DozyAP: Power-Efficient Wi-Fi Tethering

Hao Han[1,2], Yunxin Liu[1], Guobin Shen[1], Yongguang Zhang[1], Qun Li[2]
[1]Microsoft Research Asia, Beijing, China
[2]College of William and Mary, Williamsburg, VA, USA

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.

Categories and Subject Descriptors

C.2.1 [Computer-communication Networks]: Network Architecture and Design — Wireless communication

General Terms

Algorithm, Design, Experimentation, Measurement

Keywords

802.11, Energy Efficiency, SoftAP, Wi-Fi Tethering

1. 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 Figure 1, a Wi-Fi tethering mobile phone acts as a mobile software access point (SoftAP). Other devices 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.

Figure 1: Wi-Fi tethering.

Wi-Fi tethering is highly desired. For example, even before the Android platform provided built-in support on Wi-Fi tethering, there were already some third-party Wi-Fi tethering tools on Android Market [8]. Two of them, called "Barnacle Wi-Fi Tether" and "Wireless Tether for Root Users", are very popular, each with more than one million installs. There are two main reasons why Wi-Fi tethering is desirable. First, cellular data networks provide ubiquitous Internet access but the coverage of Wi-Fi networks is spotty. 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. Hence, it is desirable to share a data plan among multiple devices, e.g., sharing the 3G connection of an iPhone with a Wi-Fi only iPad. In response to this common user desire, Wi-Fi tethering is now widely supported, as a built-in feature on most smartphone platforms, including iPhones (iOS 4.3+), Android phones (Android 2.2+) and Windows phones (Windows Phone 7.5).

However, existing Wi-Fi tethering schemes significantly increase the power consumption of smartphones. When operating in the SoftAP mode, the Wi-Fi interface of a smartphone is always put in the 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 2.1). To save power, Windows Phone automatically turns off Wi-Fi tethering if the Wi-Fi network is inactive for a time threshold of several minutes. However, this simple scheme has two drawbacks. First, the Wi-Fi interface still operates in the high power state for all 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

MobiSys'12, June 25–29, 2012, Low Wood Bay, Lake District, UK.

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 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 a Power Saving Mode (PSM) [10] that allows the Wi-Fi interface of a device to stay in a lower-power state to save power. However, PSM is designed for client devices only. An AP should never enter PSM according to the IEEE 802.11 standard. It is not an issue for APs in traditional scenarios because they are externally powered. However, such old design wisdom does not work for the battery-powered SoftAPs in Wi-Fi tethering.

The IEEE 802.11 standard also defines a power saving mechanism for stations operating in ad hoc mode. In ad hoc mode all the stations equally share the responsibility for beacon generation. One station may sleep when another station generates beacons. However, in Wi-Fi tethering, a SoftAP is the single gateway to the Internet and can hardly sleep. Ad hoc mode is much less used than the infrastructure mode (i.e., AP mode) in practice. Interestingly, despite that the Wi-Fi hardware does support ad hoc mode, the OS on many mobile devices, including Windows phones, Android phones and iPhones, hides it [30], preventing a device from connecting to an ad hoc network. Thus, in this paper we focus on Wi-Fi tethering in the infrastructure mode.

We propose and 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 mode to save power when possible. We measure the traffic pattern of various online applications used in Wi-Fi tethering. We find that the Wi-Fi network stays in the idle state for a large portion of the total application time (more details in Section 2.2), which means there are many opportunities to reduce the 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.

Putting a SoftAP to 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 retrials 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 experiences 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 learn and agree on a valid sleep schedule of the SoftAP. 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 go 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%.

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

- We study the characteristics of existing Wi-Fi tethering implementations and present our findings. We show that current Wi-Fi tethering is power hungry and wastes 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 for 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 commercial 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 2 we report our findings on existing Wi-Fi tethering, focusing on the power consumption and the traffic patterns of various applications. Based on the findings, in Section 3 we design DozyAP to schedule a mobile SoftAP for sleep and present the design details. We describe our implementation in Section 4 and evaluate it in Section 5. We discuss limitations of DozyAP and future work in Section 6, survey the related work in Section 7 and conclude in Section 8.

2. 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 have focused on two characteristics: the power consumption of existing Wi-Fi tethering implementations and the traffic pattern of various online applications used in Wi-Fi tethering. We also provide some background on Wi-Fi power management to set up the context of our DozyAP design.

2.1 Power Consumption

We first measure the power consumption of existing commercial smartphones regarding the Wi-Fi tethering. We impose *no traffic* but simply turn 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), a 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 [4]. However, it is not possible to use the Monsoon Power Monitor to measure the power consumption of the iPhone 4 without damaging the phone. Instead, we used MyWi 5.0 [5], 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 was pretty low, because the Wi-Fi and 3G interfaces were in sleep for most of the time. The average power consumption was only 20mW for the Nexus One and 30mW for the HTC HD7, respectively. For the iPhone4, MyWi 5.0 read a draining current of 6mA, equivalently, a power consumption of 22mW.

With Wi-Fi tethering enabled, the power consumption of the smartphones increased significantly. Figure 2 shows the results for 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 270mW for the Nexus One and 302mW for the HTC HD7. These results indicate that Wi-Fi tethering leads to severe drop of the battery lifetime: from 70 hours to 5.2 hours for the Nexus One and from 43 hours to 4.3 hours for the HTC HD7, given their respective battery capacity of 1400mAh and 1300mAh. For the iPhone 4, MyWi 5.0 read a draining current of 90mA, equivalently a power consumption of 333mW. While this software approach may not be as accurate as using 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.

