Characterizing and Optimizing Background Data Transfers on Smartwatches

Yi Yang and Guohong Cao
Department of Computer Science and Engineering
The Pennsylvania State University
Email: {yzy123, gcao}@cse.psu.edu

***Abstract*—Smartwatches are quickly gaining popularity, but their limited battery life remains an important factor that adversely affects user satisfaction. To provide full functionality, smartwatches are usually connected to phones via Bluetooth. However, the Bluetooth power characteristics and the energy impact of Bluetooth data traffic have been rarely studied. To address this issue, we first establish the Bluetooth power model based on extensive measurements and a thorough examination of the Bluetooth implementation on Android smartwatches. Then we perform the first in-depth investigation of the background data transfers on smartwatches, and find that they are prevalent and consume a large amount of energy. For example, our experiments show that the smartwatch's battery life can be reduced to one third (or even worse) due to background data transfers. Such high energy cost is caused by many unnecessary data transfers and the energy inefficiency attributed to the adverse interaction between the data transfer pattern (i.e., frequently transferring small data) and the Bluetooth energy characteristics (i.e., the tail effect). Based on the identified causes, we propose four energy optimization techniques, which are fast dormancy, phone-initiated polling, two-stage sensor processing, and context-aware pushing. The first one aims to reduce tail energy for delay-tolerant data transfers. The latter three are designed for specific applications which are responsible for most background data transfers. Evaluation results show that jointly using these techniques can save 70.6% of the Bluetooth energy.**

I. Introduction

Smartwatches have become the most popular wearable devices, which bring mobile applications and important notifications straight to the user's wrist [1]. However, they are still suffering from the limited battery life [2]. For example, based on our experience, a fully charged LG Urbane smartwatch often cannot last for a whole day when it is simply used for reading notifications and emails. Thus, it will be of great value to characterize the energy consumption on smartwatches and propose energy saving solutions.

To provide full functionality, smartwatches need to be paired with a phone, and the communication between them is enabled by Bluetooth [3]. However, little study has been done to investigate the Bluetooth power characteristics and the energy impact of Bluetooth data traffic. To address this issue, we first build the Bluetooth power model based on extensive measurements and a thorough examination of the Bluetooth implementation on Android smartwatches. We found that the Bluetooth interface on smartwatches can operate at two modes with different power levels. In particular, the Bluetooth interface is put into the high-power active mode when transferring data, and switches to the low-power sniff mode to save energy when there is no data traffic. The mode transition is controlled by an inactivity timer, whose timeout value can be as high as several seconds. Thus, it is possible that the Bluetooth interface continues to consume a substantial amount of energy before the timer expires (referred to as the *tail effect*), even when there is no network traffic. Based on the observed power characteristics, the Bluetooth power model is established.

We perform an in-depth study of the energy impact of background data transfers, which are usually delay-tolerant (unrelated to user interactions) and can be optimized more aggressively. By collecting and analyzing smartwatch packet traces, we found that background data transfers are prevalent and generated for multiple purposes such as polling for updates, offloading the sensor data to the phone for processing, and pushing notifications. According to our experiments, the smartwatch's battery life can be shortened to one third by running two very popular applications that generate background data transfers. The high energy cost is due to the following reasons. First, some applications generate small data transfers too frequently when running in the background. For example, Sleep as Android, a popular sleep monitoring application, offloads the sensor data to the phone every twenty seconds, which is too aggressive as there is no user interaction during sleep. Second, transferring small data frequently leads to serious energy inefficiency due to the tail effect.

Based on the above findings, we propose four techniques to optimize background data transfers, which are fast dormancy, phone-initiated polling, two-stage sensor processing, and context-aware pushing. Fast dormancy is a technique widely used in cellular networks [4] to reduce tail energy for delay-tolerant data transfers. We adopt this idea and implement it on smartwatches to reduce the tail energy of Bluetooth. The latter three techniques target on specific applications which are responsible for most background data transfers, i.e., polling, data offloading and pushing. Phone-initiated polling leverages the cooperation between a smartwatch and the paired phone to fulfill the polling task while reducing the need of transferring data between them. Two-stage sensor processing enables sensor based applications to make smarter offloading decisions

978-1-5090-6501-1/17/$31.00

by preprocessing the sensor data on smartwatches, in order to reduce the offloading traffic. Context-aware pushing leverages the smartwatch's screen status as an indicator to adaptively adjust the periodicity for the pushing traffic to save energy. We evaluate the proposed techniques based on trace-driven simulations and case studies. Evaluation results show that jointly using these techniques can save 70.6% of the Bluetooth energy, and the latter three techniques can significantly reduce the data transfer volume for specific applications to save energy. To summarize, our contributions are as follows:

- We thoroughly examine the Bluetooth implementations on Android smartwatches and characterize the tail effect in Bluetooth. Then the Bluetooth power model is built.
- We perform the first characterization of background data transfers on smartwatches. We found that background data transfers are prevalent, and many unnecessary small data transfers result in serious energy inefficiency due to the tail effect.
- We propose four optimization techniques, which are fast dormancy, phone-initiated polling, two-stage sensor processing, and context-aware pushing. The first one can save tail energy for delay-tolerant data transfers. The latter three optimize specific applications which generate most background data transfers. Evaluation results show that jointly using these techniques can save 70.6% of the Bluetooth energy.

The rest of this paper is organized as follows. Section II discusses related work. Section III presents some preliminaries. Section IV introduces the Bluetooth power model. We characterize background data transfers in Section V and introduce energy optimizing techniques in Section VI. Section VII evaluates the performance of the proposed techniques. Section VIII concludes the paper.

II. Related Work

The energy characteristics of the wireless interfaces (i.e., WiFi, 3G and LTE) on smartphones have been extensively studied [5], [6]. Based on the energy characteristics, lots of research has been done to analyze the energy impact of data traffic on smartphones and propose energy saving solutions [7], [8], [9]. Huang *et al*. [7] found that off-screen data traffic accounts for 58.5% of the total radio energy consumption and proposed to use fast dormancy and batching to save energy. Chen *et al*. [8] performed extensive measurements and analysis of the energy drain of 1520 smartphone. They found that 3G/LTE cellular data traffic accounts for 11.7% of the total energy drain, and a significant portion of the cellular energy drain (10.5% out of 11.7%) is tail energy. Qian *et al*. [9] identified the energy inefficiency of the periodic transfers in mobile applications and investigated various traffic shaping and resource control algorithms. Many researchers proposed to defer and aggregate the data traffic to save energy [10], [11]. Compared to the extensive work on smartphones, little effort has been made toward measuring and optimizing the energy consumption of the Bluetooth data transfers on smartwatches.

