Optimizing Background Email Sync on Smartphones

Fengyuan Xu[1,3], Yunxin Liu[1], Thomas Moscibroda[1], Ranveer Chandra[2], Long Jin[1,4], Yongguang Zhang[1], Qun Li[3]

[1]Microsoft Research Asia, Beijing, China [2]Microsoft Research, Redmond, WA, USA

[3]College of William and Mary, Williamsburg, VA, USA [4]Tsinghua University, Beijing, China

ABSTRACT

Email is a key application used on smartphones. Even when the phone is in stand-by mode, users expect the phone to continue syncing with an email server to receive new messages. Each such sync operation wakes up the smartphone for data reception and processing. In this paper, we show that this “cost of email sync” in stand-by mode constitutes a significant source of energy consumption, and thus reduces battery life. We quantify the power performance of different existing email clients on two smartphone platforms, Android and Windows Phone, and study the impact of system parameters such as email size, inbox size, and pull vs. push. Our results show that existing email clients do not handle email sync in an energy efficient way. This is because the underlying protocols and architectures are not designed for the specific needs of operating in stand-by mode. Based on our findings, we derive general design principles for energy-efficient event handling on smartphones, and apply these principles to the case of email sync and implement our techniques on commercial smartphones. Experimental results show that our techniques are able to significantly reduce energy cost of email sync by 49.9% on average with our experiment settings.

Categories and Subject Descriptors

C.2.2 **[Computer-communication Networks]**: Network Protocols - *Applications*

General Terms

Experimentation, Measurement, Performance

Keywords

Energy Efficiency, Smartphone, Email, Cellular Network

1. INTRODUCTION

Standby time, a battery’s life in standby mode, is important for smartphones to provide good power performance to users. Even though many smartphones claim a standby time of more than 10 days in their technical specifications, in practice their standby times are often much shorter, e.g., only two or three days or even less. The main reason for such a big gap is that when a smartphone remains in standby mode, it stays connected to the Internet through its cellular data interface (e.g., 3G), waiting for various incoming events, such as emails, short messages, instant messages, notifications of social applications (e.g., Twitter and Facebook), and many other push notifications.

This *connected standby* state, in which the screen is off while the network connectivity stays active, is of fundamental importance for mobile devices. Most users keep their phone in connected standby for a large fraction of time during the day. In this state, the reception of an incoming event (email, instant message…) wakes up the cellular data interface as well as the phone’s operating system (OS), to receive the set of packets over the network and process received data. Collectively, these operations consume a significant amount of power. For example, our measurements show that the energy cost of receiving a small email on an Android smartphone is more than 14,000mJ, which is as high as the energy consumed by a typical smartphone in standby mode for 10 minutes. In other words, the reception of a single email reduces the standby time of a smartphone by 10 minutes. Assuming the phone has a standby time of 10 days without receiving any events, its standby time will be reduced to less than 6 days (a reduction of more than 40%), if it receives 100 emails per day. This is despite the screen being off during the entire duration, and without any user interaction.

In this paper, we study the power performance of email sync in connected standby on smartphones. Email is a killer application on smartphones for most users, who expect their email clients to keep synchronized with one or more email servers even when the phone is in standby mode. We measure the energy consumption of existing email clients on two major smartphone platforms: Android and Windows Phone (WP). Our results show that email sync is indeed a major drain on existing platforms, and we observe that existing mobile email clients do not handle incoming emails in an energy-efficient way. The reason is that the underlying protocols, and the overall design of today’s mobile email clients do not take into account the specific characteristics and needs of operation in connected standby mode: data processing is not sufficiently de-coupled from network communication (thus preventing the 3G interface from quickly going to sleep), memory management and storage input/output (I/O) are un-optimized, and network protocols and interface operations are tailored for operations other than event reception.

Based on our findings, we formulate new design principles for energy-efficient event handling (and specifically email sync) on smartphones in connected standby. Applying these principles to the case of email sync, we develop 5 new techniques, each one addressing one of the shortcomings we have identified in existing systems.

- In current systems, long 3G tail times keep the 3G interface on for longer than is necessary. We advocate fast dormancy and adaptive 3G tail times based on email arrival patterns, to reduce the energy cost of the 3G tail.
- In current email clients, data transmission and data processing are coupled together, e.g., a final ACK packet is sent to the email server once the received email has been completely processed. Depending on the amount of processing time required to handle an incoming email, this coupling can prevent the 3G interface from going to sleep for a long time. We propose decoupling the data transmission from data processing to improve energy efficiency.
- In some email clients, the amount of data processing and memory/storage I/O required to handle an incoming email depends on the *size of the inbox*, leading to a particularly high energy cost for receiving emails when there are many of them in the email client's inbox. We mitigate this problem by using a small cache for incoming emails, to enable fast email processing.
- Flash storage operations on smartphones are slow, which leads to long data processing times, thus keeping the OS awake longer. We show that in connected standby, it therefore makes sense as a design principle to perform email reception *entirely in-memory.*
- Finally, in some clients, a new secure network connection is established for receiving every new incoming email, resulting in repeated TCP and SSL handshakes and a waste of energy. We solve this problem by reusing network connections across the reception of multiple emails.

Collectively, these techniques achieve significant reductions in the energy cost of email syncing. Specifically, we have implemented them by modifying an existing email client on commercial smartphones. Experimental results show that our revised email client is able to significantly improve email sync's energy efficiency, reducing the average energy cost by 49.9% and up to 77.9% if we put the 3G interface into sleep immediately after an email is received.

In summary, the main contributions of this paper are:

- We show that email sync in a connected standby state is a major source of energy consumption in today's smartphones, and that existing email clients are not optimized for operation in this mode.
- We derive design principles for energy-efficient event handling in connected standby, and propose techniques for reducing the energy required per email sync.
- We implement a new email client on smartphones and show that it is significantly more energy efficient.

The paper is organized as follows. Section 2 gives background information on email sync in smartphones. In Section 3, we present a measurement study on the email-sync energy cost of existing smartphone email clients, and derive general guidelines to make event reception energy-efficient in connected standby. Section 4 describes our novel techniques. We describe our implementation and evaluation results in Sections 5 and 6, respectively. In Section 7, we demonstrate that our techniques can also be used to improve energy efficiency when receiving events in other event-triggered applications. Section 8 surveys related work before we conclude in Section 9.

2. BACKGROUND ON EMAIL SYNC

Similar to desktop or laptop PCs, smartphones usually use a client application to sync email from a server. Typically email sync can be done in one of two ways: *polling* or *pushing*. In polling, a client proactively connects to a server to check for new emails. The client can be configured to automatically poll the server periodically (e.g., every 10 minutes), or the user can manually start a polling operation. Polling has two disadvantages. First, it wastes energy if a server does not have any new emails, particularly if polling is too frequent. Second, because a new email may arrive before a client polls its server, polling causes a delay in receiving the email, in the worst case as long as the polling time interval. As a result, users may find it hard to set a proper polling time interval to balance the energy cost and email receiving delay. For these reasons, push email is widely used on smartphones. Most popular email services, such as Microsoft Exchange, Gmail and Hotmail, support push email. The Gmail application on Android supports push email, but not poll email.

In push email, instead of polling from a client, emails are received by pushing from a server. Once a server receives a new email for a client, it pushes the new email to the client through a previously established TCP connection. Consequently, the client is able to immediately receive new emails as soon as they arrive. Push email not only has minimal delay but in some cases also saves energy since a client does not need to poll a server when there is no new email. As smartphones are usually behind a Network Address Translator (NAT) and a firewall, they often use a private IP address rather than a public one. As a result, an email server cannot initialize a TCP connection to a smartphone. Therefore, push email requires a persistent TCP connection between a client and a server so that the server can send new emails to the client over the persistent TCP connection. For example, Microsoft Exchange supports push email by Direct Push [1]. A client first connects to a server using TCP. To receive push emails, the client uses a long-standing HTTP POST (the protocol used by smartphones to talk to a Microsoft Exchange server is called Exchange ActiveSync [2] which is on top of HTTP) request to ask the server to respond within a time period, e.g., 15 minutes. Then the smartphone can go to sleep or standby mode. If the server has new emails within the 15 minutes, it pushes the emails to the client. Otherwise, the server will send a HTTP 200 OK message to the client who then sends another long-standing HTTP POST to the server.

