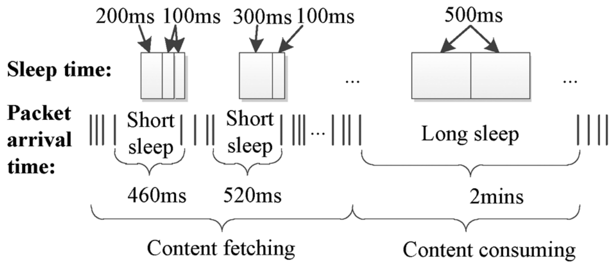


Fig. 9. Short sleep and long sleep example.

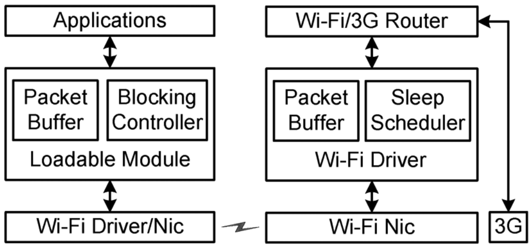


Fig. 10. Implementation architecture of *(left)* the client part and *(right)* the softAP part.

values of *cur* and *pre*, we adjust the value of *init* with a simple algorithm INIT_UPDATE as follows: If both *cur* and *pre* are greater than current *init* plus *step*, we increase *init* by *step* for the next sleep cycle. If *cur* is less than or equal to current *init* minus *step*, we decrease *init* by *step*. To avoid excessive latency that may be caused by an overly greedy *init* value, we cap it to the value of *max*. In SHORT_SLEEP stage, if the softAP has continuously been in sleep for a time period of *thresh_l*, it goes to LONG_SLEEP stage. In LONG_SLEEP stage, the softAP simply sleeps for a time period of *long* periodically until it quits from sleep to communicate with the client.

Example: Fig. 9 illustrates the algorithm with a concrete example. Some details such as the time for waking up the Wi-Fi interface between continuous sleep slots are omitted for sake of easier reading. Assuming the current value of *init* is 200 ms and the value of *step* is 100 ms, in the first short sleep circle (the 460-ms one), after *thresh* (150 ms) idle time for triggering sleep request-response protocol, the softAP will first sleep for 200 ms, followed by two 100-ms sleep slots. Suppose the value of *init* is then qualified to increase to 300 ms. In the second short sleep circle, after 150 ms idle time, the softAP will first sleep for 300 ms followed by one more 100-ms sleep slot. In the content consuming period, after sleeping for a time period of *thresh_l*, the softAP enters the long sleep stage and periodically sleeps for a time period of *long* (500 ms).

The above algorithm is specially designed for the traffic patterns of typical applications in Wi-Fi tethering as shown in Section II-B. The short sleep stage is designed for the softAP to sleep between the time when the client sends out a request to a remote server and the time when the first response packet from the remote server is received. That duration is roughly an RTT of the 3G connection (typically hundreds of milliseconds [5]). The purpose of the *init* parameter is exactly to estimate the 3G's RTT in an elegant way, based on the length of last two sleep cycles. Note that our algorithm is conservative in the sense that it tries to reduce the energy consumption under minimal impairment to user experience, i.e., extra latency incurred. We decrease the value of *init* quickly, by considering only the length of the current sleep cycle, but increase the value of *init* slowly by considering the length of both the current and the previous sleep cycles. In addition, we use the parameter *thresh* to prevent the softAP from entering the short sleep stage during burst data transmission period (e.g., multiple response packets from a remote server for the same client request such as fetching a picture. In Section IV, we describe the parameter values used in our implementation.

In summary, our sleep algorithm automatically adapts to the traffic pattern of applications and achieves a good balance between power saving and network performance.

D. Supporting Multiple Clients

DozyAP can support multiple clients by repeatedly applying the sleep request-response protocol to each client. A client goes to sleep once it agrees to the AP's sleep request. A softAP can sleep only if it receives the sleep responses from all the clients. If some clients replied to a sleep response but other clients did not, the softAP has to stay awake in this case. This design makes sense because some clients may have data to send and the softAP must serve those clients. It is expected that the softAP sleeps less and consumes more power in the multiclient case. However, extending DozyAP to support multiple clients will not break the synchronization property of the sleep protocol: *No client will send a packet when the softAP is in sleep*. Note that we considered the possibility of broadcasting the sleep requests as it can obviously reduce the overhead of the sleep protocol. However, we do not take this approach for two reasons. First, broadcast packets are less reliable because they are transmitted without link-layer retransmissions. Second, the clients in PSM likely cannot receive broadcast packets, whereas the unicast packets will be buffered in AP's hardware transmission queue until clients wake up from PSM. Therefore, the improvement of using broadcast is expected to be very small in multiclient case, and for single-client case, it is worse than using unicast.

Another minor issue with the multiclient case is the beacon. In our design, a softAP does not send out beacons in the sleep mode. Thus, a new client cannot join the Wi-Fi network when the softAP is in sleep. However, the softAP sends out periodic beacons when it is active. Even in long sleep stage, it still wakes up periodically and can send out beacons. Consequently, a new client is still able to find the softAP but may experience slightly longer latency. As this only happens when a new client joins the network, we think it is acceptable.