Recently some work has been done to improve the battery life of smartwatches [2], [12]. Min *et al*. [2] studied the practices for smartwatch battery use and management based on a combination of online surveys and a user study involving 17 Android smartwatch users. In [12], a systematic study was done to characterize the performance of Android Wear OS. It focused on the CPU execution inefficiency and its impact on energy consumption. Other studies proposed to offload tasks from smartwatch to phone in order to save CPU energy [13], [14], [15], [16]. In [13], a case study has been done based on three augmented reality apps. Experimental results show that all computationally intensive tasks should be offloaded to the phone in order to reduce the app's latency and save energy for wearable devices. Although there are some researches on Bluetooth such as characterizing its performance [17] and enhancing its functionalities [18], none of them focuses on the energy consumption.

III. Preliminaries

We first give an overview of the Bluetooth technology, and then introduce how the Bluetooth interface operates in different modes. At the end, we discuss the Bluetooth implementation on Android smartwatches.

A. Bluetooth Overview

Bluetooth [3] is a wireless technology standard for exchanging data over a short distance, which is widely adopted by mobile devices to communicate with each other. As the latest version, Bluetooth 4.0 (also known as Bluetooth Smart or Bluetooth Low Energy) and its update iterations 4.1 and 4.2 are designed primarily to achieve considerably reduced power consumption for devices to remain connected to each other and frequently exchange data.

The main Bluetooth protocols used by mobile devices are introduced as follows. The Asynchronous Connection-Less or ACL protocol is the most popular baseband layer communication protocol used to carry general data frames. The Logical Link Control and Adaptation Layer Protocol (L2CAP) is layered over the baseband protocol and resides in the data link layer. One of its core functions is to multiplex multiple upper layer connections over a single link. On top of L2CAP, there are two main protocols: Radio Frequency Communication (RFCOMM) and Attribute Protocol (ATT). RFCOMM is a transport protocol which provides reliable data stream for general data communications. ATT is a protocol designed for the Generic Attribute (GATT) profile which provides a hierarchical data structure to define attributes that can be discovered and transferred between connected devices.

B. Bluetooth Modes

Bluetooth protocols are driven by a system clock with a frequency of 3.2 kHz, which yields a period of 312.5 μs. In an ACL link, communications between devices are based on a period of 625 μs (twice the period of the basic clock), which is known as a slot. As shown in figure 1, a slave (smartwatch) can send data to the master (phone) in slave-to-master slots,

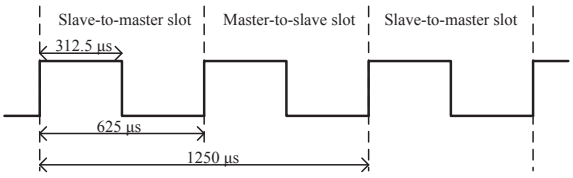


Fig. 1. Slots in an ACL link. A slave (smartwatch) can send data to the master (phone) in slave-to-master slots, and receive data from the master in master-to-slave slots.

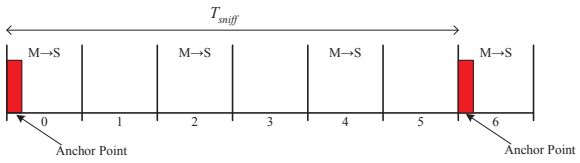


Fig. 2. Illustration of the anchor points in the sniff mode with a t_{sniff} interval of six slots. M→S represents master-to-slave slots.

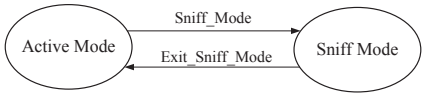


Fig. 3. Transition between the active mode and the sniff mode

and receive data from the master in master-to-slave slots. In the active mode, the slave listens in every ACL master-to-slave slots for packets. To reduce energy consumption, the Bluetooth core specification [3] defines three low-power modes: sniff mode, hold mode and park mode. The hold mode and the park mode will not be discussed in this paper since they are not enabled on smartwatches.

In the sniff mode, the master-to-slave slots that the slave listens are reduced in order to save energy. Specifically, the slave can only receive data in specified slots, known as anchor points, which are spaced regularly with an interval of t_{sniff} (illustrated in Figure 2). If a packet is received at an anchor point, the slave does not need to wait for the next anchor point to receive the following packets. Instead, it will continue to listen in the next $N_{timeout}$ master-to-slave slots, and the $N_{timeout}$ counter will restart every time a packet is received. Thus, the sniff mode can support the same data rate for data flowing as the active mode. The only difference is that there is up to t_{sniff} delay before receiving the first data packet in the sniff mode. Sniff sub-rating provides a means to further reduce power consumption by increasing the time between sniff anchor points (i.e., t_{sniff}). According to our packet traces from smartwatches, the value of t_{sniff} in the sniff mode is the same as that in the sniff sub-rating mode (798 baseband slots (498.75 ms)).

In bluetooth, the Host Controller Interface (HCI) provides a command interface to the baseband controller and link manager. The transition between the active mode and the sniff mode is controlled by HCI commands depending on the implementation of upper layer protocols. As shown in Figure 3, HCI command Sniff_Mode is used to place the link into sniff mode, and HCI command Exit_Sniff_Mode is used to end the sniff mode and switch the link to active mode.

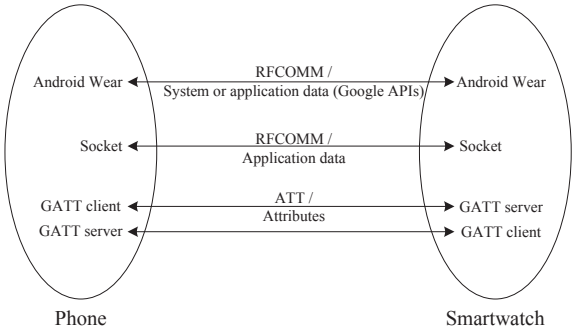


Fig. 4. Main ways to transfer data between a smartwatch and the paired (connected) phone. A single ACL link is shared among all the upper layer network connections.