Push email usually works only on cellular networks (e.g., 3G) but not on Wi-Fi. This is because the coverage of cellular networks is pervasive, and the cellular module of a smartphone can work independently from the smartphone's Operating System. When the OS is in sleep mode, the cellular module remains connected and can receive data from the cellular network. After receiving a data packet from an email server (or any other servers), the cellular module will generate an interrupt to wake up the OS. The execution of the email client is resumed, and the email can be pushed from the server. In contrast, Wi-Fi networks usually have a small amount of coverage and roaming between different Wi-Fi networks breaks the persistent connection required by push email. Furthermore, on some smartphones, the Wi-Fi interface is usually turned off when the OS is asleep, and hence the Wi-Fi interface cannot receive any data. Therefore, in this paper, we focus on studying email sync on cellular networks. Even on a cellular network, the carrier operator may tear down a TCP connection if it is idle for too long (a common timeout threshold is 10 minutes), to release the resources allocated for the TCP connection in the NAT and firewall. Email clients on smartphones handle this issue by sending a keep-alive message to the server before the timeout is reached, e.g., a PING message used in Direct Push [1].

3. PROFILING ENERGY COST OF EMAIL SYNC

In this section we measure and analyze the power performance of existing email clients on smartphones. We first describe the experimental setup and then report the results and findings.

3.1 Experimental Setup

We have studied email sync behaviors on two smartphone platforms: Windows Phone and Android. For the WP platform, we used a Samsung Omnia 7 smartphone running Windows Phone 7.5. We used the built-in email client provided by WP. For the Android platform, we used a Samsung Nexus S smartphone running Android ICS 4.0.4. We also used the built-in email client provided by Android. Both email clients support different email services.

Our study uses two popular email services: Microsoft Exchange and Google's Gmail, examining both push email and poll email. All experiments were conducted in Beijing, China, using the 3G network of China Unicom. Monsoon Power Monitor [3] was utilized to measure the power consumption of smartphones. We focused on the power consumption of the connected standby mode. That is, we measured the energy cost of receiving emails without any user interaction and the screens of the smartphones were off while receiving email. The disruption from other concurrent operations is rare in this situation, so we do not consider it in this work.

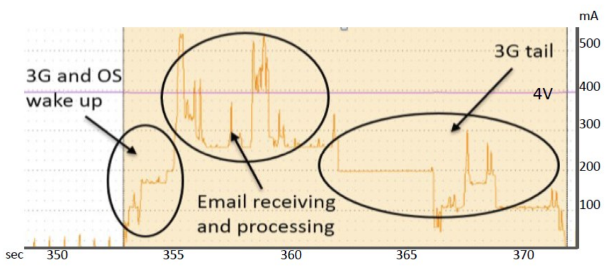


Figure 1: Power trace of receiving a push email on Android using Exchange email service

3.2 Measurement Results

3.2.1 Stages of receiving an email

We measured the power traces of receiving an email on the two platforms. Figure 1 shows the power trace of receiving a small-push email using the Exchange email service on Android. This email consisted of 2 KB random text content and the total data received was 10 KB including all headers and metadata. We can see there are several major stages in receiving this email. At the beginning, the smartphone was in sleep mode with little power consumption (about 30 mW). When the server pushed the email to the smartphone over the 3G network, the 3G interface and operating system of the smartphone woke up. We call this the *Wakeup* stage. Then the email client received and processed this email, which is called the *Receive* stage. The last stage was 3G tail time when the 3G interface stayed in a high power state but no network traffic was occurring. We call this the *Tail* stage. After the Tail stage, the 3G interface and the operating system went back to sleep. The total time duration of receiving the email is 18.42 seconds.

The 3G tail time is controlled by the cellular module using a timer. When there is no data packet transmission, the timer starts to count down. If the network remains idle after the timer expires (e.g., after 10 seconds), the 3G interface goes to sleep. If there is a data packet that is sent or received, the timer is reset. The 3G tail time is designed to reduce network latency. From Figure 1, we can see that it takes about two seconds for the 3G interface to wake up from sleep mode. During the 3G tail time, the 3G interface can keep active for continuous network traffic, avoiding wakeup latency.

We also measured the power trace in receiving the same email on Android through polling. The overall procedure is very similar to the pushing case and consists of the same three major stages (i.e., Wakeup, Receive and Tail). The difference is that, in the polling case, the operating system first wakes up itself (via a timer based on the email polling interval) and then proactively wakes up the 3G interface for querying the email server instead of being interrupted by the 3G interface passively. The total time duration of receiving the email is 18.54 seconds, similar to the push email scenario.

We further measured the power trace when receiving the same email on WP through pushing. Again it consists of the same three major stages. However, compared to the Android cases, the total time for receiving the email on the WP device is much shorter, taking only 7.24 seconds. One reason is that the WP smartphone we used has a much shorter 3G tail time than the Android smartphone. To confirm this, we conducted experiments to measure the 3G tail time on the two smartphones. We found that the WP smartphone has a 3G tail time of about 5 seconds but the Android smartphone has a 3G tail time of 10 seconds. This shows that the WP smartphone enables fast dormancy (see more details in Section 4), a technique to shorten 3G tail time, but the Android smartphone does not.

We also measured power traces of other combinations for receiving email. For example, using Gmail email service and pulling on WP, we observed the same three major stages in all the other power trace curves. We conclude that it is a common pattern of receiving an email.

Note that the stage markers in Figure 1 are just for illustration purposes, used instead of showing the exact duration of each stage. In particular the Tail stage and the Receive stage may be overlapped. For example, the 3G tail time may start immediately after all data transmissions are finished but before the data processing is done. In addition, in all the above experiments, when the email was received, the email clients on both Android and WP had already stored 200 other emails. Later in this Section we will see that the energy cost of receiving an email may depend on inbox size, particularly with WP.

3.2.2 Energy cost of receiving an email

To study the email receiving performance of different configurations, we measured the energy cost of receiving the same 10 KB email using various combinations of the two platforms (WP and Android), two email services (Microsoft Exchange and Google Gmail) and two email receiving approaches (push and poll), with an inbox of 200 messages. Table 1 shows the average results.

Our first observation was that we can see that the energy cost on Android is much higher than the one on WP. Aside from hardware differences, one reason is that the WP smartphone has a shorter 3G tail time than the Android smartphone as aforementioned. The 3G tail time is five seconds shorter on WP, leading to about 2,736 mJ energy saving. In addition, as we will show later, WP has low energy cost of email receiving when the inbox size is small.

Table 1: Average energy cost of receiving a small email with various configurations

Configuration	Energy Cost (mJ)
Android + Exchange + Push	14,976
Android + Exchange + Poll	14,674
Android + Gmail + Push	N/A
Android + Gmail + Poll	20,750
WP + Exchange + Push	5,429
WP + Exchange + Poll	5,659
WP + Gmail + Push	6,235
WP + Gmail + Poll	7,027
Android + Gmail App (Push)	12,931

Second, we could see that push email and poll email have very similar energy costs given the same email service and platform. On Android, the difference between push and poll is only 2% for the Exchange service, while this difference is less than 4% on WP. For Gmail, the difference on WP is 13%. This is reasonable because push email and poll email are fundamentally very similar: they receive the same emails and do the same data processing work. On Android, the default email client does not support push email for Gmail service and thus we do not have the number in Table 1.