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 next subsection we will show that such *idle* cases are indeed frequent in various typical Internet access scenarios.

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.

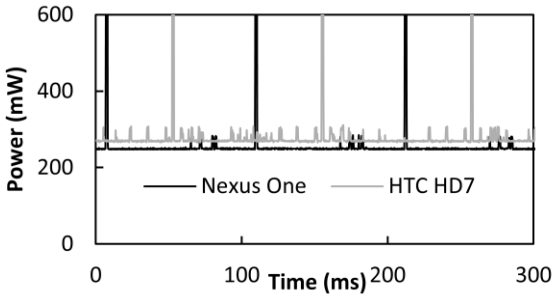

Figure 2: Measured power consumption of Wi-Fi tethering on Nexus One and HTC HD7 in idle case.

2.2 Traffic Patterns

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, and connected a Wi-Fi client to the mobile SoftAP. On the client, we launched various applications that access the Internet and used those applications normally. In the meantime, we used a Lenovo T61 laptop running Linux 2.6.32 as a Wi-Fi sniffer to capture all the frames exchanged between the client and the SoftAP. We studied two different clients: a Nexus One smartphone and a Wi-Fi version iPad 2. The following seven applications were measured.

(1) News reading. We went to http://news.baidu.com, and read five news articles one by one. **(2) Online book reading**. We went to http://www.zangdimima.org and read three chapters of a popular book. **(3) Video streaming**. We went to http://www.youku.com (a popular video website in China like YouTube in US) and watched a 76-second long video clip. **(4) Search**. We used Google Search to search for "tethering" and read the first three results one by one. **(5) Map**. We used Bing Map to search and browse three locations in Beijing: Tiananmen Square, Peking University and Microsoft R&D Asia. **(6) Email**. We used Gmail to read three new emails and replied to one of them. **(7) RSS Reader**. We used Google reader to download and read five pieces of news.

Note that some websites 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 webpages than the normal version. Similarly, Youku streams low bitrate video clips to the Nexus One smartphone but high bitrate ones of the same videos to the iPad 2. Consequently, the same application may behave differently on different devices.

For each application, we study the traffic patterns by analyzing the packet inter-arrival time of all the captured packets. Figure 3 shows the results of the Nexus One. Due to the space limitation, we omit the curves of the iPad 2 which has similar results. We first study the distribution of packet in-

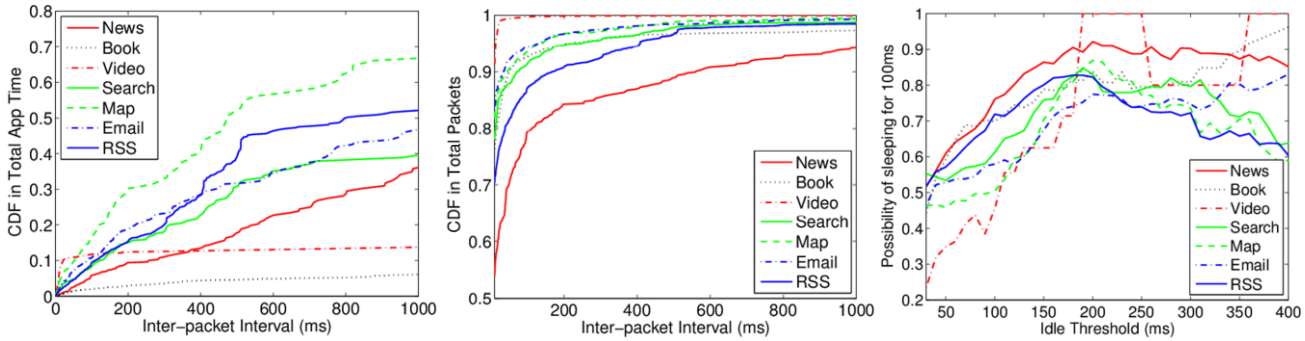

Figure 3. Traffic patterns. From left to right: CDF of packet inter-arrival interval in total application time; CDF of packet inter-arrival interval in total packets; Probability of sleeping for 100ms after an idle threshold.

ter-arrival intervals in the total application time which is the period from the first packet to the last one. The left figure in Figure 3 shows the Cumulative Distribution Function (CDF) for all the applications, where the y-axis depicts the percentage of packets with inter-packet 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 one second. We can see that the intervals under 200ms 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 inter-arrival intervals larger than 200ms, then we can say that the Wi-Fi network was idle for 70%-90% of the total application time. This 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 email case, replying to emails 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 3GTest [18], 3G typically offers 500Kbps – 1Mbps downlink throughput for US carriers but the Wi-Fi offers much higher data rates (54Mbps for 802.11a/g and 300Mbps for 802.11n). Furthermore, 3G has much higher RTTs, ranging from 200ms to 500ms [18], 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 bitrate video clip was continuously played back, the packet inter-arrival intervals larger than 200ms 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 a 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 Figure 3 shows the CDF of packet inter-arrival interval in total packets, where the y-axis depicts the percentage of the packets whose inter-packet 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 inter-packet intervals under 150ms 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 150ms to decide whether to put the SoftAP to sleep or not, most of the packets will not be affected. The right figure in Figure 3 further shows the probability that the SoftAP can successfully sleep for extra 100ms after waiting for different idle

thresholds. Again, if we use a threshold of 150ms, the probability is higher than 60% for all the applications.