TABLE I
MOBILE DEVICES AND SYSTEM VERSIONS

	Device Name	Android Version	Bluetooth Version
Smartwatch	LG Urbane	Wear 2.0	4.1
	Sony Smartwatch 3	Wear 1.5	4.0
Phone	Nexus 5x	Android 7.0	4.2
	LG G4	Android 5.1	4.1

C. Bluetooth on Android Smartwatches

Modern smartwatches need to be paired with a phone to provide full functionality. In Android, the Android Wear application is the default application used for phones to manage the pairing procedure and control the paired Android smartwatches. The pairing process is initiated the first time smartwatches are opened, which requires user authorizations (by confirming the pin code displayed on the phone that a smartwatch requests to pair with). Once the pairing process is completed, smartwatches will automatically connected to the paired phone when they are within the Bluetooth range of each other. There are three main ways to transfer data between a smartwatch and the paired (connected) phone, as shown in Figure 4. First, the Android Wear application constantly maintains a RFCOMM connection, which is used for data transfers generated by the system or Google APIs. Second, developers can use the BluetoothSocket class to establish a RFCOMM connection for data communications. Third, GATT services can be created to exchange attributes (usually small data in certain formats). All these network connections share a single ACL link, which remains opened until all upper layer connections are closed. Since the network connection created by the Android Wear application is constantly maintained, the underlying ACL link is never closed (when the smartwatch and the paired phone are within the Bluetooth range).

IV. BLUETOOTH POWER MODEL

We first introduce the methodology used to measure the power consumption of the smartwatch, and then build the Bluetooth power model. A brief description of the devices used in this paper is shown in Table I, where LG Urbane is paired with Nexus 5x and Sony Smartwatch 3 is paired with LG G4.

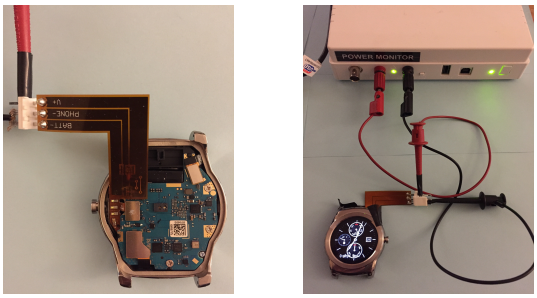


Fig. 5. Flex PCB based battery interceptor for the Lg Urbane smartwatch to be connected with the Monsoon Power Monitor

TABLE II
POWER LEVEL OF MAJOR COMPONENTS ON THE LG URBANE SMARTWATCH

	Power (mW)	Duration (sec)
Sleep (base)	9.5 ± 0.4	-
Wake up (CPU on)	40.3 ± 1.2	-
Display (lowest brightness)	60.8 ± 3.5	-
Bluetooth idle	9.6 ± 0.4	-
Bluetooth tail	40.5 ± 0.8	9.7 (ATT), 4.5 (RFCOMM)
Bluetooth data	83.1 ± 0.7	-
Bluetooth demotion	82.3 ± 2.8	0.3 ± 0.1

The base power is included in all power readings except the display.

A. Methodology

To measure the power consumption, we need to intercept the battery connection and use an external power monitor to provide power supply for the smartwatch. Different from phones, the battery connector of smartwatches like Lg Urbane is very tiny, and cannot be directly connected to an external power monitor. To solve this problem, we design a battery interceptor based on Flex PCB, which is a very thin circuit board that can be easily bent or flexed. The interceptor is connected with the smartwatch's mainboard and the battery through the corresponding battery connector, which is Hirose BM22-4 for Lg Urbane, and uses a customized circuit to modify the battery connection. As shown in Fig. 5, we use this interceptor to connect the Lg Urbane smartwatch with the Monsoon Power Monitor to measure the power consumption.

B. Power Model

Table II summarizes the power level of major components on the LG Urbane smartwatch. The energy consumption of transferring data via Bluetooth depends on the implementation of upper layer protocols.

Figure 6 shows the state transitions of using the Bluetooth interface to transfer data. When there is no data transfer, the Bluetooth interface stays in the low-power sniff mode and consumes very little energy (0.1 mW after subtracting the base power). Once a data request arrives, the Bluetooth interface is immediately turned into the active mode to transfer data. An inactivity timer t_{tail} is triggered to control when to switch back to the sniff mode, which is reset every time a packet is sent/received. The value of t_{tail} varies from

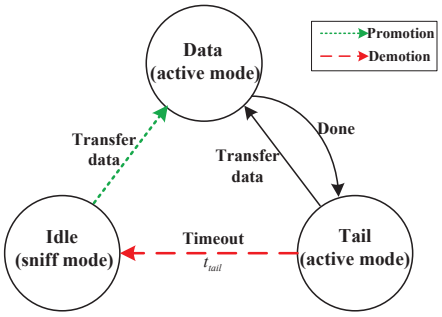


Fig. 6. State transitions of using the Bluetooth interface to transfer data

different upper layer implementations, which is 9.7 seconds for the L2CAP/ATT connection and 4.5 seconds for the L2CAP/RFCOMM connection. As a result, the Bluetooth interface stays in the high-power active mode for several seconds (i.e., the tail) after the data transfer finishes. The purpose of the tail is to avoid the latency of receiving data (up to t_{sniff} in the sniff mode) if the next phone-to-smartwatch data request arrives before t_{tail} expires. However, a large fraction of energy may be wasted in the tail relative to that used for data transmissions.