IV. Implementation

We have implemented the DozyAP system on a Nexus One smartphone running Android 2.3.6, with a Wi-Fi chipset of Broadcom BCM4329 802.11 a/b/g/n [9].

The overall architecture consists of two parts: the softAP part and the client part, as shown in Fig. 10. The softAP part is directly modified from the open-source Wi-Fi driver in which we embedded the sleep request-response protocol and the two-stage adaptive sleep algorithm. When the softAP is in sleep state, all the packets received from 3G interface are buffered. The client part is implemented as a loadable module where a packet buffer is implemented, together with a blocking controller to decide if and when application packets must be buffered. In our prototype, we use a special Ethernet type of `0xffff` (a reserved value that should not be used in products) for the packets of the sleep request-response protocol. In real deployment, other approaches can be used to implement the sleep protocol, e.g., using dedicated IP packets rather than the special Ethernet type. We use *netfilter* to intercept all the outgoing packets and to detect the packets of sleep requests and response. Implementing the client part as a loadable module does not require any modifications to the source code of the client OS. This makes it easy to deploy DozyAP on different types of client devices.

Putting a Mobile softAP Into Sleep: One practical difficulty we met is how to put a mobile softAP to sleep. On smartphones (Nexus One and other types of smartphones), most Wi-Fi MAC-layer functionalities are implemented in the firmware running on the Wi-Fi chipset, not in the CPU-hosted Wi-Fi driver. When Wi-Fi tethering is enabled, the firmware keeps the Wi-Fi always in a high power state. There is no interface available to change the power states. After trying many methods, all that we can do in the driver is to turn on/off the Wi-Fi interface when the softAP decides to wake up/sleep. By modifying the source of the driver, we hide the fact that the Wi-Fi interface is turned off. Thus, applications and the OS can work as normal as if the Wi-Fi interface is always on.

Energy Overhead of Turning On/Off the Wi-Fi: It costs extra energy to switch on/off the Wi-Fi interface. We measured such energy overhead on a Nexus One smartphone, and Fig. 11 shows the measurement results. Initially, the Wi-Fi interface was off. Then, we turned on the Wi-Fi interface for 100 ms and turned it off again. We can observe two artifacts: First, when the Wi-Fi interface is merely turned on without transmitting any packet, the system stays in an average power state of 400 mW, which is higher than the normal power consumption of Wi-Fi tethering in the idle case as shown in Fig. 2. We think this part of overhead is caused by the CPU and I/O operations for waking up the Wi-Fi interface. Second, when the off command is issued, the power consumption reduced immediately, but remains at a power level as high as 150 mW for about 1 s before entering a very low-power state of 10 mW. This finding is similar to what the authors reported in [8]. They pointed out that when the Wi-Fi interface goes to sleep, it first enters a "light sleep" state and then enters a "deep sleep" state after some time. We cannot control this behavior, but it significantly affects how much power we can save. We want to point out that this is a platform-specific limitation caused by the restricted programmability over the Wi-Fi hardware. Our design itself does not impose any limitation. If the smartphone can incur less wakeup overhead or enter the "deep sleep" state more quickly, our approach can save much more power.

Parameter Values: We determine the parameter values based on the real-world traces described in Section II. All the parameters are set in a conservative way to handle the variations of network conditions. We set *thresh* to 150 ms for triggering a softAP to enter sleep mode. With this parameter, the chance of sleeping is maximized, and the number of involved packets is limited to avoid introducing more network latency. Due to the overhead of wakeup, the sleep duration less than 100 ms may not yield much power saving. Thus, we set *min* to 100 ms. Considering the RTT of 3G and to limit the maximum extra latency, we set *max* to 500 ms. The value of *init* varies between 100 and 500 ms. We set *thresh_l* to 3 s for switching to the long sleep stage where the softAP periodically sleeps for 500 ms (i.e., *long* equals to 500 ms). It should be noticed that different Wi-Fi chipsets may have the different overhead of waking up, thereby affecting the choice of the parameter values above. The guideline is as follows: If the overhead is larger, then the parameters should be adjusted to avoid sleeping too frequently. Otherwise, it is better to set to gain more power saving.

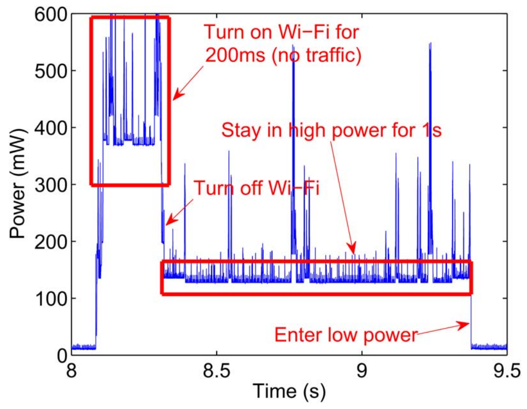


Fig. 11. Power consumption of switching on/off the Wi-Fi interface.

V. Evaluation

We evaluate the performance of DozyAP by answering the following questions. 1) How much power can DozyAP save for a mobile softAP in various applications? 2) What is the impact of DozyAP on client-side power consumption? 3) How much extra latency does DozyAP introduce? 4) How much power can be saved in multiclient case? 5) What is the performance degradation if clients are not changed?