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

2.3 Background: Wi-Fi Power Saving

The IEEE 802.11 standard defines PSM to save power of Wi-Fi client devices [10]. In PSM, the Wi-Fi interface of a client stays in a lower-power state to save power and cannot receive or transmit any data. If an AP has some packets for a PSM client, the AP buffers the packet and sets the Traffic Indication Map (TIM) in its periodic beacons which are broadcasted typically every 100ms. A PSM client periodically (e.g., every three beacon intervals) wakes up to listen to the beacons of the AP. If the client detects a TIM for itself, it sends a separate PS-Poll frame to receive each buffered packet. Otherwise it goes to sleep immediately. When the AP sends a buffered packet to the client, a MORE bit in 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 messages.

On the Nexus One, the above static PSM scheme is called "PM_MAX". PM_MAX allows a client to sleep as much as possible by allowing the client to sleep immediately if the AP does not have any packet for it. However, it leads to long network latency and hence low network efficiency because a separate PS-Poll message is required for every packet. This makes it less suitable for interactive applications. 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 200ms. 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. Many other Wi-Fi devices today also implement a similar scheme known as adaptive PSM [19]. 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.

3. DOZYAP DESIGN

Guided by the findings in Section 2, 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. Below we present the detailed design 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.

3.1 Scheduling a SoftAP for 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. Figure 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 t_3, the SoftAP wakes up and continue to communicate with the client.

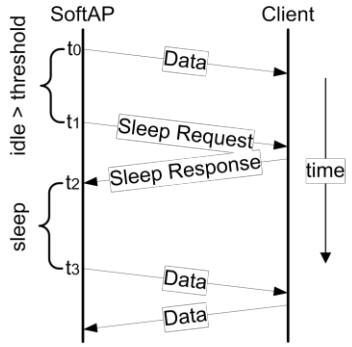

Figure 4: Interactions of a SoftAP and a client using the sleep request-response protocol.

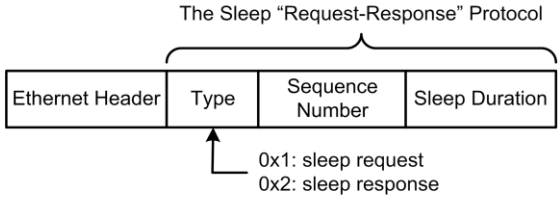

Figure 5: Packet format of the sleep protocol.

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. Figure 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: "0x1" means *sleep request* and "0x2" 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 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. Figure 6 shows the state machine of a SoftAP and a client. A SoftAP has three states: *Normal*, *Pre-sleep* and *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 pre-defined 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 enters 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 not send the received data to the client over Wi-Fi. Instead, it buffers them during the whole period of *Sleep* state. When the sleep timeout expires, if the SoftAP has any buffered data, it 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 and PM_FAST. If it 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" packet *before* we send out the sleep response. The "Null-Data" packet tells the SoftAP that the client will sleep immediately. Doing this way, the client can go to sleep as quickly as possible (i.e., immediately after the sleep response packet) to save power and the SoftAP knows that the client is sleeping. Otherwise, if the client uses PM_FAST scheme, after sending out the sleep response, it stays in active until it sends out a "Null-Data" packet to tell the SoftAP that it will go to sleep. However, as the SoftAP has already entered the *Sleep* state after receiving the sleep response, it cannot receive the "Null-Data" packet following the sleep response. As a result, the client cannot receive an ACK from the SoftAP for the "Null-Data" packet and cannot go to sleep. Note that PM_MAX is the

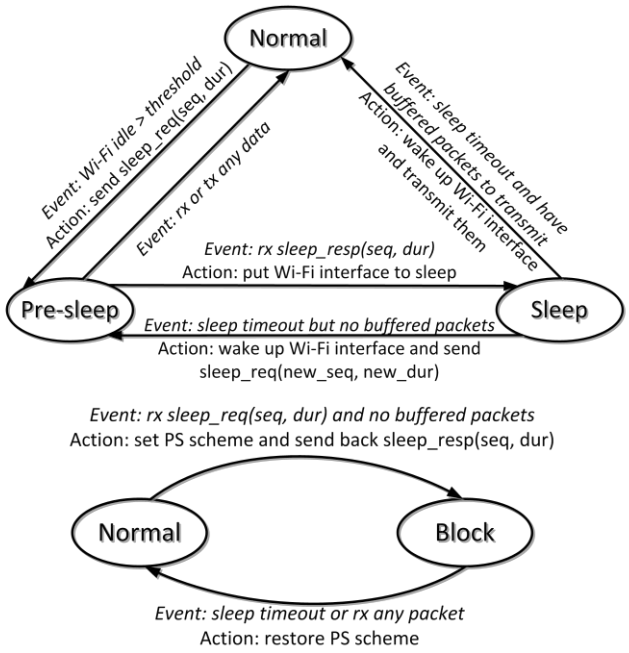

Figure 6: State machine for a SoftAP (top) and a client (bottom).

default scheme defined in PSM of IEEE 802.11 standard and is supported by all the client devices. Thus, this design does not depend on any specific devices.