We also found that the energy costs of two email services are similar on the same platform, although the Exchange service always consumes less. For example, polling-based Gmail on Android takes 20,750mJ energy, but Exchange service with polling consumes 14,674 mJ. On WP, push-based Gmail service takes 6,235mJ energy but the same service of Exchange requires 5,429 mJ energy. Despite this, different email services might be different in implementation details; they are expected to have similar patterns due to the nature of the email service. In fact, Microsoft Exchange ActiveSync (EAS) protocol has been widely used for the synchronization of emails, contacts, calendar, tasks and notes from a messaging server to a mobile device. Many companies, including Google [11] and Apple [12], have licensed and adopted EAS in their own email services for mail sync on mobile devices.

We also evaluated the Gmail application that is specifically designed for the Gmail service by Google on Android, which only supports push email. The last row of Table 1 shows the result. We can see that it requires less energy, compared to the default email client on Android. This suggests that Google does some Gmail specific optimizations, probably on both the client side and the server side. However, we do not know the technical details because the Gmail application is not open source.

Due to the similarity of push email and poll email, and the similarity of Exchange service and Gmail service, in the rest of this section, we focus more on Exchange push email.

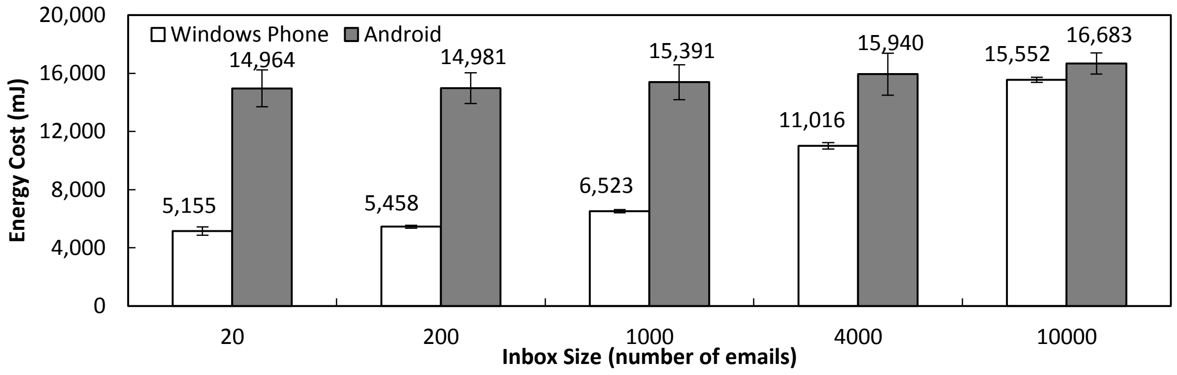


Figure 2: Energy cost of receiving a push email of 10 KB using Exchange service with different inbox sizes (five trials per experiment)

3.2.3 Energy cost vs. inbox size

We measured the energy cost of receiving the same email in different inbox sizes. Inbox size is measured in terms of the number of emails already received in the inbox on a smartphone. We used the same small email of 10 KB that we used in the aforementioned experiments.

Figure 2 shows the results using Exchange push email. Interestingly, we can see that the energy cost of receiving the same email on WP highly depends on the inbox size. When the inbox size is small, the energy cost is small and increases with the inbox size significantly. For example, the energy cost when the inbox has 10,000 emails is more than three times the energy cost when the inbox is as small as 20 emails. This is probably found by updating the metadata of the inbox database. For example, to support "conversation view", the email client needs to group all the emails of a conversation thread together.

Compared to WP, Android has a much flatter energy cost in different inbox sizes. However, when the inbox size is small, the energy cost on Android is much higher than WP. For example, when the inbox size is 20 emails, Android consumes 190% more energy than WP. Even if we exclude the extra energy cost (2,736 mJ) of the long 3G tail time on Android, Android still needs 137% more energy than WP. For large inbox size like 10,000 emails, Android and WP have similar energy cost, with a small difference of 7%.

One may argue that 10,000 emails are too many for smartphones as many users only download a small part of their whole inbox onto their smartphones, e.g., only the recent emails in the last few weeks. However, with an informal survey, we found that some people do download a large number of emails or even all their emails onto their smartphones. The main reason is that they can easily search emails on smartphones, particularly when their smartphones do not have a data connection. Furthermore, email clients on smartphones usually store only email headers and limited text

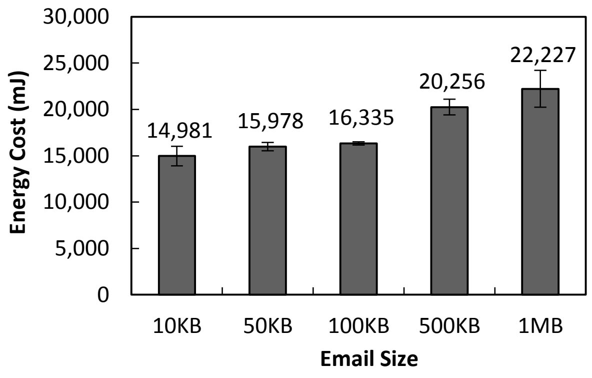


Figure 3: Energy cost of receiving an email with various email sizes (five trials per experiment)

content of email bodies (e.g., up to 5 KB bytes text). By default, attachments or embedded images are excluded. Thus, today's smartphones have enough storage space to store a large number of emails.

3.2.4 Energy cost vs. email size

We measured the energy cost of receiving an email with different email sizes. As previously mentioned, smartphone email clients usually download the email header and a limited amount of the email body for a new email. Users cannot specify the amount of data to be downloaded in both built-in email clients on the two platforms. Based on our observation, the email client on WP receives very limited email data, about only 10 KB including all email headers. The email client on Android can receive up-to several hundred KB of data (see more details in Section 3.2.6), more suitable for testing. Consequently, we only measured the energy cost of receiving an email with different sizes on Android.

Figure 3 shows the results with an inbox of 200 emails. The email size is defined as the total received data size if the email is completely received, including email content, headers and metadata. We decided the size of an email using Outlook

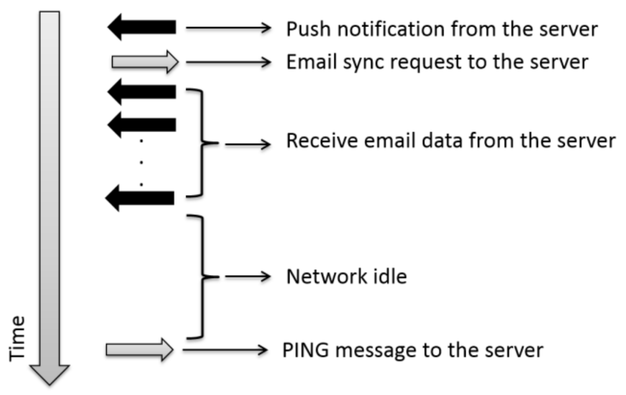

Push notification from the server
Email sync request to the server
Receive email data from the server
Network idle
PING message to the server
Time

Figure 4: Network packets transmitted in receiving a push email using Exchange service on WP

email client on a PC. Outlook email client can download all the email data and show the received data size. We can see that the energy cost increases as the email size gets larger but the difference is small, not proportional to the difference in email sizes. For example, when the email size is changed from 11KB to 107KB, an increase of 873%, the increase in energy cost is only 31%. This is for two reasons. First, the email size mainly contributes to the energy cost of receiving and processing the data, which is only a small part of the total energy consumption. Second, for large emails the Android email client does not receive all the email data. For instance, an email with a size of 500 KB and an email with a size of 1,000 KB have similar amounts of data written to the flash storage (336 KB vs. 340 KB).

3.2.5 Network activities in receiving an email

To find out what happened behind the energy cost, we profiled the network activities of the email clients when receiving an email. To do that, we captured the network data packets received and sent by the email clients. On Android, we used Tcpdump [4] to save all the network packets for offline analysis. For WP, we did not have a tool to capture packets. Thus, instead of using 3G, we conducted experiments on receiving emails over Wi-Fi. To make push email work over Wi-Fi, we kept the smartphone on at all times so that it could receive emails pushed from the email server. Then we used Wireshark [5] on a laptop to capture all network packets of the smartphone transmitted in the air.