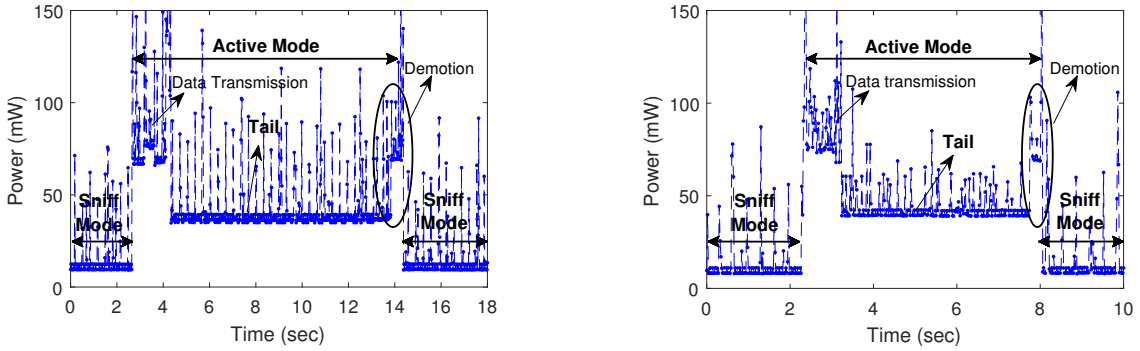
When t_{tail} expires, the Bluetooth interface starts the process of transition from the active mode to the sniff mode. During the process, the smartwatch exchanges control packets (i.e., LMP_sniff_req and LMP_accepted) with the phone to negotiate the sniff parameters (e.g., t_{sniff}). We refer to this process as the demotion process, and its duration as the demotion time (t_{dmt}), which is 0.3 seconds according to our measurements. There exists a similar promotion process of transition from the sniff mode to the active mode. The promotion process is simpler and happens simultaneously with the data transmission (i.e., the data transmission can start at the sniff mode and does not have to wait for the promotion process to finish). Thus, the promotion process does not introduce extra delay for data transmissions and has negligible impact on energy.

As shown in Figure 7, the power consumption of using the Bluetooth interface to transfer data can be generalized into three parts: data, tail and demotion, and the power of these parts are denoted as P_{data}, P_{tail} and P_{dmt}, respectively. For a data transfer T_i, its energy consumption, $E(T_i)$, can be modeled as follows. Suppose T_i starts at t_i and lasts for d_i time, and the next data transfer is T_j. The interval between data transfer T_i and T_j is $\Delta t = t_j - t_i - d_i$. Then $E(T_i)$ can be calculated depending on Δt, as shown in Equation 1.

$$E(T_i) = \begin{cases} P_{data} \times d_i + P_{tail} \times \Delta t, & \text{if } \Delta t < t_{tail}; \\ P_{data} \times d_i + P_{tail} \times t_{tail} + P_{dmt} \times (\Delta t - t_{tail}), & \\ \qquad \text{if } \Delta t \geq t_{tail} \text{ and } \Delta t < t_{tail} + t_{dmt}; & \\ P_{data} \times d_i + P_{tail} \times t_{tail} + P_{dmt} \times t_{dmt}, & \\ \qquad \text{Otherwise.} \end{cases} \quad (1)$$

C. Model Validation

The Bluetooth power model is validated as follows. We perform 1 KB, 2 KB, and 10 KB data transfers using the



(a) L2CAP/ATT connection (b) L2CAP/RFCOMM connection

Fig. 7. Power consumption of the Bluetooth radio while transferring 500 bytes data (measured by the LG Urbane smartwatch)

TABLE III
BACKGROUND DATA TRANSFERS GENERATED BY THE SYSTEM

Service name	Bytes per session (KB)	Session length (sec)	# of sessions per day
Google sync	6.4	1.8	48
Google play	52.2	6.9	23
Google now card	4.2	1.0	27

L2CAP/ATT protocol and the L2CAP/RFCOMM protocol, respectively. For each data transfer, we vary the distance between the smartwatch (LG Urbane) and the phone (Nexus 5x) from 0.5 meters to 20 meters (the smartwatch becomes disconnected at a distance farther than 20 meters). The accuracy of the Bluetooth power model is evaluated by comparing the estimated energy consumption against the actual energy measured by a power monitor. According to the experimental results, we have the following observations. 1) The distance has little impact on the power consumption and the data rate of Bluetooth. 2) The estimation error of the Bluetooth power model is under 5% for all tests.

V. BACKGROUND DATA TRANSFERS ON SMARTWATCHES

This section characterizes background data transfers using packet traces collected on smartwatches, and analyzes their energy impact based on the power model introduced in the last section.

A. Packet Traces

We collected the Bluetooth packet traces on Lg Urbane using the HCI (Host Controller Interface) snoop log, a tool that captures all Bluetooth HCI packets. We tested 34 popular applications from the Google play store by running them in the background (opening the application and then press the power button). According to the traces, we found that all packets are transferred using the L2CAP/RFCOMM connection created and maintained by the Android Wear application.

B. Origins of Background Data Transfers

Understanding the origins of background data transfers is important to determine how to optimize these data transfers without affecting the application functionalities. Since all packets are transferred through the same network connection, the network parameters (e.g., channel/port number) cannot be used to distinguish the data traffic of each application.

For packets generated by the system, we manually inspect their content to identify their origins. We found that most packets are associated with (pre-installed) services provided by Google, and a keyword based approach can be used to identify packet sessions with different purposes. For example, a Google synchronization session starts with a packet containing the keyword "WearableSync" and Google account information, followed by a group of packets containing the synchronization data (e.g., fitness data, derived speed, location). The keyword "now-cards" can be used to find the leading packets for Google now cards service.

Table III lists the main system services that generate background data transfers, where Google sync. is performed every thirty minutes to synchronize its account information and service data (e.g., fitness data), Google now card service pushes information like news and weather from phone to smartwatch, and Google play service automatically downloads updates for softwares. These services do not generate data transfers often, but each data transfer session usually involves a large burst of data. We also observed that when the paired phone lost access to Internet, connection retry is performed every five seconds, and repeated for hundreds of times for each connection request.

To characterize the data traffic generated by user applications, we collected packet traces for each application individually to analyze their traffic patterns. We found that packets are transferred in sessions with a leading packet containing the application name, followed by a group of packets containing the application data. Thus, searching based on application names can be used to recognize the data transfers generated by each application. Table IV lists applications that generate most background data transfers, and their purposes are summarized as follows.

Polling is performed periodically by some applications to detect updates. For example, Telegram, a popular SMS application with more than 100 million downloads, polls a remote server (located at 149.154.175.50:80) every one minute

TABLE IV
APPLICATIONS GENERATING PERIODIC BACKGROUND DATA TRANSFERS

Origins of data transfers	Application name	Bytes per session (KB)	Periodicity (sec)
Polling	Telegram	0.2	60
	Accuweather	4.8	900
Offloading	Sleep as Android	0.9	20, 120*
	Cinch	0.3	5
Pushing	Endomondo	0.4	1
	Runkeeper	0.3 (1.2)†	1 (5)†

*: the periodicity observed is not fixed.
†: two types of data (fitness status and location information) are transferred separately with different periodicities.

to check updates. Using this kind of traditional polling schema, a lot of polling requests / responses are transferred between smartwatch and phone. In the next section, we propose a phone initiated polling schema to reduce the need of transferring data between them in order to save energy on smartwatch.

