Optimizing Energy of HTTP Requests in Android Applications

Ding Li and William G. J. Halfond
University of Southern California
Los Angeles, California, USA
{dingli,halfond}@usc.edu

ABSTRACT

Energy is important for mobile apps. Among all operations of mobile apps, making HTTP requests is one of the most energy consuming. However, there is not sufficient work in optimizing the energy consumption of HTTP requests in mobile apps. In our previous study, we found that making small HTTP requests was not energy efficient. Yet, we did not study how to optimize the energy of HTTP requests. In this paper, we make a preliminary study to bundle sequential HTTP requests with a proxy server. With our technique, we had a 50% energy saving for HTTP requests in two market Android apps. This result indicates that our technique is promising and we will build on the result in our future work.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: Diagnostics

General Terms

Performance

Keywords

Energy optimization, HTTP requests, Mobile systems

1. INTRODUCTION

Energy is a critical resource for battery supported devices, such as smartphones and tablets. Improving the energy efficiency of mobile apps could improve the user satisfaction of the apps and potentially improve the revenue of developers. Recently, researchers have proposed many techniques to save energy for mobile apps. However, none of these approaches focus on the optimization of HTTP request energy consumption.

Unfortunately, overlooking HTTP requests is problematic. According to our previous study [8], making HTTP requests is the most expensive type of API call in mobile applications. On average, making HTTP requests can consume more than 32% of the total non-idle state energy of an app. Thus, researchers and developers can miss a significant area to optimize energy if they omit HTTP requests.

To study energy consumption of HTTP requests, we performed a preliminary study [7] and found that making small HTTP requests is not energy efficient. However, in that preliminary study, we did not propose a method to optimize HTTP energy.

In this paper, we introduce a preliminary study on how to optimize the energy consumption of HTTP requests in mobile apps. Our approach is to combine multiple HTTP requests that will definitely be made together. By doing this, our approach could reduce the number of HTTP requests and reduce the overhead of initiating and sending each individual HTTP request. In our evaluation, we found that there was an up to 50% reduction in the HTTP energy consumption in two market Android apps when using our approach.

The structure of this paper is as follows. In Section 2, we introduce some background information about making HTTP requests and our previous results. In Section 3, we introduce our ideas of how to optimize the HTTP request energy. In Section 4, we report the results of our evaluation. Finally, in Section 5, we discuss related work and conclude in Section 6.

2. BACKGROUND AND MOTIVATION

In our previous empirical study [8], we found that making HTTP requests is one of the most energy consuming operations in Android apps. In the empirical study, we measured the energy consumption of different packages of APIs in 405 Android market apps. We found that, on average, making HTTP requests could represent 32% of the total non-idle state energy consumption of mobile apps. For certain apps, this percentage could be even higher than 60%. Compared with other operations, making HTTP requests consumes significantly more energy.

Another one of our previous studies [7] further showed that making small HTTP requests is not energy efficient. In that study, we found that downloading one byte of data consumed the same amount of energy as downloading 1,024 bytes of data through HTTP requests. Furthermore, we found that downloading 10,000 bytes of data only consumed twice the amount of energy as downloading 1,000 bytes of data. In this case, making small HTTP requests consumes more energy per each byte transmitted through the network.

This inefficiency is due to the protocol of sending HTTP requests. Making an HTTP request needs three steps: es-

DeMobile'15, August 31, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3815-8/15/08...$15.00
http://dx.doi.org/10.1145/2804345.2804351

tablishing the connection, transmitting data, and closing the connection. In the steps of establishing the connection and closing the connection, the client side needs to have a 3-way or 4-way handshake protocol, which spend energy in transmitting practically empty packets. In the step of transmitting data, energy overhead will be introduced by the headers of the HTTP request and its lower-level network protocols, such as TCP and IP. Thus, most energy of a small HTTP request will be consumed by the handshake-protocol and headers.

3. APPROACH

Our approach focuses on optimizing the HTTP energy consumption of mobile apps with sequential HTTP requests. By sequential HTTP requests, we mean the HTTP requests that are always made together, in sequence, despite the user input or execution condition of the program. One example of sequential HTTP requests is in Program 1, where the HTTP requests at line 6, line 10, and line 14 are sequential HTTP requests. Since these HTTP requests will always be sent in sequence, their tasks can be accomplished in one HTTP request.

```
1  public void print_html()
2  {
3      URL url1, url2, url3;
4      URLConnection urlConnection1,urlConnection2,
           urlConnection3;
5      //query current weather
6      url1 = new URL("http://weather");
7      urlConnection1 = url1.openConnection();
8      ParseStream(urlConnection1.getInputStream());
9      //query weather forcast
10     url2 = new URL("http://daily");
11     urlConnection2 = url2.openConnection();
12     ParseStream(urlConnection2.getInputStream());
13     //query location info
14     url3 = new URL("http://location");
15     urlConnection3 = url3.openConnection();
16     ParseStream(urlConnection3.getInputStream());
17 }
```