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 are active and can communicate as normal (receiving or transmitting any data packet will cause the SoftAP moves to *Normal* state from *Pre-sleep* state). Otherwise, the SoftAP will send out a new sleep request, and both the SoftAP and the client can enter the *Sleep* or *Block* state again.

3.2 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. While very fine-grained hardware timestamps (e.g., at microsecond granularity) already 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 increas-

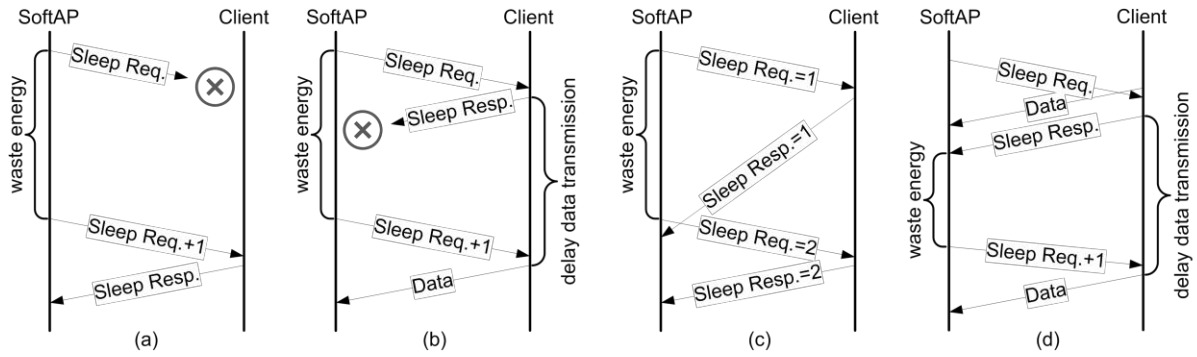

Figure 7: Abnormal cases: (a) a sleep request is lost; (b) a sleep response is lost; (c) a sleep response is delayed; (d) a data packet is delayed.

es 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. Below we analyze several possible abnormal cases and their consequences, as illustrated in Figure 7.

Packet loss. First, sleep request or response packets may be lost during their transmission. 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. Or the SoftAP will send out a new sleep request after the network remains idle for a period longer than the pre-defined idle threshold, as shown in Figure 7a. 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 Figure 7b.

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, Figure 7c 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. Figure 7d shows a more complex case. The client has already passed a packet

```
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;
```

Figure 8: The two-stage adaptive sleep algorithm.

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.

3.3 Adaptive Sleeping

We design an adaptive sleep scheduling algorithm for a SoftAP 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 fetching phase and the content consuming phase) of interactive applications, respectively.

Sleep algorithm. Figure 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 according to the probing history. 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 [25], 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* equals 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 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 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 will go to LONG_SLEEP stage. In LONG_SLEEP stage, the SoftAP simply sleeps for a time period of *long* periodically until it has to quit from sleep to communicate with the client.

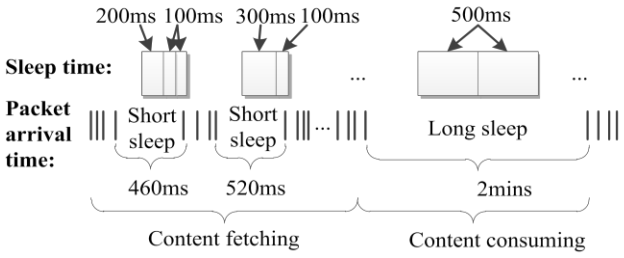

Figure 9: Short sleep and long sleep example.

Example. Figure 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 and the time taken to get permission from clients after each wake-up are omitted for sake of easier reading. Assume current value of *init* is 200ms, the value of *step* is 100ms and the value of *thresh* is 150ms. For the 460ms inter-packet interval in Figure 9, the SoftAP starts to sleep after 150ms. The SoftAP will first sleep for 200ms, followed by two 100ms sleep slots. Suppose then the value of *init* is qualified to increase to 300ms, for the 520ms inter-packet interval, the SoftAP will first sleep for 300ms followed by one more 100ms 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* (500ms).

The above algorithm is specially designed for the traffic patterns of typical applications in Wi-Fi tethering as shown in Section 2.2. 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 a RTT of the 3G connection (typically hundreds of milliseconds [18]). The purpose of *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 4 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.

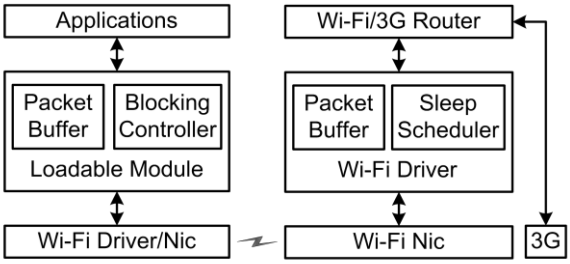

Figure 10: Implementation architecture of the client part (left) and the SoftAP part (right).