A. Experiment Setup

Hardware Devices: We used a Nexus One smartphone as a softAP with a China Unicom 3G connection (WCDMA), and another Nexus One smartphone as a client to run applications. Both smartphones run Android 2.3.6. We used a Monsoon Power Monitor [3] to measure the power consumption. We repeated every experiment for at least five times to compute average results.

Applications and Methodology: We used five of the applications described in Section II, including news reading, book reading, video streaming, search, and map. To make the experiments repeatable, we analyze the captured trace of the applications to find out all the HTTP requests contained in the traces except video streaming. Then, we wrote a test program in Java to send out those HTTP requests with the exact same order and timing as the traces. The program uses the WebView class in the WebKit package [10]. Thus, we could easily repeat every experiment. For the video streaming, we manually played the same video clip.

Traces: To evaluate DozyAP with more diverse and realistic traffic patterns, we asked the authors of MoodSense [11] for the traces collected from real users. In MoodSense, the authors conducted a two-month field study with 25 iPhone users and collected their network traffic everyday using tcpdump [12]. We selected the traces of the top eight most active 3G users. For each

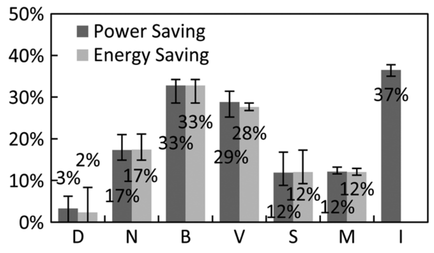


Fig. 12. Power saving and energy saving of the softAP in idle (I), busy download (D), and the five applications of news reading (N), book reading (B), video streaming (V), search (S), and map (M).

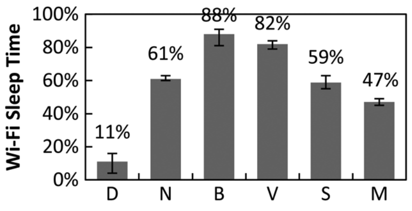


Fig. 13. Wi-Fi interface sleep time of the softAP in idle, busy download, and the five applications.

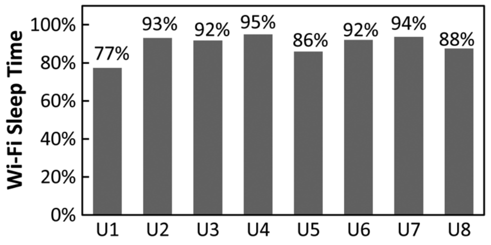


Fig. 14. Wi-Fi interface sleep time calculated based on the traces of eight real users.

of them, we further selected the trace of the day when the user generated the largest 3G traffic volume. We used the eight-day traces to evaluate the performance of DozyAP.

B. Power Consumption

Average Power: We first measured the power consumption of a mobile softAP with DozyAP and without DozyAP. Besides the five applications, we also measured two extreme cases: *idle* and *busy download*. In the idle case, we measured the power consumption of the softAP with one client associated but without any network traffic. In the busy download case, we measured the power consumption of downloading a 1-MB file from a Web server. The dark bars in Fig. 12 show the average power saving of DozyAP. Without explicit mention, the error bars depict the minimum and maximum values in all the experiments. We see that DozyAP can reduce the average power by 12.2%–32.8% for the five applications. In the idle case, it can save power by 36.5%. Even for the busy download case, the average power can be reduced by 3.3%. It is worth noting that the power saving percentage is calculated in *the total power consumption of the whole system*, including the power consumed by CPU and 3G. 3G consumes significant power when transmitting and receiving data. If we only consider the power consumption of Wi-Fi, the power saving percentage will be even higher in busy download and the five applications.

Total Energy: As DozyAP buffers packets and delays their transmission, it may lead to longer application time compared to the case without DozyAP. Thus, we also measured the total energy for the busy download case and the five applications. Total energy does not make sense for the idle case. The light bars in Fig. 12 show the results. We see that DozyAP does not increase the total energy. Instead, it can save the total energy by 12.2%–32.9% for the five applications, which is almost the same as the result of average power. As we show in Section V-C, DozyAP indeed introduces very little network latency that has negligible impact on the total energy. Even in the busy download case, DozyAP can save the total energy by 2.3%.

Wi-Fi Interface Sleep Time: As we point out in Section IV, with the current commercially available smartphones, forcing the softAP to go to sleep or wake up can be only achieved by turning off/on the Wi-Fi interface. That results in significant overhead (see Fig. 11). If we have more control on the power states of the Wi-Fi hardware (e.g., if we can directly modify the firmware or if we have a MadWifi [13] style driver that implements most MAC-layer functions in driver rather than in firmware), DozyAP should be able to save significantly more power. We measured how much time DozyAP can put the Wi-Fi interface of a mobile softAP to sleep. Fig. 13 shows the results. We see that the Wi-Fi interface of a softAP can stay in sleep mode for 47%–88% of the total time in the five applications. Even in the busy download case, we can turn the Wi-Fi interface to sleep for 11% of the total time. These results demonstrate the potential of DozyAP to significantly reduce the power consumption of Wi-Fi tethering. Given proper control over the Wi-Fi hardware, more energy is expected to be saved from sleeping.