Figure 4 shows the packets (excluding TCP ACKs) transmitted in receiving the small email sized 10KB on WP pushed from the Exchange server. Basically, the email client first received notification of the new incoming email from the server over the previously established TCP connection. Then it started to fetch the email from the server and processed it. After that, the client sent out a PING message packet to the server to wait for the next incoming new email.

We can see that there is a network-idle time period between the PING packet and the prior burst data transmission. During this time period, the email client is processing the received email. We found that this processing time increases with inbox size. As shown in Table 2, with an inbox size of 200 emails, it was 0.5 seconds but increased to 10 seconds when the inbox had 10,000 emails.

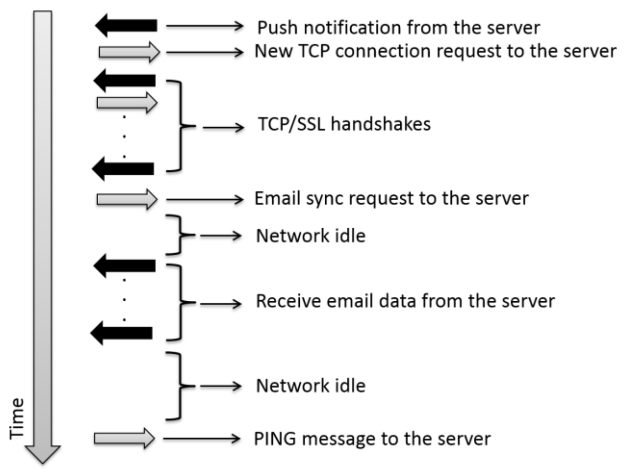

Push notification from the server
New TCP connection request to the server
TCP/SSL handshakes
Email sync request to the server
Network idle
Receive email data from the server
Network idle
PING message to the server
Time

Figure 5: Network packets transmitted in receiving a push email using Exchange service (the same one used for Fig. 4) on Android

Figure 5 shows the network packets transmitted on Android when receiving the same push email from the same Exchange server. Compared to WP, the network activities on Android are different as follows. First, instead of using the same TCP connection to fetch a new email, after receiving a notification over the existing TCP connection, the Android email client establishes a new TCP connection to the email server to receive the new email. Doing so introduces some overhead on TCP and SSL handshakes between the client and the server, where 12data packets (6.9KB in total) are transmitted. Second, on Android there are two network-idle time periods. One is between the email sync request sent from the client to the server and the server's reply. The other one is after receiving the burst event data and before the final PING message sent from the client to the server. However, unlike WP, the network-idle time periods do not depend on inbox size. Table 2 also shows the total network-idle time on Android. It bears no obvious relationship to the inbox size, ranging from 3 seconds to 4.6 seconds (3.6 seconds on average).

3.2.6 Storage activities in receiving an email

Besides the network activities, we also profiled the storage activities in receiving an email. For Android, we developed a tool to intercept the storage access APIs (e.g., file reading and writing) to capture all the flash-storage operations of the email client. For WP, we couldn't develop such a tool due to the limited programmability offered by Windows Phone. Therefore, we only conducted the flash-storage experiments on Android.

Table 3 shows the total amount of data written to flash storage for different incoming email sizes. We can see that the

Table 2: Network idle time during email receiving

Inbox size (number of emails)	Network idle time (seconds)	
	WP	Android
20	0.4	3.5
200	0.5	4.6
1,000	0.9	3.0
4,000	3.5	3.4
10,000	10.0	3.5

Table 3: Data size written to flash in email receiving

Email size (KB)	10	50	100	500	1,000
Data size (KB)	139	156	188	336	340

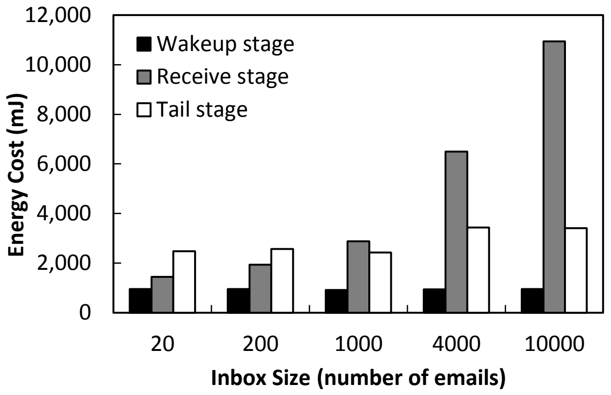
amount of data written to flash storage increases with email size. This is because the email client needs to write the received email data into the inbox database. Thus, more data were written to flash storage for larger email. However, when the email is very large, the data amount written to flash storage does not increase any more. This is because the Android email client limits the maximum data received for emails. Unfortunately it does not provide an option for users to configure such a behavior.

The results show that the Android email client immediately writes received email data onto flash storage. All flash operations in receiving an email were distributed in a time period of one to two seconds, each writing a small amount of data. As we will show in Section 4.3, small flash writes are time and energy expensive and should be avoided in receiving emails.

3.2.7 *Energy distribution*

Finally we studied how the total energy cost of receiving an email is distributed in the three stages of the power curves: *Wakeup* stage for the 3G interface and operating system to wake up, *Receive* stage for the email receiving and processing, and *Tail* stage for the 3G tail time. We denote the time duration of the Wakeup stage as the time period between the time when the smartphone wakes up and the time before receiving the notification packet from the server. We define the time duration of the Tail stage as the time period between the time when the last packet is transmitted and the time when the smartphone goes to sleep again. We treat the rest of time as the time duration of the Receive stage.

Figure 6 shows the results. We can see that the Wakeup stage takes stable and fixed energy costs on both WP (907-936 mJ) and Android (979-1,022 mJ) no matter how big the inbox is. The energy cost of the Tail stage is also quite stable on the two platforms. On WP, the Tail stage takes 2,419-3,427 mJ (2,866 mJ on average) of energy but on Android it takes 5,832-6,307 mJ (5,962 mJ on average) of energy, due to the difference of 3G tail time on the two platforms. Different from the Wakeup stage and the Tail stage, the energy cost of Receive stage depends on the inbox size. Particularly for WP, the energy cost increases significantly with the inbox size,



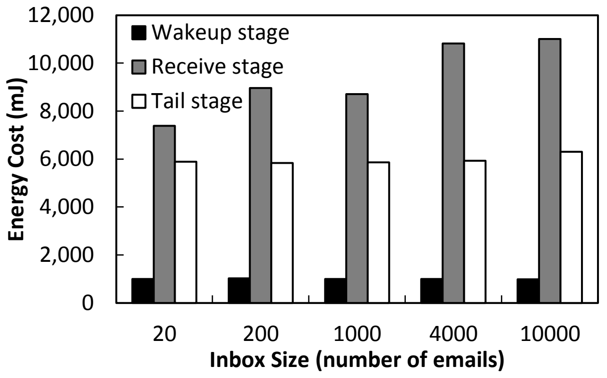


Figure 6: Energy distribution among the three stages in receiving an email with different inbox sizes. Top: on WP. Bottom: on Android.

from only 144mJwhen the inbox size is 20 to 10,944 mJ when the inbox size is 10,000. For Android, the numbers are relatively flat: 7,387 mJ for inbox of 20 and 11,002 mJ for inbox of 10,000.

In addition, we can see that the Receive stage and Tail stage take more energy than the Wakeup stage. The Receive stage takes 30% - 71% (49% on average) of the total energy cost on WP, and takes 52% - 61% (57% on average) of the total energy cost on Android. For the Tail stage, it takes 22% - 51% (38% on average) of the total energy on WP, and takes 33% - 42% (37% on average) of the total energy cost on Android. Therefore, to reduce the energy cost of receiving emails, we should focus on optimizing these two stages.

3.3 Summary of Findings

Based on above measurement results in Section 3.2, we can see that the existing smartphone email clients are not energy efficient in following aspects.