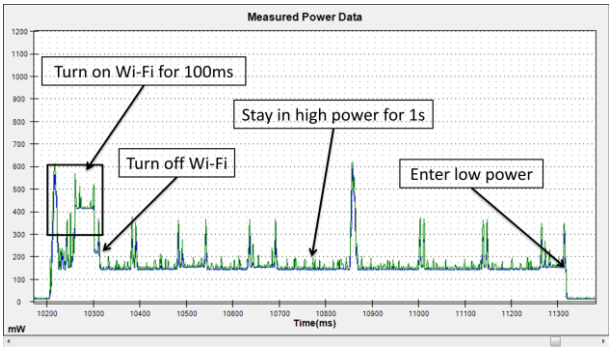

Figure 11: Power consumption of turning the Wi-Fi interface on/off in Wi-Fi tethering.

3.4 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. It is possible that some clients replied to a sleep request but other clients did not so that the SoftAP has to stay awake. However, this is not wrong 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 multi-client 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 several reasons. First, broadcast packets are less reliable because they are transmitted without link layer retransmissions. In particular, the clients in PSM likely cannot receive them. Second, broadcast packets are transmitted at lowest data rate and thus take more air time than normal packets. Third, the sleep responses are still sent in unicast packets. Last but not the least, Wi-Fi tethering typically happens with very few simultaneous clients, often only one or two clients. Therefore, the improvement of using broadcast is expected to be very small in multi-client case, and for single client case, it is worse than using unicast.

Another minor issue with the multi-client 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.

4. IMPLEMENTATION

We have implemented a DozyAP prototype on Nexus One smartphones running Android 2.3.6 (the latest official version for Nexus One), with a Wi-Fi chipset of Broadcom BCM4329 802.11 a/b/g/n [2].

The overall architecture consists of two parts: the SoftAP part and the client part, as shown in Figure 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 0xffffff (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 a *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 to 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 of the Android One smartphone keeps the Wi-Fi hardware always on and in a high power state. There is no interface available to change the power states. All that we can do in the driver is to turn on/off the Wi-Fi interface. Consequently, in our implementation, we simply turn off and on the Wi-Fi interface when the SoftAP decides to sleep and wakeup, respectively. 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 some extra energy to turning on/off the Wi-Fi interface. We measured such energy overhead on a Nexus One

smartphone and Figure 11 shows the measurement results. Initially, the Wi-Fi interface was off. Then we turned the Wi-Fi interface on for 100ms and turned it off again. We can observe two artifacts: first, when the Wi-Fi interface is turned on, the system jumps to a high power state of 600mW and stays in an average power state of 400mW which is higher than the normal power consumption of Wi-Fi tethering in idle case (270mW as shown in Figure 2). Second, when the off command is issued, the power consumption reduced immediately, but remains at a power level as high as 150mW for about one second before entering a very low power state of 10mW. This finding is similar to what the authors reported in [25]. 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: the SoftAP never enters the "deep sleep" state because we have limited the maximum sleep time to 500ms. 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 enter the "deep sleep" state more quickly, our approach can save much more power.

Parameter values. We determine the parameter values in the adaptive sleep algorithm (refer to Figure 8) based on the findings in Section 2 and the measured power data. We set *thresh* to 150ms for triggering a SoftAP to enter sleep mode. With this setting, the SoftAP can handle over 80% of the total packets in its short awake time and over 60% cases it can successfully sleep for 100ms if the SoftAP decides to. Due to the energy overhead of turning on the Wi-Fi interface as shown in Figure 11, there is no obvious benefit if the SoftAP sleeps less than 100ms. Thus, we set *min* to 100ms. Considering the RTT of 3G and to limit the maximum extra latency, we set *max* to 500ms. The value of *init* varies between 100ms and 500ms. We set *thresh*_l to 3s for switching to the long sleep stage where the SoftAP periodically sleeps for 500ms (i.e., *long* equals to 500ms).

5. EVALUATION

We evaluate the performance of DozyAP by answering the following questions. 1) How much power can DozyAP save for a mobile soft AP 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 multi-client case?

5.1 Experiment Setup

Hardware devices. We use one 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 use a Monsoon Power Monitor [4] to measure the power consumption. We repeat every experiment for five times and present the average results.

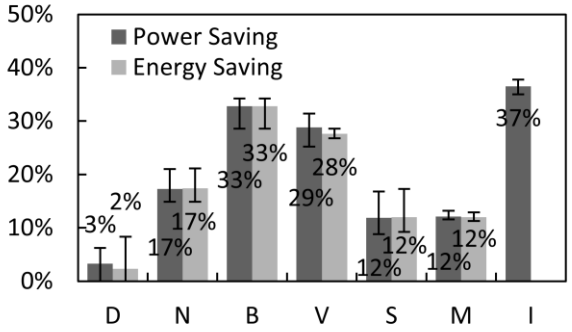

Figure 12: Power saving and energy saving of the SoftAP in idle (I), busy download (D) and the five applications of news reading (N), book reading (B), video streaming (V), search (S) and map (M).

Applications and methodology. We use five of the applications described in Section 2, including news reading, book reading, video streaming, search, and map. To make the experiments repeatable, except video streaming, we analyze the captured trace of the applications to find out all the HTTP requests contained in the traces. Then we write 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 [1]. Thus, we can easily repeat every experiment. For video streaming, we manually play the same video clip.