Data offloading: Many wearable applications leverage the rich sensors on smartwatches to monitor user behaviors. Considering the limited CPU and battery capability of smartwatches, these applications usually send the raw sensor data to their phone-side counterparts for processing. For example, Sleep as Android, a sleep monitoring application with more than 10 million downloads, sends collected sensor data to the phone for processing every twenty seconds (sometimes every two minutes), which is too frequent as there is no user interaction during sleep. As another example, Cinch, an application to track weight loss, sends heart rate data to the phone every five seconds, which will keep the Bluetooth interface in the high-power active mode and drain the smartwatch's battery very quickly.

Pushing: Google provides a set of APIs for applications to push notifications to smartwatch. Some applications aggressively use this feature to update their status. For example, two popular fitness applications, Endomondo (10 million downloads) and Runkeeper, use the phone to track fitness status (e.g., running distance and duration) and push the status to the smartwatch every one second. Runkeeper also pushes the location information (estimated by the phone) to the smartwatch every five seconds. By doing so, the latest status can be displayed once the user opens the smartwatch. However, this kind of aggressive pushing generates a huge number of data transfers and causes energy problems.

C. Energy Impact

We leverage the Bluetooth power model established in section IV to compute the energy consumption of data transfers. We first calculate the energy consumption for each application based on isolated traces, and then perform experiments to show the overall impact of background data transfers on the smartwatch's battery life.

1) Energy breakdown for each application: We collect a three-hour packet trace for each application (service) and calculate the energy consumption. According to the results

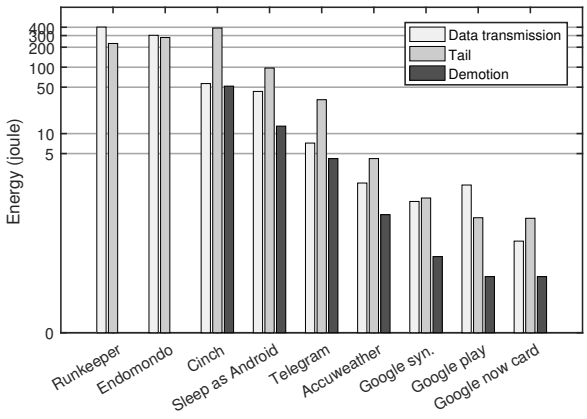


Fig. 8. Energy consumption of background data transfers generated by each application (service) based on isolated packet traces

(Figure 8), we have the following observations. First, applications that generate data transfers with small periodicities, such as Runkeeper, Endomondo and Cinch, are extremely energy hungry. For example, Runkeeper consumes 600 joules in three hours, which drains over ten percent of the battery (410 mAh) on a typical smartwatch. Note that these applications are expected to run for a very long time (if not constantly) in the background. Running several such applications simultaneously will cause severe energy drain. Second, tail energy accounts for a large proportion of the total energy, which could be optimized. Third, the system traffic (Google sync, Google play and Google now card) does not consume much energy due to its small volume and traffic pattern (transferring large bursts of data).

2) Total energy impact: To show the overall impact of background data transfers on energy, we compared the smartwatch's battery life under two scenarios. In scenario one (S1), a factory reset is performed before the experiment and the Bluetooth interface is turned off. Only the base power is consumed. In scenario two (S2), we install Sleep as Android and Telegram on the smartwatch and run them in the background. The reason for selecting these two applications is because of their popularity. Then, besides the base power, energy is also consumed by transferring data (Bluetooth) and running background tasks (CPU and sensor). In both scenarios, the smartwatch is kept in the sleep mode with the screen turned off.

Table V summarizes the results for LG Urbane and Sony Smartwatch 3. The battery life of both smartwatches in S1 is three times the battery life in S2. Note that in S2 we do not select the most energy hungry applications (shown in Figure 8). If those applications are chosen, the smartwatch's battery life will become significantly shorter. Figure 9 shows the energy breakdown of Lg Urbane in S2, where the base energy is obtained from S1, the Bluetooth energy is calculated based on the packet trace and the Bluetooth power model, and the remaining part is the CPU and sensor energy consumed by background tasks. As can be seen, the Bluetooth energy consumption of background data transfers accounts for 43%

TABLE V
BATTERY LIFE OF SMARTWATCHES UNDER DIFFERENT SCENARIOS

	Scenario	Battery Life (hours)
LG Urbane	S1	110
	S2	34
Sony Smartwatch 3	S1	225
	S2	78

S1: a factory reset is performed before the experiment and the Bluetooth is turned off. Only the base power is consumed.

S2: two applications (Sleep as Android and Telegram) are installed and running in the background.

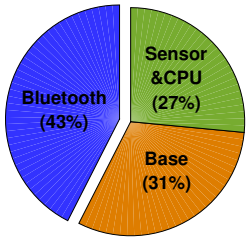


Fig. 9. Energy breakdown of the Lg Urbane smartwatch in S2. The base energy is obtained from S1. The Bluetooth energy is calculated based on the packet trace and the Bluetooth power model.

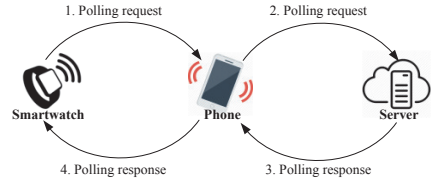
of the total energy, which is more than the base energy (31%) and the energy consumed by CPU and sensor (27%).

VI. Energy Optimizing Techniques

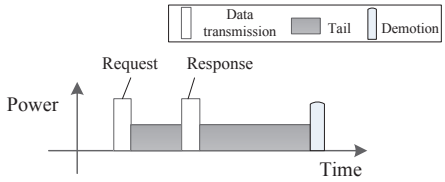
Background data transfers on smartwatches are prevalent and can drain battery quickly. There exist lots of works to optimize the background traffic for phones, which communicate with remote servers via the cellular networks [7], [8], [19]. However, these techniques cannot be easily adopted on smartwatches due to the following three factors. 1) Bluetooth has some unique characteristics different from cellular networks, which should be considered. 2) Smartwatches need to leverage the connected phone as a proxy to communicate with remote servers. Thus, more optimization opportunities can be found by exploring the cooperation between a smartwatch and the connected phone. 3) The sources of background data transfers are different. For example, a large amount of data transfers on smartwatches are originated from sensor based applications, which frequently offload the sensor data to the phone for processing. Optimizations targeting on these applications can significantly reduce the background data traffic. By considering the above factors, we propose four techniques to optimize background data transfers on smartwatches, which are fast dormancy, phone initiated polling, two-stage sensor processing, and context-aware pushing.