First, the 3G tail time takes a large part of the total time of receiving an email. The 3G tail time is designed to reduce the network latency for burst data transmissions. However, the inter-arrival time of incoming events is not short in the connected standby mode. Occasional transmissions make the 3G tail time unnecessary in most cases. Therefore, the energy is wasted on the 3G interface.

Second, the data transmission and processing are coupled together, leading to more wasted energy. Receiving an email is composed of the following steps: first is communicating with the server to receive some data, processing the received data locally, and then communicating with the server again. The second step of the communication resets the 3G tail timer, causing the 3G interface on for longer time and thus wasting energy.

Third, the email clients write data onto the flash storage when receiving an email. As we will show in Section 4.3, flash operations are energy-expensive and should be batched together to save energy.

Fourth, when the inbox size is large, receiving an email costs more energy than small inbox case. It is desirable to reduce the energy cost for a large inbox size.

Finally, on Android, the email client always initializes a new TCP connection to the server to receive an email. Doing so wastes energy due to the duplicated TCP and SSL handshakes.

Next in Section 4 we propose techniques to address above energy inefficiency and reduce the energy cost of receiving email. Those techniques can also be used as general design guidelines to improve power performance of other applications on smartphones.

4. REDUCING ENERGY COST OF EMAIL SYNC

From Figure 1, we can see that the baseline power, when the system is active but idle (e.g., during the 3G tail time), can reach over 200 mW. This means that once the 3G interface and the OS wake up, the smartphone will consume a large amount of energy even without doing any tasks. Therefore, to reduce the energy cost of receiving an email, one effective way is to shorten the total time period of email receiving as much as possible. In this section we show how we achieve it with five techniques, each of them addressing one energy inefficiency issue we observed in Section 3.

4.1 Reducing 3G Tail Time

The 3G tail time causes a lot of energy to be wasted in receiving emails. After a new email is received and processed, the 3G interface enters the tail state in which the whole smartphone does nothing but consume energy. While the 3G tail is designed to save energy and reduce latency in continuous network data transmissions, it is not suitable for the connected standby mode. In the connected standby mode, the events received by a smartphone are pretty sparse, often arriving at a time period much longer than 3G tail time. Thus, it is likely that the 3G tail time cannot cover multiple events. To verify it, we measured the inter-arrival time of the emails of four researchers at Microsoft Research Asia. In total 31,303 emails, only 1.3% emails come within a ten-second time frame of previous one. Therefore, a 3G tail time of ten seconds rarely covers more than one incoming emails and thus wastes a considerable amount of energy.

The problem of wasted energy caused by the 3G tail time has already been identified recently. To solve the problem, a technique called fast dormancy [7] has been proposed to force the 3G interface to quickly sleep faster than before, e.g., five seconds rather than ten seconds. However, rapid dormancy increases the signaling overhead of cellular networks if the sleep timers are too short. In fact, in the early days of fast dormancy, some popular smartphones using aggressive timers have led to severe signaling channel congestions [25]. Later the network-controlled fast dormancy was adopted by 3GPP in Release 8 [16] to reduce the signaling overhead caused by the fast dormancy. Researchers have proposed to adaptively use the fast dormancy based on application traffic patterns [8, 9, 14, 18]. For example, in [8] the authors proposed a tail optimization protocol based on fast dormancy. In [9] a system has been built to predict the end of communication to invoke the fast dormancy without increasing the network signaling load.

We advocate for the fast dormancy to be used in connected standby. As shown by our measurement results in Section 3, the WP smartphone we used enables fast dormancy and thus has a shorter 3G tail time than the Android smartphone. We also agree with the authors in [8] and [9] for adaptive 3G tail time based on application traffic patterns.

In particular, because that network events are very sparse in the connected standby, 3G tail time still wastes energy even if fast dormancy is enabled. In an ideal case, the 3G interface should go to sleep immediately after the event receiving is finished. However, to avoid interference with other applications, individual applications must not directly control the 3G tail time. Instead, we propose that the OS should provide a global service that collects the network usage information of all the applications running in connected standby, decides the shortest length of 3G tail time, and collaborates with the cellular network to put the 3G interface into sleep mode as quickly as possible, minimizing the energy waste of 3G tail time. Due to the long inter-arrival time of network events in the connected standby, the signaling overhead is small.

4.2 Decoupling Data Transmission from Data Processing

Due to the 3G tail effect, data transmission should be decoupled from data processing to save energy. In an ideal case, an email client should first finish all network communication with a server and then process the received data, so that the 3G interface is able to go to sleep as soon as possible. In receiving a push email, there are three steps: 1) fetching the email content from the server, 2) processing the received

Table 4: Energy and time cost of databasewriting

	Energy cost per email (mJ)		Time cost per email (ms)	
	Database in memory	Database on flash	Database in memory	Database on flash
Individual writes	1	32	1	55
Batch writes	0.59	0.64	0.58	0.65

email locally (e.g., updating the inbox database), and 3) telling the server that it is ready to receive the next push email and then go to sleep. As we show in Section 3, in such a "transmission-processing-transmission" process, existing email clients do the second transmission part after the processing part is finished, which seems a natural design choice due to the sequential nature but is not energy efficient. During the data processing part, the 3G interface stays awake unnecessarily and the second data transmission resets the 3G tail timer, wasting energy. For example, assume that the data processing needs two seconds and the power is 200 mW, 400 mJ energy will be wasted. Therefore, to save energy, an email client should batch all its data transmissions together. It should first fetch new email content and immediately tell the server that it is ready to receive the next push email before waiting for the email processing to be finished.

Batching all data transmissions together may be difficult if a network protocol depends on the result of data processing. For example, if an email server requires an email client to tell whether the email processing is successful or not, the email client cannot start the second data transmission part before the data processing part is finished. We propose to solve this problem using speculative execution [6]. That is, we predict the result of the data processing part and send the predicted result to the server without waiting for the data processing part to be finished. If we find that the predicted result is wrong after the data processing part is finished, we can resync with the server. If we can correctly predict the data processing results with a high probability, we can still batch data transmissions together to reduce the 3G tail effect. To determine feasibility, we conducted an experiment to measure the failure rate of receiving and processing 10,000 emails. We found that all the 10,000 emails were successfully processed without any error. This indicates that the email receiving is reliable and we can correctly predict the email processing results with a high probability. Therefore, it is possible to leverage speculative execution to save energy. Furthermore, for Microsoft Exchange and Gmail services on smartphones, they do not depend on the email processing results. Thus, we can easily batch data transmissions and decouple them from data processing.

4.3 In-Memory Data Processing

Writing data to flash storage is slower than writing data to memory, particularly for small, random data writes [13, 14, 19]. We conducted experiments to measure the performance difference of the flash storage and memory. We used a Nexus S smartphone running Android 4.0.4. We inserted an email of 270 bytes into a SQLite database for 1,000 times and measured averaged energy and time cost of inserting one email. We used the SQLite database because the Gmail client and the default email client on Android use SQLite to store emails. The SQLite library provides two types of APIs to write data into a database: individual writes and batching multiple writes together as a transaction. Thus, we measured the performance of writing the 1,000 emails one by one and batching the 1,000 email writes together.

Table 4 shows the results. For individual writing, when the entire database was loaded in memory, the average energy and time cost per email was just 1 mJ and 1 ms. When the database was stored on the flash storage, the corresponding energy and time costs increased to 32 mJ and 55 ms. For batching writes, when the database was loaded in memory, the average energy and time costs were 0.59 mJ and 0.58 ms. When the database was stored on flash storage, the corresponding figures were 0.64 mJ and 0.65 ms. Those results show that for small writes, flash storage costs much more energy (32 times) and takes much more time (55 times) than memory. However, for writing a large amount of data together, the flash storage and memory have similar performance in terms of the energy and time costs.