Traces. To evaluate DozyAP with more diverse and realistic traffic patterns, we asked the authors of MoodSense [21] 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 [7]. We select the traces of the top eight most active 3G users. For each of them, we further select the trace of the day when the user generated the largest 3G traffic volume. We use the eight-day traces to evaluate the performance of DozyAP.

5.2 Power Consumption

Average power. We first measure the power consumption of a mobile SoftAP with DozyAP and without DozyAP. Besides the five applications, we also measure two extreme cases: *idle* case and busy *download* case. In the idle case, we measure the power consumption of the SoftAP with one client associated but without any network traffic. In the busy download case, we measure the power consumption of downloading a 1MB file from a Web server. The dark bars in Figure 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 can see that DozyAP can reduce the average power by 12.2% to 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 con-

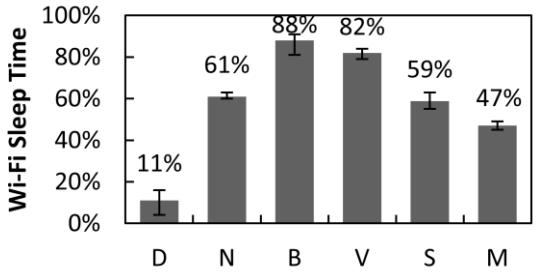

Figure 13: Wi-Fi interface sleep time of the SoftAP in busy download and the five applications.

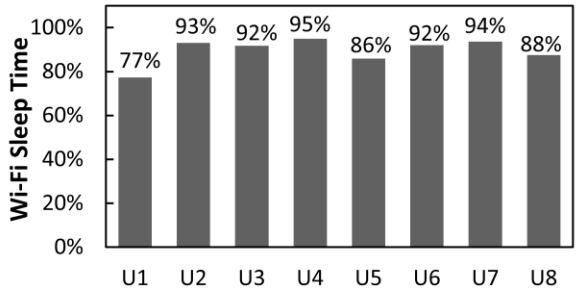

Figure 14: Wi-Fi interface sleep time calculated based on the traces of eight real users.

sumed 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 comparing with the case without DozyAP. Thus, we measure 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 Figure 12 show the results. We can see that DozyAP never increase the total energy. Instead, it can save the total energy by 12.2% to 32.9% for the five applications, which is almost the same as the result of average power. As we will show in Section 5.3, DozyAP introduces very little network latency which has negligible impact on 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 4, with the current commercially available smartphones, forcing the SoftAP to go to sleep or wakeup can be only achieved by turning off/on the Wi-Fi interface. That results in significant overhead (see Figure 11). If we can 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 [9] style driver which implements most MAC layer functions in driver rather than in firmware), DozyAP should be able to save significantly more power. Therefore, we measure how much time DozyAP can put the Wi-Fi interface of a mobile SoftAP to sleep. Figure 13 shows the results. We can see that the Wi-Fi interface of a SoftAP can

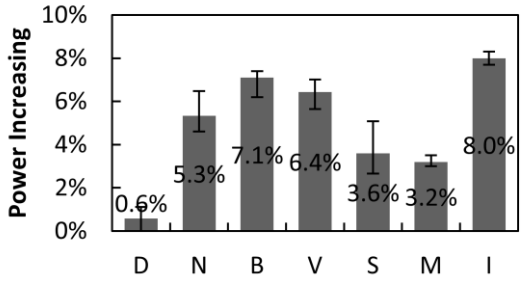

Figure 15: Power increasing of the Nexus One client in idle, busy download and the five applications.

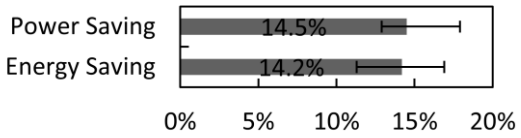

Figure 16: Power saving and energy saving of the SoftAP with two applications running on single client.

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 evaluate the Wi-Fi interface sleep time with the real traces of the eight users in MoodSense [21]. To do it, we wrote a program to analyze the packet inter-arrival 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 ignore all the inter-packet arrival intervals larger than 5 minutes. That is, for any intervals larger than 5 minutes, we treat 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 minutes to read a long news article or Wi-Fi tethering might not be turned off even the user stopped using the phone for 5 minutes. Figure 14 shows the calculated results. We can 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, multi-application real user traffic. The numbers in Figure 14 are higher than the ones in Figure 13. The reason is because the experiments in Figure 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 measure the power consumption of the Nexus One client in the idle case, busy download and the five applications. Figure 15 shows the results. We can see that DozyAP increases the power consumption of the client by less than 7.1% for the five applications. The reason is because that the client needs to wake up to receive the sleep requests from the SoftAP and

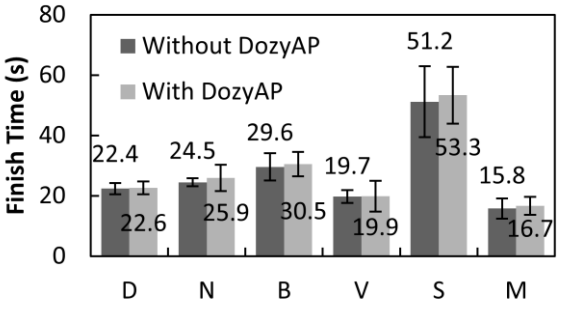

Figure 17: Finish time of busy download and the five applications.