A. Fast Dormancy

Fast dormancy [4] is a technique widely used in cellular networks to reduce the tail energy by switching the cellular interface into the low-power state immediately after the data transmission. We adopt similar idea to reduce the tail energy



(a) Message flow diagram



(b) Energy consumption of the smartwatch

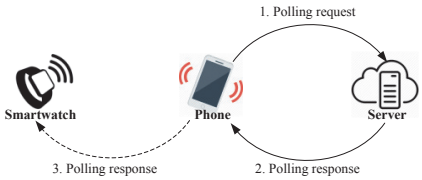
Fig. 10. An illustration of the traditional polling schema. In each polling, two packets (i.e., one request and one response) are transferred between smartwatch and phone.

of Bluetooth, i.e., actively switching the Bluetooth interface into sniff mode instead of waiting for the tail timer to expire. Although fast dormancy can save tail energy, it may bring extra energy waste due to the energy cost of the demotion process (see Section IV), if performed too aggressively. Another disadvantage of fast dormancy is that cutting the tail may introduce additional latency for data transmissions (i.e., up to t_{sniff} latency for receiving data in the sniff mode). Thus, fast dormancy should be adopted to optimize delay tolerant data transfers including background data transfers discussed in the paper.

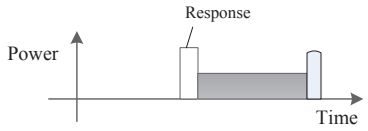
We have implemented fast dormancy on our LG Urbane smartwatch. A fast dormancy timer (shorter than t_{tail}) is used to control when to switch from the active mode to the sniff mode, which is reset every time a packet is sent/received. According to our packet traces, 99.7% intra-burst intervals are less than 0.5 seconds. So the timer is suggested to be more than 0.5 seconds to avoid unnecessary demotion energy cost. The packet sent/received events are monitored by using hcidump [20], a tool that records all HCI events in real time. Once the fast dormancy timer expires, a Sniff_Mode command is sent via hcitool [21] to start the demotion process of switching the Bluetooth interface into sniff mode.

B. Phone Initiated Polling

Polling is a basic method for applications to check updates from remote servers, which may result in lots of data transfers if performed periodically. As the traditional polling schema (Figure 10), a polling request is initiated by the smartwatch and sent to the remote server through the phone, and then the phone forwards the received response to the smartwatch. In each polling, two packets (i.e., one request and one response) are transferred between smartwatch and phone. To reduce the data transfer volume and save energy on smartwatches, we propose the phone initiated polling schema (Figure 11), in which the phone is responsible for polling and only sends responses to smartwatch if updates are detected. Then at most one packet (i.e., response with update information) needs to



(a) Message flow diagram



(b) Energy consumption of the smartwatch

Fig. 11. An illustration of the phone initiated polling schema. The phone is responsible for polling and only sends responses to smartwatch if updates are detected. In each polling, at most one response needs to be transferred between smartwatch and phone.

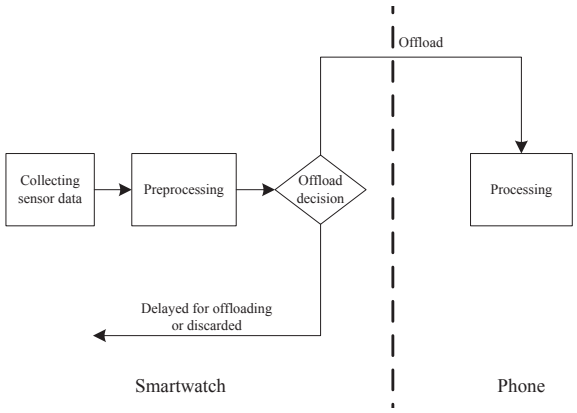


Fig. 12. The two-stage sensor processing framework. A preprocessing module is deployed on smartwatch to check the application requirement or the effectiveness of the sensor data. An offloading decision is made adaptively based on the preprocessing result.

be transferred in each polling between smartwatch and phone, and thus energy can be saved on smartwatches. We implement the logics of the phone initiated polling schema as an Android library, which provides a configure file to set parameters (e.g., polling periodicity), and exposes two main interfaces for developers to provide implementations for 1) communicating with the remote server and 2) parsing the response message to check updates.

C. Two-stage Sensor Processing

Smartwatches provide rich sensor resources that can be used for various purposes such as health monitoring or activity tracking. These sensor based applications usually keep running in the background to collect sensor data and offload the data to their phone-side counterparts for processing. An offloading timer is used to determine when to send the data (e.g., every one minute). To meet the real-time requirements, the offloading timer should be short enough so that the sensor data can be offloaded and processed quickly. However, using a short timer also leads to heavy background data traffic and thus wasting energy on smartwatches.

To reduce the offloading traffic, we propose a two-stage sensor processing framework, which enables sensor based applications to make smarter offloading decisions by preprocessing the sensor data on smartwatches. The idea is based on the following intuition. Usually a simple method can be used to check the application requirement and the effectiveness of the sensor data (i.e., whether the data are duplicated or have little variation), and then an adaptive offloading decision can be made based on the result. The framework is demonstrated in Figure 12, where the preprocessing module should be simple enough in order to prevent consuming too much CPU energy on the smartwatch. We rewrite two applications (Sleep as Android and Cinch) based on the framework.

1) Sleep as Android: This sleep monitoring application has been downloaded more than 10 million times from Goole play store. In the application, the smartwatch-side module keeps running in the background to collect accelerometer data and offload them to the phone-side module every twenty seconds. Based on the sensor data, the phone analyzes the user's sleep phases and draws graphs to display the analysis results. Since the graphs are displayed only when the phone is opened and there is usually no user interaction during sleep, a much longer offloading timer (i.e., larger than twenty seconds) can be used to reduce the amount of data transfers.

Delaying offloading the sensor data may lead to violation of the real-time requirements of the application. By reverse engineering the source code and reading the design documents, we found that the only task that has real-time requirements is smart wake up, which allows the user to specify a time window during which the user wants to be woken up, and looks for light sleep phases to trigger the alarm.

