Android Application Power Saving Techniques

Cody Azevedo

CS4960

Dr. Melanie Martin

13 November 2017

Abstract

 This research will aim to take a closer look at how Android applications manage their internal power consumption, and will analyze techniques used by developers who successfully manage to reduce their app's power consumption. By looking at commits made on source-providing repositories such as GitHub, we can examine common power management practices that were consciously made by developers, and mark how well the power consumption techniques actually work. This paper will extend the research of Lingfeng Bao, David Lo, Xin Xia, Xinyu Wang, and Cong Tian in which they perform empirical studies on power consumption-specific commits made in both Android applications and APIs that have been made publicly available through GitHub

Introduction

 It goes without saying that smartphones are ubiquitous throughout the globe. According to the IDC Quarterly Mobile Phone Tracker, 344.3 million smartphones were shipped around the world in the first quarter of 2017. The IDC tracker also shows us that roughly 85% of those smartphones shipped were Android OS devices. Given this vast population of Android devices, along with the increasing functionality of the apps themselves, the devices become more powerful and as a result, the power consumption increases as well. The main focus of this paper will be to identify the main power-saving techniques such as Power Adaptation- dynamic power adaptation based on the smartphone's battery state; Power Consumption Improvement- general methods used to either disable application features or optimizing functions to improve power consumption; Power Usage Monitoring- inform the users about the status of the battery; and Optimizing Wake Lock- the use of a tool that helps developers control the CPU, screen, and keyboard; This paper will also extend its research into the implementation of Wake Lock, and how it is used to help developers manage their power consumption.

 Power Adaptation

 Power Adaptation is described as the dynamic ways a smartphone adapts its power usage based on a measure of the smartphone's battery state, such as power-save mode, low-power mode, or high-power mode. Using these strategies, some Android applications define their power management techniques to make sure that the main features perform as optimally as possible in accordance to the state of the battery. By observing different commits on GitHub made of applications on F-Droid, an app store for open-source Android applications, we can see how certain developers take advantage of different power-saving modes.

Taking a look at how developers make use of the power-save mode battery state, one developer committed “Only disable WiFi (battery saving mode) when JAWS switched it automatically on” (Lingfeng Bao, 42). According to the F-Droid repository, JAWS (Just Another Wifi Scanner) is a simple wifi scanner that supports real time scans of nearby networks. This developer made the conscious decision to disable Wifi *only* when JAWS switched it on through its default setting. Here’s a code sample of the application:

private void startScanning() {

 isScanning = true;

 if (!wifiManager.isWifiEnabled()) {

 jawsAutoEnabledWifi = true;

https://github.com/jannispinter/jaws/blob/master/app/src/main/java/is/pinterjann/jaws/activities/JAWSActivity.java

This code sample taken from the main method of the application *immediately* checks to see if Wifi is enabled, and automatically enables it if it’s not already on. WiFi is a high source of battery usage among Android devices, and it’s important to note that constantly making sure Wifi isn’t automatically turned on by a different application such as JAWS while in power-save mode.

 Android developers commonly take into consideration low battery mode, a state in which the battery is low on power or under a certain predetermined power threshold. This is the result of looking at different commits made on GitHub within the context of a low powered battery: “ActiveMode: Option to disable on low battery.”, “Send turn off broadcast when battery is low”, “Stop scanning on low battery”, “Auto-off on low battery only if the device is not plugged-in”, “shutdown at certain battery level” (Lingfeng Bao, 42). While not many features here are specific enough to get into detail about how they function, it is worth noting that Android applications tend to disable some of their functionality in order to preserve battery life past a certain threshold.

 There also exists methods of power saving when the battery is in high-power mode or is charging. This usually consists of adding the ability to *only* allow certain features of the application when the battery is in this mode. This is a mode in which the developer themselves has to know the power consumption rate of their own features in order to know which features to enable only when the battery is high or charging. Here’s a few examples of commits made in conjunction with high-power mode:“activate artist and album search by default on higher power phones”, “Added Wifi on when power connected rule”, “added turn wifi on when power connected and don’t turn wifi on when in airplane mode feature”. From these commits we can see that mobile devices in high-power or charging mode can enable certain features when the battery isn’t in a critical state, enabling higher functionality as a result, and improving the user experience without risk of draining the battery percentage.