Therefore, to reduce the energy and time cost in email receiving, an email client should not issue many small flash writes. Instead, it should batch multiple small writes together and commit them to flash in a batch. As mentioned before, email clients on smartphones only download email headers and up to several kilobytes of email bodies. Thus, the data received for a new email is small. However, as we show in Section 3, existing email clients write received data of every email to the flash storage immediately, which is not energy efficient. Therefore, when an email client receives a new email, it should cache the received data in memory rather than writing them to flash immediately. Once the client has received a sufficient number of emails (e.g., 10 emails) or the cached data are larger than a threshold, it can flush all the data to flash storage together. Furthermore, the client may also piggyback data writes with other activities. For example, when the user turns on the smartphone to check emails or use other applications, the client can write its cached data to flash. As those flash writes share the same baseline power with other activities of the user, they will not introduce much extra energy or time overhead.

Delaying flash writes may cause data loss if the email client or the smartphone crashes before the cached data are written to flash storage. However, modern smartphone operating systems including Windows Phone, Android and iOS all run each application in a separate protection domain (i.e., the so called “application based security model”). Failures of one application will not affect other running applications. As a result, today’s smartphones are more reliable and crash less than before. Even when running out of battery, the operating system will terminate applications gracefully, to give them chance to save data onto flash. Smartphone email clients are also pretty reliable. We measured the default email client on Android by receiving 10,000 emails and did not observe any failure or crash. In addition, even if the email client crashes before writing the cached data to flash storage, it can re-sync the emails with the server. Therefore, in-memory data processing is able to reduce the energy cost of email receiving.

4.4 Reusing Existing Network Connections

In receiving emails, email clients should reuse existing network connections rather than making new ones. In particular, for the push email, because an email client already maintains a persistent network connection with a server to receive notification of new incoming emails, it should receive a new email over the continuous network connection to save energy. Even for a poll email, when an email client does a second polling before the network connection of the last polling times out, it should also reuse the previous network connection.

While it may be easy and natural to make a new network connection to receive every new email, the energy cost may not be negligible. As most email services require a secure network connection, making a new network connection includes not only TCP layer handshakes but also SSL layer handshakes to negotiate the encryption method and exchange security keys. As shown in our experiments on Android in Section 3, the overhead of making a secure TCP connection is not negligible. Without transmitting any application data packets, it consumed 1,757 mJ energy to make a TCP/SSL connection. In total 6.9 KB data were transmitted between the client and server. WP is more energy efficient. It always reuses the same TCP connection to receive new emails.

4.5 Data Structure Partitioning

One interesting finding on WP is that the energy cost of receiving the same email increases significantly when the inbox size is large. On Android, the energy cost also increases with inbox size but the increase is not as significant as WP. The possible reason is that when storing a new email into the database of an inbox, it takes more time to update the metadata. For example, the email client needs to search the whole inbox to associate the new email with the right conversation thread. The cost of doing so depends on the inbox size. In addition, when the inbox is too large to be loaded into the memory of the email client, searching the inbox takes more energy and time due to the frequent flash operations.

To solve this problem and reduce the energy cost of receiving an email, we propose partitioning a large inbox into two parts:

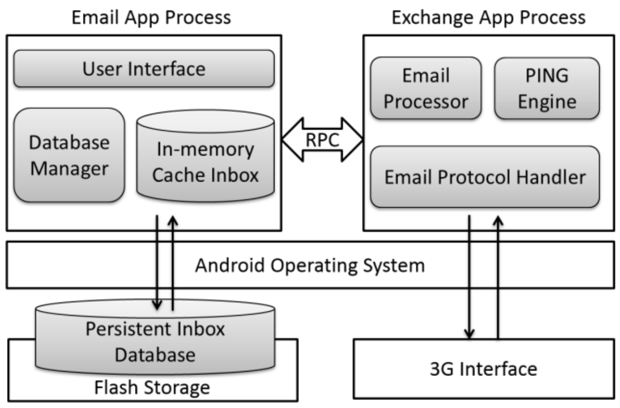


Figure 7: Implementation architecture

one small inbox with recently received emails (e.g., emails received in last two weeks) and one large inbox containing all remaining emails. The email client only uses the small inbox to handling email receiving. That is, when receiving an email, the client only inserts it into the small inbox and updates the metadata without using the large inbox. Thus, the energy cost of receiving emails is reduced even if a smartphone stores many emails. This approach is based on a key observation: most of the time users only need to check recent emails on their smartphones. Therefore, it is not necessary to touch all the emails already stored in a smartphone in receiving new emails. When a user does need to access all the emails, e.g., in searching the whole inbox, the client can search both the small inbox and the large one to return the combined results. As searching a whole inbox happens much more rarely than receiving emails, inbox partitioning is able to save energy.

5. IMPLEMENTATION

Implementing the techniques proposed in Section 4 only requires modifications on the email client side. Since the Gmail application on Android and the built-in email client on WP are not open source, implementation is done on the built-in email client of Android, which is accessible and widely used. The source code we used is vanilla Android 4.0.4 with kernel 3.0.8.

Figure 7 illustrates the architecture of our implementation, focusing on email syncing and processing. On Android the built-in email client consists of two parts, each running in a separate process. The first part is the network communication part which implements the email protocol and handles all the communication details with the email server to receive and send emails. For Exchange email service, this part runs as an app named “Exchange”. It runs in the background without a UI. The other part is the data processing which processes received emails and provides user interfaces to handle all the interaction details with users including reading emails and

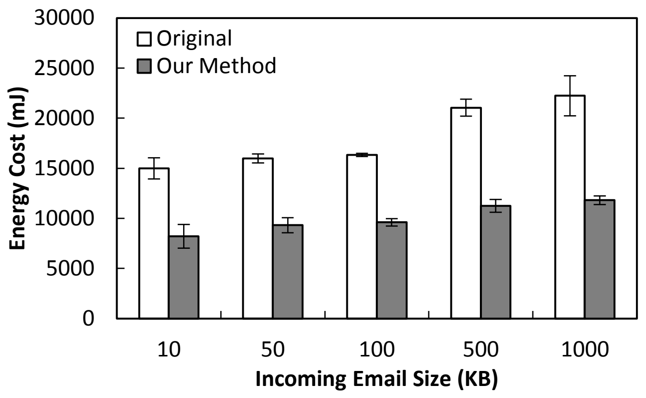


Figure 8: Energy saving with different email sizes

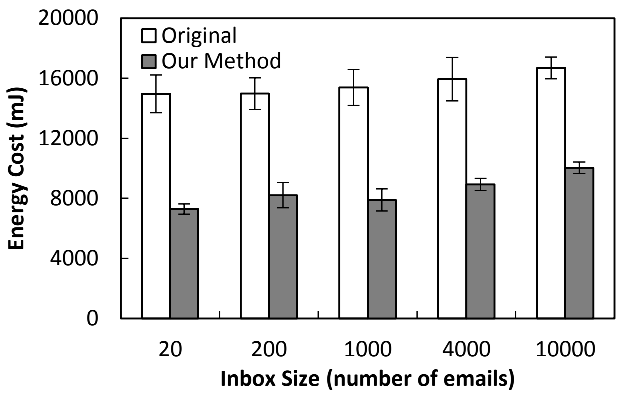


Figure 9: Energy saving with different inbox sizes

composing emails to send out. It also deals with email storage, saving emails to and loading emails from the SQLite database. This part runs as an app named "Email". The Email app process and the Exchange app process exchange data through Remote Procedure Calls (RPCs).

To decouple data transmission from data processing, we revised the Email Protocol Handler in the Exchange application to do all data transmissions without waiting for the Email Processor to finish processing a received email. Specifically, PING messages are sent via the PING Engine in Figure 7. In the original client, the PING Engine checks up in a two-second interval whether the Email Protocol Handler allows it to resume PING messaging after an email-receiving event ends. So there is a delay between the completion of receiving an email and the start of PING. The energy is wasted because the cellular module has to delay its sleep mode. We improved the signaling between the Email Protocol Hander and the PING Engine so that a PING message can be sent out immediately. Thus, all data transmissions are batched together to save energy. We also revised the Email Processor to report an error if anything is wrong in processing an email so that the Email Protocol Handler can re-sync with the email server. In addition, we revised the Email Protocol Handler to receive emails by re-using the existing TCP connection rather than making a new one, avoiding energy waste of TCP/SSL handshakes.