Our two-stage sensor processing framework can be easily adopted to reduce the offloading traffic while satisfying the real-time requirements. We added several lines of codes in the smartwatch-side module to check whether the current time falls in the wake up time window. If the condition is met, the offloading timer is set to twenty seconds (i.e., the same as the original timer) to satisfy the real-time requirements for the smart wake up task. Otherwise, the sensor data are offloaded every thirty minutes or when the screen turns on, which is an indicator that the application may be opened.

2) Cinch: This application continuously monitors the user's heart rate based on the heart rate sensor equipped on modern smartwatches. The heart rate value is read every five seconds and offloaded to the phone for analysis. The phone will issue an alert if abnormal condition is found (e.g., heart rate is out of the normal range). Due to the low sensitivity of the sensor, most heart rate values read are duplicated. We rewrite this application based on our two-stage sensor processing framework. At the smartwatch side, codes are added to check duplicated values and only offload data that has a different value from its previous one.

D. Context-Aware Pushing

Some applications aggressively push notifications to update their status on smartwatches with a very small periodicity (e.g.,

TABLE VI
TRAFFIC OPTIMIZATION FOR INDIVIDUAL APPLICATIONS

Application name	Techniques applied	ΔE (%)	ΔT (%)	ΔD (%)	ΔO (%)
Sleep as Android	Fast dormancy	-56.1%	-56.3%	+0.2%	-
	Two-stage sensor processing	-71.1%	-45.1%	-6.0%	+2.1%
	Fast dormancy + Two-stage sensor processing	-87.3%	-61.4%	-5.9%	+2.1%
Cinch	Fast dormancy	-69.4%	-69.5%	+0.1%	-
	Two-stage sensor processing	-76.4%	-58.8%	-7.8%	+3.0%
	Fast dormancy + Two-stage sensor processing	-92.8%	-75.2%	-7.8%	+3.0%
Telegram	Fast dormancy	-44.1%	-49.5%	+5.4%	-
	Phone initiated polling	-36.0%	-18.4%	-0.1%	-
	Fast dormancy + Phone initiated polling	-72.1%	-54.5%	-0.1%	-
Endomondo	Fast dormancy	+28.3%	-9.4%	+37.7%	-
	Context-aware pushing	-85.6%	-40.1%	+1.1%	+1.5%
Runkeeper	Fast dormancy	+22.8%	-6.4%	+29.2%	-
	Context-aware pushing	-87.6%	-34.2%	+1.0%	+1.5%

ΔE = ΔT + ΔD if only fast dormancy is applied.

one second used by Endomondo). We propose the context-aware pushing technique to reduce the pushing traffic based on the smartwatch's screen status, i.e., whether the screen is on or off. When the screen is on, a smaller periodicity is used in order to timely display the latest application status to the user. When the screen is off, a larger periodicity is used to reduce the pushing traffic. Additional messages are needed to reset the periodicity when the screen status changes.

The implementation is straightforward. The change of screen status is detected by registering a BroadcastReceiver in a background service for intents ACTION_SCREEN_ON and ACTION_SCREEN_OFF. The communication is implemented by using the Google MessageApi. This implementation is used as a prototype to evaluate the overhead (i.e., running additional codes and generating extra messages).

VII. PERFORMANCE EVALUATIONS

In this section, we first evaluate the performance of the proposed techniques for individual applications and then perform a case study to show the overall effectiveness of these techniques for all background data transfers.

A. Traffic Optimization for Individual Applications

1) Methodology: We collect packet traces for each application. To evaluate the energy saving of an optimization technique, we modify the packet traces according to the optimization results and calculate the energy consumption using the Bluetooth power model established in IV. Assumptions and experimental setups are listed below. 1) The fast dormancy timer is set to 0.5 seconds. 2) For Telegram, we assume all polling responses contain update information and need to be transferred to the smartwatch. So the evaluation results show the lower bound of the energy saving potential for the phone initiated polling schema. 3) Suppose the wake up time window in Sleep as Android is from 6 am to 8 pm. We leave the data transfers within the time window unmodified, and aggregate other data transfers every thirty minutes. 4) We collect a trace of screen status changes during walking, and use it to evaluate the context-aware pushing technique. The periodicity is set to the original value (i.e., one second) when the screen is on and set to one minute when the screen is off. In the evaluation, the Bluetooth energy consumption is calculated based on the power model of the LG Urbane smartwatch. Four metrics are considered as follows, which are normalized as percentage of the original Bluetooth energy consumption (calculated using unmodified traces).

- ΔE: the Bluetooth energy change (excluding the energy overhead).
- ΔT: the change of the tail energy.
- ΔD: the change of the demotion energy.
- ΔO: the energy overhead, which is consumed by running additional codes on smartwatches and transferring extra messages. The CPU energy overhead is captured by locally running the codes without transferring data.

2) Evaluation results: Table VI summarizes the evaluation results. We observe that fast dormancy does not always save energy due to the existence of the demotion process. For applications (Endomondo and Runkeeper) with a very small periodicity (one second), although fast dormancy can help reducing some tail energy ($\Delta T = -9 \sim -6\%$), a large amount of demotion energy is wasted (ΔD = $29 \sim 37\%$), and thus the overall energy is increased ($\Delta E = 22 \sim 28\%$). Techniques designed for specific applications can significantly reduce the data transfer volume for those applications to save energy. For example, two-stage sensor processing can save more than 70% energy for Sleep as Android and Cinch. Phone initiated polling saves at least 36% energy (lower bound) for Telegram. Context-aware pushing brings more than 85% energy saving for Endomondo and Runkeeper.

B. Case Study

To understand how much energy can be saved on a real smartwatch, we evaluate the proposed techniques using a packet trace that captures all background data transfers on a LG Urbane smartwatch. In the experiment, two applications (i.e., Telegram and Sleep as Android) are installed on the smartwatch due to their popularity. The packet trace contains background data transfers generated by these two applications and the Android system.