We also evaluated the Wi-Fi interface sleep time with the real traces of the eight users in MoodSense [11]. To do it, we wrote a program to analyze the packet interarrival time of the traces and calculate the Wi-Fi interface sleep time as if these traces have happened in Wi-Fi tethering. To make the calculation reasonable, we ignored all the interpacket arrival intervals larger than 5 min. That is, for any intervals larger than 5 min, we treated it as if the user stopped using the phone and turned Wi-Fi tethering off. This treatment is conservative because a user may spend more than 5 min to read a long news article or Wi-Fi tethering might not be turned off even the user stopped using the phone for 5 min. Fig. 14 shows the calculated results. We see that DozyAP is able to allow the Wi-Fi interface of a softAP stay in sleep mode for 77%–95% of the time for the mixed multiapplication user traffic. The numbers in Fig. 14 are higher than the ones in Fig. 13. The reason is that the experiments in Fig. 13 focused on single application usage only. In practice users may use multiple applications one by one. Switching from one application to another leads to more network idle time.

Power Consumption of a Client: We also measured the power consumption of the Nexus One client in the idle case, busy download, and the five applications. Fig. 15 shows the results. We see that DozyAP can increase the power consumption of the client by less than 7.1% for these five applications. The reason is that the client needs to wake up to receive the sleep requests from the softAP and send back the sleep responses when the network is idle. Thus, the idle case introduces the highest overhead, but it is still only 8%. Compared to the large power saving of the softAP, this small overhead is acceptable.

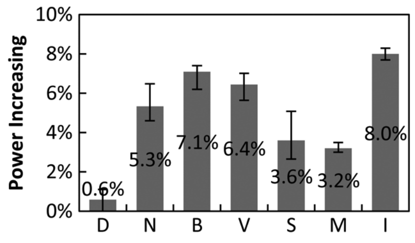


Fig. 15. Power increasing of the Nexus One client in idle, busy download, and the five applications.

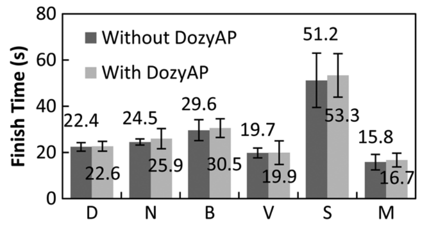


Fig. 16. Finish time of busy download and the five applications.

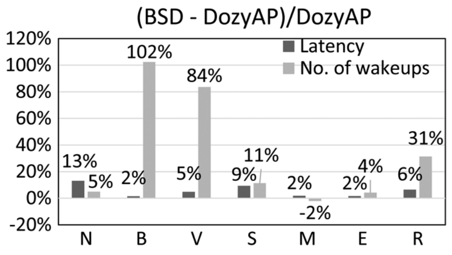


Fig. 17. BSD versus DozyAP in news reading (N), book reading (B), video streaming (V), search (S), map (M), e-mail (E), and RSS (R).

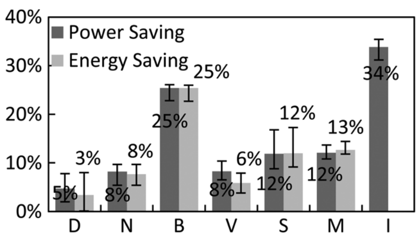


Fig. 18. Power saving and energy saving of softAP with two clients.

Multiple Applications on Single Client: In some cases, multiple applications may run on a single client simultaneously. We evaluated DozyAP in a typical scenario where a user is reading news in the foreground, meanwhile listening to online music in the background. To do it, we first started *Douban FM* (which is a popular app in China like *Last.fm*). Once the music began to load, we started the news reading program (the same as before) immediately. The average power saving and energy saving over 10 experiments is 14.5% and 14.2% respectively.

C. Latency

DozyAP incurs extra network latency because it delays packet transmissions when a softAP is in sleep mode. If the extra latency is user-perceivable, it may impair user experience. As all the five applications are about fetching remote Web content, users care about the page loading time, which is the period from the time when a user sends out a Web page request to the time when the Web page is fetched and rendered by the browser. The page loading time metric is widely used to evaluate the performance of browsers and Web servers. We evaluated the finish time of loading content, which is the sum of the page load time of all the Web page requests in an application. The WebView object used in our test program could tell when a Web page is loaded. In the experiments, we sent out all the Web page requests of an application one by one without any time interval and calculated the total finish time.

Fig. 16 shows the average result and the variance in busy download and the five applications. We see that DozyAP introduces very small extra network latency, ranging from 0.9% to 5.1%. Such small extra latency is hardly perceivable by users because of two reasons. First, as the 3G network has limited throughput and large RTT, it takes several hundred milliseconds to even seconds to load a Web page. Second, the time variance of the page loading time is pretty large, up to several seconds. That is, even without DozyAP, users already experience long page loading time with large variance. Therefore, the small latency increase of less than 5.1% is very hard to detect.