In the Email application, we added a new Database Manager, loading a small cache inbox into the memory to implement in-memory email receiving. On the flash storage, all the emails are stored in a big database. The Database Manager manages the data sync between the in-memory cache inbox and the persistent one on the flash storage. When the user interacts with the smartphone or we receive more than a customized number of emails, the Database Manager commits the new email data onto the flash storage.

Android does not provide an API for fast dormancy and we cannot control the 3G tail time on the fly. We have managed

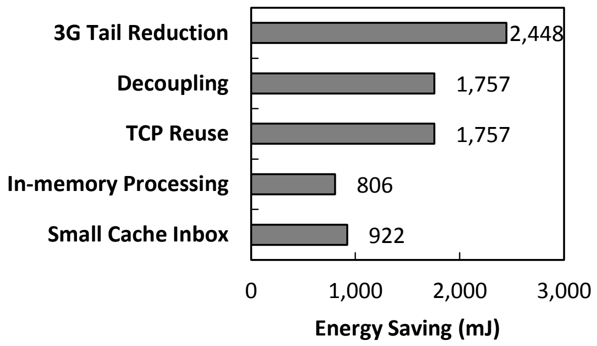


Figure 10: Anatomy of total energy saving

to enable fast dormancy on the Nexus S smartphone by flashing a new baseband firmware.

6. EVALUATION

We evaluate our implementation by measuring the total energy saved in receiving an email and the individual ones contributed by each technique we proposed and implemented. For all the experiments, we used Exchange push email service with the original built-in email client provided by Android and our revised one on a Nexus S smartphone. For each experiment, we repeated for the procedure five times and report the average results with standard deviations.

Total energy saving. Figure 8 shows the total energy saving of our revised email client, compared to the original one when the inbox size is 200 emails but the new incoming email has different sizes. On average the total energy saving is 44.3%, with a narrow range from 41.2% to 46.9%. These results demonstrate that our proposed techniques are able to significantly reduce the energy cost of email receiving. The energy savings mainly come from the shortened email receiving time. On average our revised email client reduces the time period of receiving the emails by 47.6%, compared to the original one.

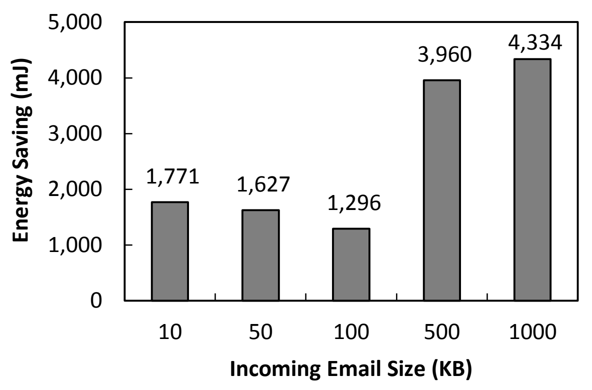


Figure 11: Energy saving of decoupling data transmission from data processing in different email sizes

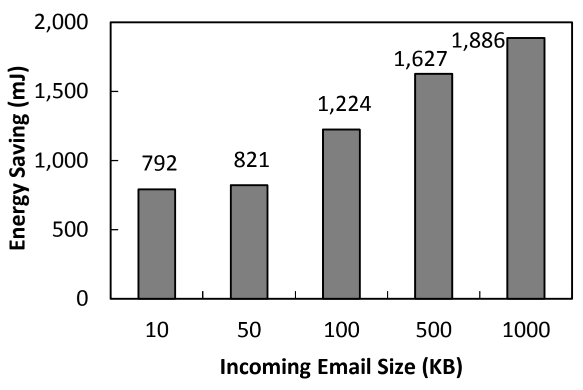


Figure 12: Energy saving of in-memory data processing in different email sizes

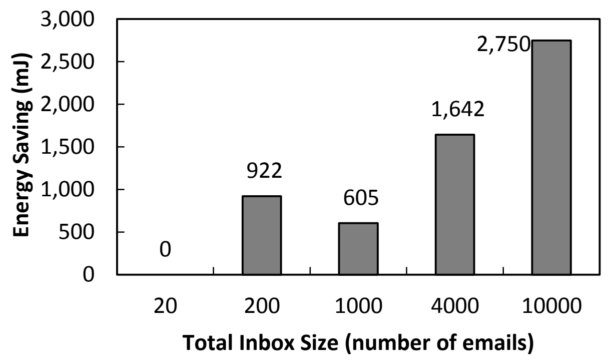


Figure 13: Energy saving of using a small cache inbox for email receiving with different total inbox sizes

Figure 9 shows the total energy savings of our revised email client in receiving a small email of 10 KB with different inbox sizes. Similar to Figure 8, we can see that our revised email client is able to effectively reduce the total energy cost of email receiving no matter how many emails the inbox has. The average energy saving is 45.8%, also with a small range from 39.9% to 51.3%. Compared to Figure 8, the average energy saving is higher because when the email is large, the received data volume becomes large and more energy will be saved by our proposed techniques, as shown later in this Section.

Energy saving of each technique. Figure 10 shows the anatomy of the total energy saving in receiving the 10 KB email with an inbox of 200 emails. For the total energy savings of 7,690 mJ, the largest part comes from reducing 3G tail time which saves 2,448 mJ (32%) energy. Decoupling data transmission from data processing part and reusing TCP connections also save a significant energy cost, each with 1,757 mJ (23%). The in-memory processing part and using small cache inbox part reduce relatively less energy, saving 806 mJ (10%) and 922 mJ (12%)energy, respectively.

Reducing the 3G tail time and reusing the TCP connections have fixed energy savings, independent from incoming email size and inbox size. However, the energy saving of other techniques may depend on the email size or inbox size. Figure 11 shows how much energy can be saved by decoupling data transmission from data processing for different email sizes. We can see that generally the energy saving increases when the email size becomes large. The decoupling technique tries to batch all data transmissions together. When the email is large, more data is transmitted over the network and thus the energy saved by batching becomes large.

Similarly, Figure 12 shows how much energy can be saved by in-memory processing for different email sizes. We can see that the energy saving is high for large emails. This is because the processing cost (e.g., writing operation) heavily depends on email size. Thus, when the email size is large, processing the email entirely with memory operations will save more energy.

Figure 13 shows the benefit of using a small cache inbox for email receiving when the total inbox size is different. The size of the small cache inbox we used is 20. Thus, the benefit for the inbox size of 20 in Figure 13 is zero. For a larger inbox size, we can see that the extra energy saving caused by using the small cache inbox increase as the total inbox size becomes large. For the total inbox size of 10,000 emails, the extra energy saving is 2,750 mJ. These results demonstrate the effectiveness of using the small cache inbox.

Note that in all the above experiments, we measured the energy cost of receiving individual emails without counting the energy saving of batch writing multiple emails onto flash. In our implementation we batch 20 emails together for writing their data onto flash. By doing so, we can further save 634 mJ energy per email. On average this increases the total energy saving of receiving a 10 KB email to 49.9% in different inbox sizes.

Furthermore, we also calculate how much extra energy can be saved if we can put 3G interface into sleep immediately after the end of receiving an email. Doing so we can further save 2,448 mJ of energy. For receiving a 10 KB email with an inbox size of 200, this will increase the total energy saving by 77.9%.

7. EVENT RECEIVING IN OTHER APPLICATIONS

Despite this paper focusing only on email receiving, the techniques we proposed may also be applied to other applications. Example applications include various social network applications such as Facebook and Twitter, locations based applications which receive notifications upon location changes, and many other applications which receive push notifications when new updates or new in-application content are available. Similar to email, when in standby mode, those applications maintain their own persistent connection or use a separate push notification service like Google client notification service [15] to keep connected to a server for receiving various events or poll changes from a server. Such event receiving is very similar to email receiving and follows the "transmission-processing-transmission" pattern. Each event receiving also wakes up the 3G interface and a smartphone's entire operating system for a small amount of data transmission and processing. Due to the similarity between receiving emails and receiving other events, those applications may also have the issues with energy inefficiency we found in email receiving, such as sparse data transmissions resetting 3G tail timer, frequent small flash writes, and making new network connections unnecessarily. Therefore, our proposed techniques can be used as general design guidelines for those applications rather than specific techniques designed for email receiving only.