Power Consumption Improvement

 The techniques used to improve power consumption vary depending on the application. Some techniques include disabling features completely, optimizing functions or entire code models, or converting to more power efficient libraries. This differs from power adaption in that these are permanent changes made to the application. Whereas power adaptation compensates some of its functionality when it detects the device is in a certain power state, these improvements are made towards the application as a whole to reduce the overall power consumption during normal use.

One commit made in this category demonstrates how reducing the frequency of updates or the number of function calls reduces overall power consumption: “Revert ‘Add event content observer’. Content observer is called multiple times without any apparent reason which leads to high battery usage.” (Lingfeng Bao, 42). From the official Android development library reference, the ContentObserver receives call backs for changes to content. It is constantly checking for updates to content, allowing the developer to make changes to the UI based on the content received, such as updating a record in a Call Logs database when a call is received. Due to the repetitive and complex nature of ContentObserver, this developer has made the decision to revert the implementation of it altogether in their application, thus increasing overall power consumption.

 With regards to saving power in the context of using more power efficient functionalities, oftentimes developers will use different API tools. Again from utilizing the GitHub commit search interface, one commit was made claiming to “use a more battery efficient heartbeat”. Here’s a snippet of code from the commit before the change:

if (interval > 0)

 alarmManager.set(AlarmManager.ELAPSED\_REALTIME\_WAKEUP,

SystemClock.elapsedRealTime() + interval,

mPendingIntent);

And after the change:

if (interval > 0) {

alarmManager.setInexactRepeating(AlarmManager.ELAPSED\_REALTI

ME\_WAKEUP, SystemClock.elapsedRealtime() + interval,

interval, mPendingIntent);

}

https://github.com/joskarthic/chatsecure/commit/f8c4d5f10d900e822467ef9ba1b8557e395d068f

Here we can see that the function call alarmManager.set was changed to alarmManager.setInexactRepeating. From the official Android AlarmManager reference, the “set” function call simply schedules an alarm, triggered every given millisecond. After the switch, the application now calls “setInexactRepeating” and schedules a repeating alarm that has inexact trigger time requirements. As a result, the function isn’t called every predetermined interval, but instead has inexact time triggers. This results in a more power efficient application by switching from a less efficient to a more efficient method call.

Power Usage Monitoring

 Applications in this category are described by Lingfeng Bao as applications whose developers incorporate some version of user interface that informs the user about how quickly the application is draining battery power. This can include things such as showing the battery is low by displaying the label red, or even using the application’s interface itself as a warning for low power. Taking a look at this GitHub commit made in this category “Use CircleChart for battery level” and examining some code from the commit:

mBatteryLevelCard = new UsageCardView.DUsageCard();
 mBatteryLevelCard.setText(getString(R.string.battery\_level));

addView(mBatteryLevelCard);

https://github.com/robcore/MKM/commit/ed82d50a509a96f8bc69b211d3066ea0bc2fa158

This code segment shows us that the developer has access to the operating system’s battery level by calling R.string.battery\_level and chooses to add the mBatteryLevelCard that was previously set to the battery level of the device to the a certain predetermined view in the application. While this doesn’t provide much insight as to how developers reduce the power consumption of their application, it’s worth mentioning that this is an approach some applications take with regards to optimizing battery usage, by simply allowing the user to decide when the application is using too much battery, then stop using the application altogether.

Optimizing Wake Lock

 A more technical approach many Android applications utilize is through the use of WakeLock, which is a mechanism provided by the official Android library class PowerManager that allows the developer’s application that it requires the device to stay on. According to the Android reference on WakeLock, creating and holding a wake lock usually has a dramatic impact on the device’s battery life. It’s also advised that the developer should use wake locks only when it’s necessary, and utilize them for as little a time as possible. From this advice, it’s apparent that while wake lock provides heavy functionality in terms of background CPU usage, it consumes large amounts of resources from the host device. Here’s a code sample from a commit made in which “Release WakeLock in Presto’s MediaPlayer when it is not needed”:

 private void stayAwake(boolean awake) {

 if(mWakeLock != null) {

 if(awake && !mWakeLock.isHeld()) {

 mWakeLock.acquire();