In addition, we compared our adaptive sleeping algorithm to BSD algorithm [7]. The reason why we choose BSD is that it is the-state-of-the-art algorithm that can adapt the sleep duration without the MAC-layer or lower-layer support. To be fair, we investigated BSD and our algorithm based on the same traffic traces collected from the user studies (as mentioned in Section II). The results are shown in Fig. 17. As we see, BSD algorithm may introduce extra network latency and more wakeups in most cases. This extra latency may delay the application finish time. The increased number of wakeups may cost more energy due to the wakeup overhead.

D. Multiple Clients

We first evaluated the performance of DozyAP with two clients associated: a smartphone and a tablet. Each client ran the same programs simultaneously. Fig. 18 shows the average energy saving and power saving in busy download, the five applications, and the idle case. As expected, the most power and energy savings are lower than the ones in the single-client scenario. However, the saving in the download case does not drop as much as other applications. That is because no matter if one or multiple clients were downloading, the cellular bandwidth was similarly saturated so that the chance for softAP to sleep is equivalent. Another finding is that the power saving for video streaming has a significant drop from about 28% to less than 10%. The reason is that two clients were competing in streaming video so that both of them needed more time to finish. Thus, the softAP had less opportunity to sleep.

We also conducted user studies to evaluate DozyAP with more clients. In the experiments, four clients with two phones, a tablet, and a laptop were tethered to a DozyAP-enabled smartphone. They were asked to access the Internet freely, such as reading news, checking and replying e-mails, listening to Internet radios, and searching interesting places on Google Maps. Since clients behave differently in each experiment, it is difficult for us to obtain the ground truth about the power consumption without DozyAP. Thus, we only measured the sleep time

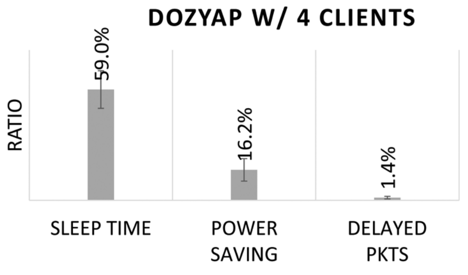


Fig. 19. Performance of DozyAP with four clients. The ratios of sleep time and delayed packets are measured, whereas the ratio of power saving is approximated based on the ratio of sleep time.

to approximate the power saving and count the buffered packets to show the incurred network latency. The results are shown in Fig. 19. We see that DozyAP allows the Wi-Fi to sleep for 59% of total application time and only causes about 1.4% delayed packets. From the experiments in Figs. 12 and 13, we observed that the *sleep-to-power-saving* translation ratio was around 18% –36%. Therefore, the approximated power saving for our four client tests is about 16.2% on average.

It is worth noting that the energy gain of DozyAP mainly depends on the traffic rather than the number of clients. Even if a single client is associated, it may generate continuously bursty traffic such as downloading. The power saving in that case is still less than the case of multiple clients, but with sparse traffic.

E. Comparison to Client-Independent Solution

To show the necessity of changing clients, we compared DozyAP to a straightforward client-independent solution, where the AP periodically wakes up after a sleep. If not receiving any data, the AP waits for a fixed time period and then goes back to sleep again. In this approach, the softAP wakes up and sleeps without notifying clients, so the packets transmitted by the clients may be lost. We use α to denote the ratio of the wakeup duration to the sleep duration and investigated the performance degradation with varying α.

We selected four applications including browsing Web sites (both 3G and regular Web sites), searching, downloading, and video streaming. The performance metrics are as follows.

- *Timeout ratio*: It depicts the percentage of the connection timeout reported by the client's Web browser. The connection timeout is caused by the loss of HTTP requests when the client sent those requests but the softAP in sleep mode did not receive them.
- *Latency increasing ratio*: With respect to page (or download) finish time, latency increasing ratio presents the increased latency normalized by the benchmark finish time in DozyAP. A large ratio means that more delays are introduced by the client-independent solution mentioned above.
- *Traffic increasing ratio*: Due to the retransmission of lost packets, the client may send or receive more packets. Traffic increasing ratio is used to describe the percentage of the number of increased packets to the number of total packets.
- *Power saving ratio and energy saving ratio*: The meanings of these metrics are the same as mentioned before. Note that these metrics are measured on the AP side, whereas the above three metrics are measured on the client side.

TABLE I
PERFORMANCE OF CLIENT-INDEPENDENT SOLUTION

	Timeout	Latency	Traffic	Power	Energy
Web(3G)	20%	46%	15%	20%	-11%
Web	40%	38%	7%	20%	-7%
Search	30%	74%	50%	29%	-26%
Download	40%	17%	4%	17%	-3%
Video	20%	12%	1%	16%	-3%

Table I shows the averaged results over 10 tests when $\alpha = 0.4$. Other values of α yielded worse results, so we do not present them here. As we see, without modifying the clients, the timeout ratio is between 20%–40%, thereby impairing the user experience of Internet access. Both latency and traffic are heavily increased, where video streaming has the lowest increasing ratio and searching has the highest increasing ratio. The power saving by the client-independent solution is competitive to softAP because the Wi-Fi interface is also turned off periodically. However, due to longer application finish time, the client-independent solution actually spends more energy than existing Wi-Fi tethering schemes. Therefore, it demands modifying the clients to achieve better performance.