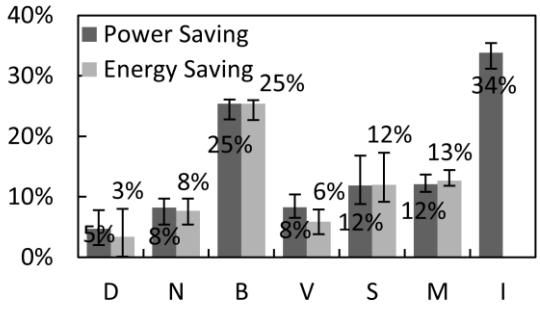

Figure 18: Power saving and energy saving of the SoftAP with two clients.

send back the sleep responses when the network is idle. Thus, the idle case introduces the highest overhead but it is still only 8%. Comparing to the large power saving of the SoftAP, this small overhead is acceptable.

Multiple applications on single client. In some cases, multiple applications may run on a single client simultaneously. We evaluate DozyAP in a typical scenario where a user is reading news in the foreground meanwhile listening online music in the background. To do it, we first started *Douban FM* (which is popular app in China like *Last.fm*). Once the music began to load, we started the news reading program (the same as before) immediately. Figure 16 shows the power saving and energy saving of the SoftAP. The average result over ten experiments is 14.5% and 14.2% for power saving and energy saving, respectively.

5.3 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 webpage request to the time when the webpage 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 evaluate the *finish time* of loading content which is the sum of the page load time of all the webpage requests in an application. The WebView object used in our test program is able to tell when a webpage is loaded. In the experiments, we sent out all the webpage requests of an application one by one *without any time interval* and calculated the total finish time.

Figure 17 shows the average result and the variance in busy download and the five applications. We can 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 webpage. 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.

5.4 Multiple Clients

We evaluate the performance of DozyAP when two clients are associated with a SoftAP. One client is a Nexus One smartphone and the other is a Kindle Fire tablet. Each client ran the same programs simultaneously. Figure 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 single client scenario. However, the saving in download case does not drop as much as other applications. It is because in the download case the 3G connection was always fully utilized. No matter a client or multiple clients were downloading, the 3G connection was always the bottleneck so that the sleep time for the SoftAP would not drop much. Another finding is that the 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.

It is worth noting that, in the rare case where many clients share the same SoftAP, the energy gain of DozyAP becomes less and the overhead on clients may increase significantly. DozyAP can disable and enable the sleep protocol on the fly, depending on the number of clients. In our implementation, we disable the sleep protocol if the SoftAP has more than two clients and enable the sleep protocol if there are only one or two clients.

6. DISCUSSIONS AND FUTURE WORK

DozyAP requires installing a loadable module on a client, which restricts the number of clients it can support. It may be hard or impossible 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. The loadable module in our implementation can immediately work on all the Android-based smartphones and tablets. It can also be easily ported to the other devices running a Linux-style OS kernel

including iOS. For Windows and Windows Phone based devices, a similar approach can be used as well. One can implement the client part in a loadable Network Driver Interface Specification (NDIS) [6] driver without modifying the source code of the OS. Therefore, DozyAP is able to support a wide range of diverse Wi-Fi client devices.

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 fast too. 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 100ms. As shown in Figure 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.

7. RELATED WORK

Wi-Fi power saving. There has been a lot of research effort devoted to power saving in Wi-Fi [11-13, 15, 17, 19, 23, 25-28], focusing on improving the existing PSM in general or targeting at specific applications or usage scenarios. To name some recent work, Catnap [15] exploits the bandwidth discrepancy between Wi-Fi and broadband to save energy for mobile devices. NAPman [28] employs an energy-aware scheduling algorithm to reduce energy consumption by eliminating unnecessary retransmissions. SleepWell [25] 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 [20, 32], the authors propose to extend the IEEE 802.11 standard to support power saving access points for multi-hop 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 [29] 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.

Traffic-driven design. Adapting to traffic load for better sleeping is not a new idea [11, 27]. 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) [19, 23]. DozyAP builds on top of the basic techniques and applies them to Wi-Fi tethering scenario. Furthermore, DozyAP can be improved by leveraging existing literature, e.g., by traffic shaping [13, 15, 26] and sleeping in short intervals [23].

Sleep scheduling. Sleep/wake scheduling has been extensively studied in Bluetooth domain, e.g., [16, 22] and sensor network domain, e.g., [14, 24, 31]. 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 [3] 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 420mw in idle case. We believe MiFi devices can benefit from DozyAP design if they are programmable.

In addition, MyWi Ondemand [5] makes Wi-Fi tethering easy to use for an iPhone and an iPad paired over Bluetooth. When a user leaves a Wi-Fi network and uses her iPad, Wi-Fi tethering can be automatically enabled between her iPad and her iPhone. MyWi Ondemand provides a convenient way to decide when to enable and disable Wi-Fi tethering but does not save power when Wi-Fi tethering is enabled.

8. CONCLUSIONS

In this paper we have studied the power efficiency of Wi-Fi tethering. We show that Wi-Fi tethering on existing smartphones is power hungry and wastes energy unnecessarily, but there are many opportunities to save power by putting a mobile SoftAP to sleep. We propose DozyAP sys-

tem 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 on-going 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.