We plan to investigate more applications to further study how the techniques we proposed in this paper may be used to reduce the energy cost of receiving events in connected standby. In addition, recent tablet devices and other types of mobile devices have started to function like smartphones, keeping connected in standby mode to receive various network events including emails. For example, Windows 8 introduces a new connected standby power mode [17] to provide such support. We also plan to study the power performance of applications on those devices and look for potential improvements.

8. RELATED WORK

System power management. Power management is important to mobile devices including smartphones due to the limited power supply of batteries. All smartphone operating systems come with components to leverage hardware capabilities and manage system activities to save power, e.g., adjusting screen backlight levels, running CPU at low frequency, putting network interfaces into sleep or killing power-hungry applications in low battery mode. Various techniques have been proposed to reduce the power consumption of each key component of a smartphone such as the screen [22], CPU [24] and network communication [20, 23]. However, even with good system-level and component-level power management, applications that are not well designed can still cause high energy consumption.

Finding energy bugs of applications. Recently researchers have started to study how to improve the power performance of mobile applications by finding energy bugs. Energy bugs are not real program bugs causing failures but they lead to unnecessarily high energy consumption. For example, in [10] the authors proposed techniques for automatically finding non-sleep energy bugs in applications. However, such a tool depends on pre-configured program patterns to find energy problems, e.g., requiring a Wakelock to prevent the system from sleeping without releasing it, and only works for serious non-sleep bugs. Authors in [21] built tools for developers to estimate the energy consumption of their applications and find potential energy inefficiency issues. Our work focuses on finding energy inefficiency when receiving email by measuring the energy performance of existing email clients. The techniques we proposed are complimentary with existing work and can be used as general design guidelines for other applications.

Fast dormancy. The problem of the cellular data network tail has drawn a lot of attention. Smartphone manufacturers have developed their own and inconsistent ways to reduce the energy cost of a cellular tail, which results in the standard approach called network-controlled fast dormancy in 3GPP [7, 8]. Researcher also proposed schemes to reduce the power consumption of applications by using fast dormancy adaptively [8, 9]. We believe that fast dormancy should be used in connected standby. However, even with fast dormancy, there is still wasted energy in the event receiving during the connected standby due to very sparse network activities. We propose that the OS should provide a global service to collect the network usage information of all the applications running in the connected standby. When all the applications finish their event receiving, the service works with the cellular network to immediately put the 3G interface into sleep mode and thus minimizes the energy waste of the 3G tail time in connected standby.

9. CONCLUSIONS

In this paper we have studied the power performance of email sync on the Windows Phone and Android operating systems. We conducted experiments to measure the energy cost of email receiving in existing email clients, using both push and poll emails, with different email services, email sizes and inbox sizes. Together with our analysis of the network and flash storage activities, the measured results indicate that existing email clients are not energy-efficient in several aspects. Besides 3G tail time, the main source of inefficiency stems from the coupling of the data transmission and data processing, which is a bad design choice in connected standby mode. Frequent storage access, making new TCP connections to receive new emails and updating a large inbox are further components that increase the time duration of email sync and lead to energy waste.

Based on the experiment, we advocate that the fast dormancy should be used in the connected standby and propose other techniques to address the energy inefficiency of existing email clients, including the decoupling of data transmission from data processing, in-memory processing, reusing existing network connections and using a small cache inbox to handle the email sync. We have implemented these techniques and evaluation results show that average energy savings reach 49.9%. If we can put the 3G interface into sleep mode immediately after receiving an email, the total energy savings can be increased to 77.9%.

ACKNOWLEDGEMENTS

We thank our shepherd, Mahadev Satyanarayanan, and anonymous reviewers for their valuable comments and insightful feedback. This project was supported in part by US National Science Foundation grants CNS-1117412 and CA-REER Award CNS-0747108.

REFERENCES

[1] Microsoft Direct Push.http://technet.microsoft.com/en-us/library/cc539119.aspx.

[2] Microsoft ActiveSync Command Reference Protocol Specification.http://msdn.microsoft.com/en-us/library/dd299441(v=exchg.80).aspx.

[3] Monsoon Power Monitor. http://www.msoon.com/LabEquipment/PowerMonitor/.

[4] Tcpdump. http://www.tcpdump.org/.

[5] Wireshark.http://www.wireshark.org/.

[6] B. W. Lampson. Lazy and Speculative Execution, talk at *OPODIS*, 2006.

[7] UE "Fast Dormancy" behavior. 3GPP discussion and decision note RP-090960, 2009.

[8] F. Qian, Z. Wang, A. Gerber, Z. M. Mao, S. Sen and O. Spatscheck. TOP: Tail Optimization Protocol for Cellular Radio Resource Allocation, In *ICNP*, 2010.

[9] P. K. Athivarapu, R. Bhagwan, S. Guha, V. Navda, R. Ramjee, D Arora, V. N. Padmanabhan and G. Varghese. RadioJockey: Mining Program Execution to Optimize Cellular Radio Usage. In *Mobicom*, 2012.

[10] A. Pathak, A. Jindal, Y. C. Hu and S. P. Midkiff. What is keeping my phone awake? Characterizing and Detecting No-Sleep Energy Bugs in Smartphone Apps. In *MobiSys*, 2012.

[11] Google adopts Microsoft sync protocol, http://www.windowsfordevices.com/c/a/News/Google-adopts-Microsoft-sync-protocol/.

[12] Apple finally acknowledges iPhone's Exchange support, http://www.zdnet.com/blog/microsoft/apple-finally-acknowledges-iphones-exchange-support/1246.

[13] A. Carroll and G. Heiser. An Analysis of Power Consumption in a Smartphone. In *Usenix ATC*, 2010.

[14] S. Deng and H. Balakrishnan. Traffic-Aware Techniques to Reduce 3G/LTE Energy Consumption. In *CoNEXT*, 2012.

[15] A. Adya, G. Cooper, D. Myers and M. Piatek. Thialfi: A Client Notification Service for Internet-Scale Applications. In *SOSP*, 2011.

[16] 3GPP Release 8, http://www.3gpp.org/Release-8.

[17] P. Stemen and S. Berard. Understanding Connected Standby.*Microsoft Build* conference talk, 2011.

[18] F. Qian, Z. Wang, A. Gerber, Z. M. Mao, S. Sen and O. Spatscheck. Profiling Resource Usage for Mobile Applications: A Cross-layer Approach. In *Mobisys*, 2011.

[19] H. Kim, N. Agrawal and C. Ungureanu. Revisiting Storage for Smartphones. In *FAST*, 2012.

[20] H. Han, Y. Liu, G. Shen, Y. Zhang and Q. Li. DozyAP: Power-Efficient Wi-Fi Tethering. In *MobiSys*, 2012.

[21] R. Mittal, A. Kansal, and R. Chandra. Empoweringdevelopers to estimate app energy consumption. In*MobiCom*, 2012.

[22] M. Dong and L. Zhong. Chameleon: a color-adaptive-web browser for mobile oled displays. In*MobiSys*, 2011.

[23] F. Dogar, P. Steenkiste and K. Papagiannaki. Catnap: Exploit High Bandwidth Wireless Interfaces to Save Energy for Mobile Devices. In *Mobisys*, 2010.

[24] M. Ra, B. Priyantha, A. Kansal and J. Liu. Improving Energy Efficiency of Personal Sensing Applications with Heterogeneous Multi-Processors. In *UbiComp*, 2012.

[25] How smartphones are bogging down some wireless carriers. http://arstechnica.com/gadgets/2010/02/how-smartphones-are-bogging-down-some-wireless-carriers/.