VI. Discussion and Future Work

DozyAP requires patching the OS of smartphones working in Wi-Fi tethering and installing a loadable module on a client, which may be a hurdle for device vendors to overcome in practice. Despite this, we believe this problem is not very difficult to tackle, for instance, through Over-the-Air (OTA) upgrade. It also may be difficult to upgrade the software of dumb Wi-Fi client devices, e.g., music players and e-readers. However, to access the Internet, most people use "smart" devices including smartphones, tablets, and laptops. All these devices are programmable and upgradable. Although our current implementation is based on a Linux-style OS kernel including iOS, for Windows-based devices, a similar approach can be used via la oadable Network Driver Interface Specification (NDIS) [14] driver.

DozyAP takes advantage of the speed discrepancy between cellular and Wi-Fi. One may argue that such an advantage will not exist when 4G is deployed. However, the speed of Wi-Fi increases quickly as well. With 11n and 11ac, there is still a big gap between cellular and Wi-Fi. In addition, our solution benefits not only from such a speed discrepancy, but also from the long content consuming time of users.

Our implementation uses fixed parameter values derived from the measurement results, which can be improved. For example, one may use a dynamic approach to tune the parameters to better adapt to the network conditions. Even though we use fixed values, we take a conservative way, e.g., the sleep time starts from a small value of 100 ms. As shown in Fig. 8, the tuning procedure of parameter *init* is also conservative.

More power can be saved through transmission power adaptation. The built-in Wi-Fi tethering on existing smartphones always uses the highest transmission power. It wastes energy because a softAP is often close to its clients in Wi-Fi tethering. We plan to design a scheme to automatically adjust the transmission power based on the network conditions (e.g., RSSI and packet loss). We also plan to further take advantage of the bandwidth discrepancy between 3G and Wi-Fi to create more opportunities for a softAP to sleep. The basic idea is shaping the traffic between 3G and Wi-Fi. For downlink traffic, the softAP can buffer the packets received from 3G and send them to the client over Wi-Fi in batch. For uplink traffic, if the 3G connection is congested, the softAP can ask the client to stop sending more data. Thus, the Wi-Fi interface of both the softAP and the client can sleep longer.

DozyAP could be implemented in the MAC layer for further improvement if the Wi-Fi firmware is open on smartphones. The current implementation of DozyAP incurs the performance penalty from three aspects. First, explicit transmission of sleep request and response packets consumes extra power for both AP and clients. Second, the overhead of switching on and off Wi-Fi interface is considerable. Third, additional power consumption is imposed by the CPU computation since DozyAP has to involve the CPU to generate packets and run algorithms that are supposed to run on Wi-Fi chipsets.

VII. Related Work

Wi-Fi Power Saving: There has been a lot of research effort devoted to power saving in Wi-Fi [7], [8], [15]–[22], focusing on improving the existing PSM in general or targeting specific applications or usage scenarios. To name some recent work, Catnap [17] exploits the bandwidth discrepancy between Wi-Fi and broadband to save energy for mobile devices. NAPman [22] employs an energy-aware scheduling algorithm to reduce energy consumption by eliminating unnecessary retransmissions. SleepWell [8] coordinates the activity circles of multiple APs to allow client devices to sleep longer. All these solutions are for Wi-Fi clients only. DozyAP is complementary, focusing on the power efficiency of APs. Putting an AP to sleep is more challenging than putting a client to sleep because client devices expect that their AP is always on. To avoid packet loss, a softAP in DozyAP must coordinate its sleep schedule with its clients, which is different from existing work.

There is little work on power saving of APs. In [23] and [24], the authors propose to extend the IEEE 802.11 standard to support power-saving access points for multihop solar/battery-powered applications. Without building any real systems, they focus on protocol analysis and simulation, assuming Network Allocation Vector (NAV) can be used. Our work focuses on system design and implementation. We build real systems on commercial smartphones and do evaluation with real experiments. In addition, the NAV-based approach cannot work on existing smartphones because NAV is only visible in firmware. Cool-Tether [1] considers an alternative way to address the mobile hotspot problem that involves reversing the role of the phone and the client. However, it significantly increases the power consumption of the client and does not support multiple clients. In [25], the authors design algorithms to save power for APs with Wi-Fi Direct. Since Wi-Fi Direct is a separate mode on devices, it cannot be used for tethering until now. Furthermore, we measured the power consumption of current Wi-Fi Direct-enabled devices, and it is slightly higher than tethering.

Traffic-Driven Design: Adapting to traffic load for better sleeping is not a new idea [15], [21]. Traffic patterns in different applications and scenarios have also been studied in some papers, and the similar observations are identified (e.g., the large portion of network idle time) [7], [19]. DozyAP builds on top of the basic techniques and applies them to the Wi-Fi tethering scenario. Furthermore, DozyAP can be improved by leveraging existing literature, e.g., by traffic shaping [17], [20] and sleeping in short intervals [19].

Sleep Scheduling: Sleep/wake scheduling has been extensively studied in Bluetooth domain, e.g., [26] and [27], and sensor network domain, e.g., [28] and [29]. However, those approaches usually focus on MAC-layer design, resulting in a new MAC protocol, and often require time synchronization. DozyAP employs a simple application-level protocol to coordinate the sleep schedule of a softAP with its client, without requiring time synchronization or any modifications on existing IEEE 802.11 protocol. Thus, DozyAP is easy to deploy on existing smartphones.