Program 1: Example of sequential HTTP requests

To optimize the energy consumption of sequential HTTP requests, we propose an approach that combines sequential HTTP requests into a larger HTTP request. This approach can reduce the number of HTTP requests and increase the size of data transmitted by a single HTTP request. Our basic idea is to redirect the original HTTP requests to a proxy server that combines the sequential HTTP requests into a single one request. The work-flow of our approach is shown in Figure 1, where the dashed-line boxes represent the code from the original app and solid-line boxes represent the components that are developed in our approach. In our workflow, we first use code rewriting techniques to replace the Android HTTP APIs with the Agent HTTP APIs in an Android app. Then the Agent HTTP APIs will redirect the HTTP requests to the Proxy, which is a process that intercepts the HTTP requests from the client to the server. It contains two components, the Bundling Calculator and the Redirector.

When the client starts to send an HTTP request, it sends the request through the Agent HTTP APIs. Then, the Agent HTTP APIs redirect the request to the Proxy. When the Proxy gets the HTTP request, it passes the request to the Bundling Calculator, which calculates if there are any other HTTP requests should be bundled. Its decision is based on the rules provided by developers of the app. After that, the Bundling Calculator uses the Redirector to query for the data of all of the HTTP requests that need to be bundled from the server. When the responses are retrieved, the Bundling Calculator bundles the responses for all HTTP requests in one package and replies with it to the Agent HTTP APIs. Then, when the bundled responses return, the Agent HTTP APIs unpack the bundled responses and recover the response for each request. These responses are then store in a local cache or returned for the current HTTP request. Finally, when other HTTP requests in the same set of sequential HTTP requests are invoked, the responses will be returned directly from the local storage.

3.1 Example

We use the example in Program 1 to explain our approach. In Program 1, we first replace the HTTP APIs at line 8, line 12, and line 16 with our Agent HTTP APIs. When the HTTP request at line 8 is made, it is sent to the Proxy. Then the Proxy finds the incoming request is the first one of a set of sequential HTTP requests, which contains three HTTP requests of the link *weather* at line 6, *forecast* at line 10, and *location* at line 14. Such a decision is based on rules provided by the developers manually with their domain specific knowledge. Then the Proxy queries for the response to *weather*, *forecast*, and *location* from the server and returns them all in a single packet to the client.

When the Proxy sends back the packed response, our Agent HTTP APIs will capture this response at line 8 in Program 1. The Agent HTTP API first unpacks the response from the Proxy, retrieves the response for *forecast* and *location* from the packed response, and stores them in the local cache. Then it retrieves the response for *weather*, which is the target URL of the HTTP request at line 8, and returns the response as an InputStream, which is the same data structure as the original Android HTTP request.

After the Agent HTTP API at line 8 returns, the code makes the HTTP request to *forecast* at line 12 with the Agent HTTP APIs. At line 12, the Agent HTTP API checks the local cache and finds that the response is already stored, so it returns the response of *forecast* directly. This is the same for the case for *location*.

In this case, Program 1 only makes one HTTP request to the Proxy at line 8, which retrieves the responses for *weather*, *forecast*, and *location* together. This compared to the original version of Program 1, in which there are three HTTP requests made to retrieve the responses for *weather*, *forecast*, and *location*.

3.2 Code Rewriting

Our approach first needs to replace the original Android HTTP APIs with the Agent HTTP APIs. In this process, our approach parses the bytecode of each method in the original app and detects the signature of each invocation instruction. If there is an invocation instruction invoking an Android HTTP API, our approach redirects the invocation instruction to the corresponding Agent Android APIs.

3.3 The Agent HTTP APIs

The Agent HTTP APIs are static methods that replace the original HTTP APIs in Android SDK, such as *URLConnection.getInputStream*. The parameters of the Agent

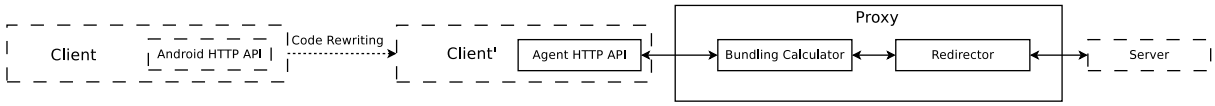

Figure 1: The work flow of our approach

HTTP APIs are the target object and the parameters of the original HTTP API. The return values of the Agent HTTP APIs are the same as the original API.