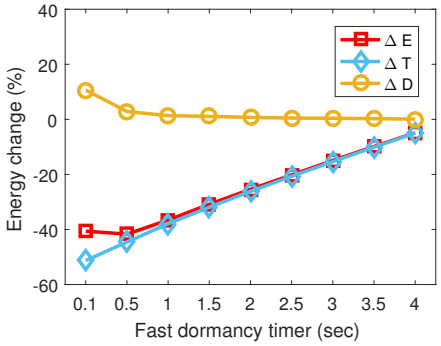


Fig. 13. Performance of fast dormancy with different settings

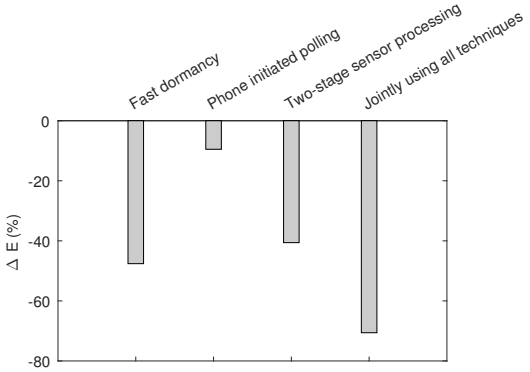


Fig. 14. Performance evaluations of the optimization techniques for all background data transfers

Figure 13 shows the results for fast dormancy with different settings. As can be seen, ΔD decreases as the fast dormancy timer becomes longer. A big reduction of ΔD can be observed between 0.1 seconds and 0.5 seconds. According to the packet trace, the intra-burst interval of over 60% data transmissions is within the range from 0.1 seconds to 0.5 seconds, so using a 0.1-second timer triggers a lot of demotion processes and wastes energy. As the timer grows from 0.5 seconds to 4 seconds, the overall energy change ($\Delta E = \Delta T + \Delta D$) is dominated by the tail energy part (ΔT). Fast dormancy achieves its best performance by setting the timer to 0.5 seconds. Figure 14 shows the performance of different optimization techniques. For each individual technique, the energy saving rate is 47.6% (fast dormancy with a 0.5-second timer), 9.5% (phone initiated polling) and 40.6% (two-stage sensor processing). Jointly using these techniques leads to 70.6% energy saving.

VIII. Conclusion

In this paper, we established the Bluetooth power model and performed the first in-depth study of the energy impact of background data transfers on smartwatches. We found that background data transfers are prevalent and generated for multiple purposes, many of which are unnecessary and result in serious energy inefficiency due to the tail effect. Based on these findings, we proposed four energy optimization techniques: fast dormancy, phone-initiated polling, two-stage sensor processing, and context-aware pushing. The first one can save tail energy for delay-tolerant data transfers. The latter three optimize specific applications which generate most background data transfers. The proposed techniques are evaluated based on trace-driven simulations and case studies. Evaluation results show that jointly using all techniques can save 70.6% of the Bluetooth energy, and the latter three techniques can significantly reduce the data transfer volume for specific applications. Our work is an important step towards understanding the data traffic on smartwatches, and determining better optimization strategies to extend the battery life.

Acknowledgment

This work was supported in part by the National Science Foundation (NSF) under grants CNS-1421578 and CNS-1526425.

References

[1] R. Rawassizadeh, B. A. Price, and M. Petre, "Wearables: Has the age of smartwatches finally arrived?" *Commun. ACM*, vol. 58, no. 1, pp. 45–47, 2014.
[2] C. Min, S. Kang, C. Yoo, J. Cha, S. Choi, Y. Oh, and J. Song, "Exploring current practices for battery use and management of smartwatches," in *ACM ISWC*, 2015.
[3] "Bluetooth technology," https://www.bluetooth.com.
[4] "Configuration of fast dormancy. rel. 8," http://www.3gpp.org.
[5] Y. Geng, W. Hu, Y. Yang, W. Gao, and G. Cao, "Energy-efficient computation offloading in cellular networks," in *IEEE ICNP*, 2015.
[6] W. Hu and G. Cao, "Energy-aware video streaming on smartphones," in *IEEE INFOCOM*, 2015.
[7] J. Huang, F. Qian, Z. M. Mao, S. Sen, and O. Spatscheck, "Screen-off traffic characterization and optimization in 3g/4g networks," in *ACM IMC*, 2012.
[8] X. Chen, N. Ding, A. Jindal, Y. C. Hu, M. Gupta, and R. Vannithamby, "Smartphone energy drain in the wild: Analysis and implications," in *ACM SIGMETRICS*, 2015.
[9] F. Qian, Z. Wang, Y. Gao, J. Huang, A. Gerber, Z. Mao, S. Sen, and O. Spatscheck, "Periodic transfers in mobile applications: Network-wide origin, impact, and optimization," in *ACM WWW*, 2012.
[10] B. Zhao, W. Hu, Q. Zheng, and G. Cao, "Energy-aware web browsing on smartphones," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 3, pp. 761–774, 2015.
[11] W. Hu and G. Cao, "Quality-aware traffic offloading in wireless networks," in *ACM MobiHoc*, 2014.
[12] R. Liu and F. X. Lin, "Understanding the characteristics of android wear os," in *ACM MobiSys*, 2016.
[13] B. Shi, J. Yang, Z. Huang, and P. Hui, "Offloading guidelines for augmented reality applications on wearable devices," in *ACM MM*, 2015.
[14] Z. Cheng, P. Li, J. Wang, and S. Guo, "Just-in-time code offloading for wearable computing," *IEEE Transactions on Emerging Topics in Computing*, vol. 3, no. 1, pp. 74–83, March 2015.
[15] Y. Yang, Y. Geng, L. Qiu, W. Hu, and G. Cao, "Context-aware task offloading for wearable devices," in *IEEE ICCCN*, 2017.
[16] D. Huang, L. Yang, and S. Zhang, "Dust: Real-time code offloading system for wearable computing," in *IEEE GLOBECOM*, 2015.
[17] R. Friedman, A. Kogan, and Y. Krivolapov, "On power and throughput tradeoffs of wifi and bluetooth in smartphones," *IEEE Transactions on Mobile Computing*, vol. 12, no. 7, pp. 1363–1376, 2013.
[18] A. A. Levy, J. Hong, L. Riliskis, P. Levis, and K. Winstein, "Beetle: Flexible communication for bluetooth low energy," in *ACM MobiSys*, 2016.
[19] T. Zhang, X. Zhang, F. Liu, H. Leng, Q. Yu, and G. Liang, "etrain: Making wasted energy useful by utilizing heartbeats for mobile data transmissions," in *IEEE ICDCS*, 2015.
[20] "hcidump," http://www.linuxcommand.org/man_pages/hcidump8.html.
[21] "hcitool," http://linuxcommand.org/man_pages/hcitool1.html.