Dedicated Wi-Fi Tethering Devices: MiFi [30] is a dedicated mobile Wi-Fi hotspot device. However, such a device also stays in a high power state even without any ongoing traffic. We measured a Huawei E5830 MiFi device and found the average power consumption was as high as 420 mW in idle case. We believe MiFi devices can benefit from DozyAP design if they are programmable.

VIII. Conclusion

In this paper, we propose the DozyAP system to improve the power efficiency of Wi-Fi tethering. DozyAP employs a lightweight yet reliable sleep request-response protocol for a mobile softAP to coordinate its sleep schedule with its clients without requiring tight time synchronization. Based on our findings on the traffic patterns of typical applications used in Wi-Fi tethering, we design a two-stage adaptive sleep algorithm to allow a mobile softAP to automatically adapt to the ongoing traffic load for the best power saving. We have implemented DozyAP system on commercial smartphones. Experimental results demonstrate that DozyAP is able to significantly reduce the power consumption of Wi-Fi tethering without impairing the user experience.

Acknowledgment

The authors thank the anonymous reviewers for their valuable comments and the Associate Editor, Prof. A. Capone, for his guidance throughout the review process.

References

[1] A. Sharma, V. Navda, R. Ramjee, V. N. Padmanabhan, and E. M. Belding, “Cool-tether: Energy efficient on-the-fly wifi hot-spots using mobile phones,” in *Proc. CoNEXT*, 2009, pp. 109–120.

[2] H. Wirtz, R. Backhaus, R. Hummen, and K. Wehrle, “Establishing mobile ad-hoc networks in 802.11 infrastructure mode,” presented at the WiNTECH Demo Session, 2011.

[3] Monsoon Solutions, Inc., Bellevue, WA, USA, “Monsoon power monitor,” [Online]. Available: http://www.msoon.com/LabEquipment/PowerMonitor/

[4] Intelliborn, Colorado Springs, CO, USA, “Mywi and Mywi Ondemand,” [Online]. Available: http://intelliborn.com/mywi.html

[5] J. Huang, Q. Xu, B. Tiwana, Z. M. Mao, M. Zhang, and P. Bahl, “Anatomizing application performance differences on smartphones,” in *Proc. MobiSys*, 2010, pp. 165–178.

[6] *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*, IEEE Std 802.11, 2007.

[7] R. Krashinsky and H. Balakrishnan, "Minimizing energy for wireless web access with bounded slowdown," in *Proc. MobiCom*, 2002, pp. 119–130.

[8] J. Manweiler and R. R. Choudhury, "Avoiding the rush hours: Wifi energy management via traffic isolation," in *Proc. MobiSys*, 2011, pp. 253–266.

[9] Broadcom, Irvine, CA, ISA, "BCM4329 product brief," 2013 [Online]. Available: http://www.broadcom.com

[10] "Android Webkit package and Webview class," 2013 [Online]. Available: http://developer.android.com/reference/android/webkit/package-summary.html

[11] R. LiKamWa, Y. Liu, N. D. Lane, and L. Zhong, "MoodSense: Can your smartphone infer your mood?," 2011 [Online]. Available: http://research.microsoft.com/en-us/projects/moodsense/

[12] "Tcpdump," [Online]. Available: http://www.tcpdump.org/

[13] "The Madwifi Project," 2013 [Online]. Available: http://madwifi-project.org/

[14] Microsoft, Redmond, WA, USA, "Network Driver Interface Specification (NDIS)," [Online]. Available: http://msdn.microsoft.com/en-us/library/ff559102.aspx

[15] M. Anand, E. Nightingale, and J. Flinn, "Self-tuning wireless network power management," in *Proc. Mobicom*, 2003, pp. 176–189.

[16] D. Bertozzi, L. Benini, and B. Ricco, "Power aware network interface management for streaming multimedia," in *Proc. IEEE WCNC*, 2002, pp. 926–930.

[17] F. Dogar, P. Steenkiste, and K. Papagiannaki, "Catnap: Exploit high bandwidth wireless interfaces to save energy for mobile devices," in *Proc. MobiSys*, 2010, pp. 107–122.

[18] Y. He and R. Yuan, "A novel scheduled power saving mechanism for 802.11 wireless LANs," *IEEE Trans. Mobile Comput.*, vol. 8, no. 10, pp. 1368–1383, Oct. 2009.

[19] J. Liu and L. Zhong, "Micro power management of active 802.11 interfaces," in *Proc. MobiSys*, 2008, pp. 146–159.

[20] C. Poellabauer and K. Schwan, "Energy-aware traffic shaping for wireless real-time applications," in *Proc. RTAS*, 2004, p. 48.

[21] D. Qiao and K. Shin, "Smart power-saving mode for IEEE 802.11 wireless LANs," in *Proc. IEEE INFOCOM*, 2005, pp. 1573–1583.

[22] E. Rozner, V. Navda, R. Ramjee, and S. Rayanchu, "NAPman: Network-assisted power management for wifi devices," in *Proc. MobiSys*, 2010, pp. 91–106.