The Agent APIs have three main functions. First, check if the response for the current HTTP request is stored in the local cache, and if so, return it directly. Second, redirect the HTTP request to the Proxy instead of the server if the current HTTP request is not stored in the local cache. Third, unpack the bundled responses for a set of sequential HTTP requests, store the responses for them in the local cache, and return the response for the current HTTP request.

3.4 The Proxy

The Proxy is the process on the server side that intercepts the incoming HTTP requests between the client and the server. The first responsibility of the Proxy is to check if the incoming HTTP request is the first request in a sequence of HTTP requests and its second responsibility is to bundle the sequential HTTP requests. The Proxy accepts HTTP requests from the client and returns the bundled responses for those requests. The Proxy is normally deployed on the same machine or in the same network domain of the server. Thus, it can have a very high-speed connection to the server and will not significantly increase the latency between the server and the client. We believe this requirement is realistic because the user of our approach, who are the owners or developers of the app under optimization, generally have control of the app's server.

The first task is accomplished by the Redirector, which accepts the incoming HTTP request, retrieves the URLs of the corresponding sequential requests and, sends those requests to the server to get the responses.

The second task is accomplished by the Bundling Calculator, which accepts the URL and parameters of an incoming HTTP request and determines which requests should be bundled. This decision is made based on rules provided by the app developers with their domain specific knowledge about the app. For example, in Program 1, the developers can provide the rule "bundle the requests to *weather*, *forecast*, and *location* together" to the Bundling Calculator. Then, when the request of *weather* comes, the Bundling Calculator would query this rule and then bundle the response of *weather*, *forecast*, and *location* in one package.

3.5 The Communication Protocol

The Agent HTTP APIs and the Proxy use a specific protocol for communication. In this protocol, responses for several HTTP requests can be returned in a single packet in JSON format. When the Proxy returns data to the Agent HTTP APIs, it contains three types of information: first, the number of responses; second, the current request and its response; and third, all of the bundled requests and their responses.

When the Agent HTTP APIs get the response, they process the data with the following steps. First, they retrieve the response for the current request, this response will be used as the return value. Second, they iterate over all of the bundled requests, and retrieve their responses. Third, they cache the responses of all of the bundled requests in a local cache.

4. EVALUATION

In our evaluation, we answer the research question: how much HTTP energy could be saved by bundling the sequential HTTP requests as one HTTP request?

4.1 Test Apps

To answer our research question, we found 2 market Android apps from the Google Play Market that contain sequential HTTP requests. These two apps are **bob's weather** and **LIRR Schedule**. These apps have 22,517 and 4,408 lines of Java bytecodes, respectively.

4.2 Implementation

In the implementation, we used dex2jar and apktool to decompile the Dalvik bytecode of Android apps to Java bytecode. Then we used the BCEL library to replace the APIs that make HTTP requests with our Agent HTTP APIs. We used Node.js to implement the proxy server.

4.3 Evaluation Protocol

For our evaluation, we could not obtain the server side code of the apps since they were not open source apps. Thus, to evaluate our technique, we mimicked the behavior of the original server with a Node.js based mock-server. In our mock-server, we recorded and stored the responses of all HTTP requests that could be made by our two apps. When there was any incoming HTTP request, the mock-server simply replied with stored data according to the URLs and parameters of the requests. The mock-server and the proxy were all deployed on the Amazon EC2 cloud platform.

To evaluate the energy savings, we used our previous technique, vLens [9], to measure the HTTP energy consumption of the two apps. We compared the HTTP energy of the unoptimized version, which accessed the mock-server directly, with the HTTP energy of the optimized version, which accessed the proxy server, and reported the energy savings of our approach

4.4 Result

In our experiment, we found that there were energy savings of 50% for both apps for making HTTP requests. We believe this result is significant because as reported in our previous study, on average, making HTTP requests can consume 30% of the total non-idle state energy [8], we expect that bundling sequential HTTP requests could have a 10-15% reduction in the energy at the application level.

5. RELATED WORK

A large group of current studies on optimizing energy for mobile devices focus on detecting resource leakage of sensors (e.g., [13, 1]). Pathak and colleagues [15] proposed a tech-

nique to detect if an app fails to release a wake-lock of in a smartphone app.

Another group of energy optimization techniques focus on optimizing the display energy of mobile apps. Our previous work, Nyx [10], optimizes the energy consumption of mobile web apps by automatically transforming the color scheme of the apps. Mian and colleagues proposed dLens [17] to detect the display energy hotspots. Dong and colleagues also proposed a tool, Chameleon [3], to change colors of mobile web apps manually.

Besides detecting sensor resource leakage and display energy optimization, another group of techniques optimizes the energy consumption of mobile apps. Our previous work, EDTSO [11], optimizes the energy consumption of in-situ test suits by mapping the test suit optimization problems to an integer linear programming problem. Bruce and colleagues [2] used Genetic Improvement technique to optimize the energy consumption of MiniSat programs. Manotas and colleagues proposed a framework, SEEDS [14], to automatically make energy optimization decisions.