ACKNOWLEDGEMENTS

We sincerely thank our shepherd, Dr. Kameswari Chebrolu, as well as anonymous reviewers, whose comments and feedback helped improve this paper. We also thank Dr. Chiu C. Tan (Temple University), Ahmad Rahmati (Rice University), and our colleagues at Wireless and Networking Group, Microsoft Research Asia for fruitful discussions and valuable suggestions. The W&M team was supported in part by NSF grants CNS-1117412 and CAREER Award CNS-0747108.

REFERENCES

[1] Android WebKit package and WebView class, http://developer.android.com/reference/android/webkit/package-summary.html
[2] Broadcom, BCM4329. http://www. broadcom.com.
[3] MiFi. http://en.wikipedia.org/wiki/MiFi.
[4] Monsoon Power Monitor. http://www.msoon.com/LabEquipment/PowerMonitor/.
[5] MyWi and MyWi Ondemand. http://intelliborn.com/mywi.html.
[6] Network Driver Interface Specification (NDIS), http://msdn.microsoft.com/en-us/library/ff559102.aspx.
[7] Tcpdump. http://www.tcpdump.org/.
[8] The Android Market, https://market.android.com/.
[9] The MadWifi project. http://madwifi-project.org/.
[10] Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications. *IEEE Std 802.11*, 2007.
[11] M. Anand, E. Nightingale, and J. Flinn. Self-Tuning Wireless Network Power Management. In *MobiCom*, 2003.
[12] D. Bertozzi, L. Benini, and B. Ricco. Power Aware Network Interface Management for Streaming Multimedia. In *IEEE WCNC*, 2002.
[13] S. Chandra and A. Vahdat. Application-specific network management for energy-aware streaming of popular multimedia formats. In *USENIX ATC*, 2002.
[14] T. Dam and K. Langendoen. An Adaptive Energy-Efficient MAC Protocol for Wireless Sensor Networks. In *SenSys*, 2003.
[15] F. Dogar, P. Steenkiste and K. Papagiannaki. Catnap: Exploit High Bandwidth Wireless Interfaces to Save Energy for Mobile Devices. In *Mobisys*, 2010.
[16] S. Garg, M. Kalia and R. Shorey. MAC Scheduling Policies for Power Optimization in Bluetooth: A Master Driven TDD Wireless System. In *IEEE VTC*, 2000.
[17] Y. He and R. Yuan. A Novel Scheduled Power Saving Mechanism for 802.11 Wireless LANs. *IEEE Transactions on Mobile Computing*, 8(10):1368–1383, 2009.
[18] J. Huang, Q. Xu, B. Tiwana, Z. M. Mao, M. Zhang and P. Bahl. Anatomizing Application Performance Differences on Smartphones. In *MobiSys*, 2010.
[19] R. Krashinsky and H. Balakrishnan. Minimizing energy for wireless web access with bounded slowdown. In *MobiCom*, 2002.
[20] Y. Li, T. D. Todd and D. Zhao. Access Point Power Saving in Solar/Battery Powered IEEE 802.11 ESS Mesh Networks. In *IEEE QShine*, 2005.
[21] R. LiKamWa, Y. Liu, N. D. Lane and L. Zhong. Can Your Smartphone Infer Your Mood?. In *PhoneSense* workshop, 2011.
[22] T. Lin and Y. Tseng. An Adaptive Sniff Scheduling Scheme for Power Saving in Bluetooth. In *IEEE Wireless Communications*, 9(6):92-103, 2002.
[23] J. Liu and L. Zhong. Micro Power Management of Active 802.11 Interfaces. In *MobiSys*, 2008.
[24] G. Lu, N. Sadagopan and B. Krishnamachari. Delay Efficient Sleep Scheduling in Wireless Sensor Networks. In *Infocom*, 2005.
[25] J. Manweiler and R. R. Choudhury. Avoiding the Rush Hours: WiFi Energy Mangement via Traffic Isolation. In *Mobisys*, 2011.
[26] C. Poellabauer and K. Schwan. Energy-aware traffic shaping for wireless real-time applications. In *RTAS*, 2004.
[27] D. Qiao and K. Shin. Smart Power-Saving Mode for IEEE 802.11 Wireless LANs. In *Infocom*, 2005.
[28] E. Rozner, V. Navda, R. Ramjee, and S. Rayanchu. NAPman: Network-Assisted Power Management for WiFi Devices. In *MobiSys*, 2010.
[29] A. Sharma, V. Navda, R. Ramjee, V. N. Padmanabhan, and E. M. Belding. Cool-tether: energy efficient on-the-fly Wi-Fi hotspots using mobile phones. In *CoNEXT*, pages 109-120, 2009.
[30] H. Wirtz, R. Backhaus, R. Hummen and K. Wehrle. Establishing Mobile Ad-Hoc Networks in 802.11 Infrastructure Mode. In *WiNTECH* demo session, 2011.
[31] Y. Wu, S. Fahmy and N.B. Shroff. Optimal Sleep/Wake Scheduling for Time-Synchronized Sensor Networks with Qos Guarantees. In *IEEE/ACM Transactions on Networking*, 17(5):1508-1521, 2009.
[32] F. Zhang, T. D. Todd, D. Zhao and V. Kezys. Power Saving Access Points for IEEE 802.11 Wireless Network Infrastructure. *IEEE Transactions on Mobile Computing*, Vol. 5, No. 2, 2006.