[23] T. D. T. Y. Li and D. Zhao, "Access point power saving in solar/battery powered IEEE 802.11 ESS mesh networks," in *Proc. IEEE QShine*, 2005, pp. 45–49.

[24] F. Zhang, T. D. Todd, D. Zhao, and V. Kezys, "Power saving access points for IEEE 802.11 wireless network infrastructure," *IEEE Trans. Mobile Comput.*, vol. 5, no. 2, pp. 144–156, Feb. 2006.

[25] D. Camps-Mur, X. Pérez-Costa, and S. Sallent-Ribes, "Designing energy efficient access points with wi-fi direct," *Comput. Netw.*, vol. 55, no. 13, pp. 2838–2855, 2011.

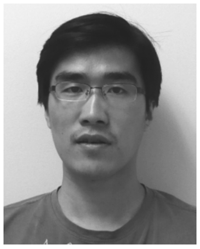
[26] S. Garg, M. Kalia, and R. Shorey, "MAC scheduling policies for power optimization in Bluetooth: A master driven TDD wireless system," in *Proc. IEEE VTC*, 2000, pp. 196–200.

[27] T.-Y. Lin and Y.-C. Tseng, "An adaptive sniff scheduling scheme for power saving in Bluetooth," *IEEE Wireless Commun.*, vol. 9, no. 6, pp. 92–103, Dec. 2002.

[28] T. van Dam and K. Langendoen, "An adaptive energy-efficient MAC protocol for wireless sensor networks," in *Proc. SenSys*, 2003, pp. 171–180.

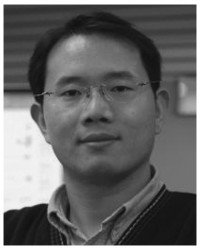
[29] Y. Wu, S. Fahmy, and N. B. Shroff, "Optimal sleep/wake scheduling for time-synchronized sensor networks with QoS guarantees," *IEEE/ACM Trans. Netw.*, vol. 17, no. 5, pp. 1508–1521, Oct. 2009.

[30] "MiFi," 2013 [Online]. Available: http://en.wikipedia.org/wiki/MiFi



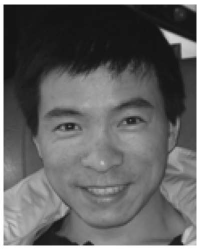
Hao Han received the B.S. degree in computer science from Nanjing University, Nanjing, China, and is currently pursuing the Ph.D. degree in computer science at the College of William and Mary, Williamsburg, VA, USA.

His research interests include RFID systems, wireless networks, and mobile computing.



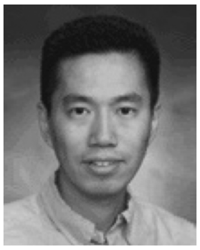
Yunxin Liu (M'05) received the B.S. degree from the University of Science and Technology of China, Hefei, China, in 1998, the M.S. degree from Tsinghua University, Beijing, China, in 2001, and the Ph.D. degree in computer science from Shanghai Jiao Tong University, Shanghai, China, in 2011.

In 2001, he joined Microsoft Research Asia, Beijing, China, where he is currently a Lead Researcher. His research interests are mobile systems and networking.



Guobin Shen (S'99–M'02–SM'06) received the Ph.D. degree in electrical engineering from the Hong Kong University of Science and Technology, Hong Kong, in 2001.

He is now a Lead Researcher with the Wireless and Networking Group, Microsoft Research Asia, Beijing, China. His current research focuses on mobile sensing and actuation systems, mobile multimedia, and low-latency mobile-cloud systems, while he has worked on topics in peer-to-peer networking and systems, wireless sensor networks, video compression and streaming, and GPU acceleration of video signal processing.



Yongguang Zhang (M'94–SM'11) received the Ph.D. degree in computer science from Purdue University, West Lafayette, IN, USA, in 1994.

He is a Research Manager with Microsoft Research Asia, Beijing, China, in the areas of mobile systems and networking. Before joining Microsoft, he was a Senior Research Scientist with HRL Labs, Malibu, CA, USA. He has published over 50 technical papers and one book. He has made technical contributions in internetworking techniques, system developments, and security mechanisms for satellite networks, ad hoc networks, and 3G wireless systems.

Dr. Zhang was a Guest Editor in *Mobile Networks and Applications*, has organized and chaired/co-chaired several international conferences and workshops, and was a founding Co-Chair of the IETF UDLR Working Group.

Qun Li (M'05–SM'12) received the Ph.D. degree in computer science from Dartmouth College, Hanover, NH, USA, in 2004.

He is an Associate Professor with the Department of Computer Science, College of William and Mary, Williamsburg, VA, USA. His research focuses on wireless networks and embedded systems, including pervasive computing, cognitive radio, wireless LANs, mobile ad hoc networks, sensor networks, and RFID systems.

Dr. Li received the US National Science Foundation (NSF) Career Award in 2008.

Chiu C. Tan (M'07) received the Ph.D. degree in computer science from the College of William and Mary, Williamsburg, VA, USA, in 2010.

He is an Assistant Professor with the Computer and Information Sciences Department, Temple University, Philadelphia, PA, USA. His research interests include wireless security (802.11, vehicular, RFID), cloud computing security, and security for mobile health (mHealth) systems.