Although all approaches mentioned in the above three paragraphs achieved significant effect on energy optimization of mobile apps, none of them focus on optimizing the energy consumption of HTTP requests. As we discussed in Section 2, making HTTP requests is the most energy consuming operation in mobile apps.

Besides energy optimization, many techniques are also proposed to measure and estimate the energy consumption of mobile apps. In our previous work, we proposed two tools, eLens [5] and vLens [9], to estimate and measure the energy consumption of mobile apps at the source line level. Hindle [6] proposed a frame work, Green Mining, to measure energy consumption of mobile apps and related it with app code changes. eProf [15] models energy with a state machine. All of these approaches only measure the energy consumption of mobile apps, but do not do any optimization.

The last group of our related work is empirical studies. Our previous studies [7, 8] investigated how energy is generally consumed in mobile apps. Gui and colleagues performed an empirical study [4] about the hidden costs of mobile ads. Their hidden costs also included energy consumption. Linares-Vasquez and colleagues [12] conducted an empirical study on analyzing API methods and mining API usage patterns in Android apps. Li and colleagues [11] studied the energy consumption of different storage systems. Sahin and colleagues [16] proposed an empirical study about energy impact of code ossification.

6. CONCLUSION AND FUTURE WORK

Making HTTP requests is one of the most energy consuming operation in mobile apps. In this paper, we performed a preliminary study on how to optimize the energy consumption of HTTP requests in Android apps. Our approach is to bundle sequential HTTP requests into a single request. By doing so, we achieved 50% energy savings in HTTP requests in two Android market apps. Based on this result, we conclude that our technique is promising.

In our feature work, we will develop new techniques to automatically generate the bundling rules in the proxy and evaluate our technique with more market apps. Further, in this paper, our technique only optimizes the sequential HTTP requests in a single thread. In our future work, we will also optimize HTTP requests across different threads.

7. ACKNOWLEDGMENTS

This work was supported by NSF grant CCF-1321141.

8. REFERENCES

[1] A. Banerjee, L. K. Chong, S. Chattopadhyay, and A. Roychoudhury. Detecting energy bugs and hotspots in mobile apps. In *FSE*, 2014.

[2] B. R. Bruce, J. Petke, and M. Harman. Reducing energy consumption using genetic improvement. In *17th Annual Conference on Genetic and Evolutionary Computation. ACM*, 2015.

[3] M. Dong and L. Zhong. Chameleon: A Color-Adaptive Web Browser for Mobile OLED Displays. *IEEE Transactions on Mobile Computing*, 2012.

[4] J. Gui, S. Mcilroy, M. Nagappan, and W. G. J. Halfond. Truth in advertising: The hidden cost of mobile ads for software developers. In *ICSE*, 2015.

[5] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan. Estimating Mobile Application Energy Consumption Using Program Analysis. In *ICSE*, 2013.

[6] A. Hindle. Green mining: A methodology of relating software change to power consumption. In *MSR*, pages 78–87. IEEE Press, 2012.

[7] D. Li and W. G. Halfond. An Investigation Into Energy-Saving Programming Practices for Android Smartphone App Development. In *GREENS*, 2014.

[8] D. Li, S. Hao, J. Gui, and H. William. An Empirical Study of the Energy Consumption of Android Applications. In *ICSME*. IEEE, 2014.

[9] D. Li, S. Hao, W. G. J. Halfond, and R. Govindan. Calculating Source Line Level Energy Information for Android Applications. In *ISSTA*, 2013.

[10] D. Li, H. Tran, Angelica, and G. J. Halfond, William. Making Web Applications More Energy Efficient for OLED Smartphones. In *ICSE*, 2014.

[11] J. Li, A. Badam, R. Chandra, S. Swanson, B. L. Worthington, and Q. Zhang. On the energy overhead of mobile storage systems. In *FAST*, pages 105–118, 2014.

[12] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk. Mining energy-greedy api usage patterns in android apps: an empirical study. In *MSR*, 2014.

[13] Y. Liu, C. Xu, and S. Cheung. Where has my battery gone? finding sensor related energy black holes in smartphone applications. In *PerCom*, 2013.

[14] I. Manotas, L. Pollock, and J. Clause. Seeds: A software engineer's energy-optimization decision support framework. In *ICSE*, 2014.

[15] 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.

[16] C. Sahin, P. Tornquist, R. Mckenna, Z. Pearson, and J. Clause. How does code obfuscation impact energy usage? In *ICSME*, 2014.

[17] M. Wan, Y. Jin, D. Li, and W. G. J. Halfond. Detecting display energy hotspots in android apps. In *ICST*, April 2015.