 } else if (!awake && mWakeLock.isHeld()) {

 mWakeLock.release();

 }

 }

 }

 https://github.com/robcore/MKM/commit/ed82d50a509a96f8bc69b211d3066ea0bc2fa158

In this figure, within the context of this specific application there’s a method declared called stayAwake in which the WakeLock is activated when the screen is already awake, and releases the WakeLock when the screen isn’t already awake and the WakeLock is being held on. Regarding the nature of the heavy resource consumption of WakeLock itself, this example further demonstrates how utilizing WakeLock only when necessary can optimize both the application’s functionality *and* power consumption when used appropriately.

Using it appropriately entails releasing the wake lock after using it, lowering the wake lock time as much as possible, and utilizing a flag value of WakeLock known as PARTIAL\_WAKE\_LOCK. If you hold a partial wake lock, the CPU will run while ignoring things such as display timeouts, the screen being awake or not, and instances of the user pressing the power button. When you don’t use a partial wake lock, the CPU will run but the device can still be put to sleep after the power button is pressed. Efficient usage of a partial wake lock can be seen in this commit that claims “optimized partial wake lock usage to save battery life”:

 stayAwake = pm.newWakeLock(PowerManager.PARTIAL\_WAKE\_LOCK,

"MyWakeLock2");

stayAwake.acquire();

 //check when screen turns off to acquire partial lock and

//release when on

lockFilter.addAction(Intent.ACTION\_SCREEN\_ON);

 lockFilter.addAction(Intent.ACTION\_SCREEN\_OFF);

 registerReceiver(sleepReceiver, lockFilter);

 stayAwake.release();

https://github.com/eochoa5/PhoneFindr/commit/0ee188521980feaf94eacc7d998a4df5bc27f672

In this code snippet, the application developer created a PowerManager with the PARTIAL\_WAKE\_LOCK flag value named stayAwake. As the comment indicates, when the screen is detected to turn off, the partial wake lock is turned on as a result. This ensures that the CPU is running in the background, even when the screen and keyboard backlight are turned off by the user. By implementing this function, while it does consume power while the screen is turned off, power is being saved by not requiring both the screen and keyboard to be light while the CPU is running.

Conclusion

 In this paper, I analyzed different ways in which Android developers optimized power consumption in their apps. The applications mentioned in this paper were sourced from F-Droid, an open source Android app store, and the commits observed were from the applications hosted on GitHub. By extending the research of Lingfeng Bao, David Lo, Xin Xia, Xinyu Wang, and Cong Tian in which they curated a list of commits made within the different types of power consumption saving techniques, the main common power management activities were described with detail while looking at specific code examples of how these techniques were implemented. While no particular strategy for improving power consumption proves dominant over the others, it’s important to take note of the major steps these applications have made in light of Optimizing power consumption in Android devices.

References

“AlarmManager.” Android Developers, 25 Oct. 2017,

developer.android.com/reference/android/app/AlarmManager.html.

Bao, L., Lo, D., Xia, X., Wang, X., & Tian, C. (2016). How android app developers manage

 power consumption? Proceedings of the 13th International Workshop on Mining

 Software Repositories - MSR 16. doi:10.1145/2901739.2901748

“IDC: Smartphone OS Market Share.” Www.idc.com,

www.idc.com/promo/smartphone-market-share/os.

“Keeping the Device Awake.” Android Developers, 1 Sept. 2016,

 developer.android.com/training/scheduling/wakelock.html#cpu.

Liu, Y., Xu, C., Cheung, S., & Terragni, V. (2016). Understanding and detecting wake lock

 misuses for Android applications. Proceedings of the 2016 24th ACM SIGSOFT

 International Symposium on Foundations of Software Engineering - FSE 2016.

doi:10.1145/2950290.2950297

 “PowerManager.” Android Developers, 25 Oct. 2017,

developer.android.com/reference/android/os/PowerManager.html#PARTIAL\_WAKE\_L

OCK.