Android Resource Monitor

Christian Heimlich
Traz Daczkowski

Intro to Mobile Platform Development
12/19/16

Table of Contents

Introduction

The Android operating system allows developers tools to monitor the performance of the system that the apps are running on. However sometimes the user wishes to know about the system's performance without having to go through the menus of the built in settings app. A resource monitor application will give the user insight into the inner workings of the phone/tablet and provide an assortment of status updates to the user about the phone's current condition. This can include information such as RAM usage, CPU utilization, storage usage/capacity, Wi-Fi status, etc.

Background

Android devices tend to be either phones or tablets. Usually the only significant difference between these devices are that the phones tend to have weaker processors, but connect to cell towers. Both of these devices are weaker than desktop computers. Therefore it is easy for a single task to consume all of the system's resources. It is important for the end user to be able to understand why their device is slow so that they have the ability to remedy it.

Due to the nature of the android operating system, Google has found it necessary to require a strict and explicit permissioning system. Unfortunately, they also feel the need to continuously update this system. This makes any application requiring abnormal or extensive permissions to include compatibility with each API version separately. Because of this restriction the APIs targeted were 18-23 due to a large overhaul of the permissions system that occurred during the transition to API 18. As a result of this and other complications to be discussed later, it is difficult to know just how many devices this application will support.

Objectives/Scope

The objective that we had for the app was to provide an all inclusive task *manager* style resource monitor. This app was to allow users to manually terminate threads and free up resources in use by other apps. Additionally, we hoped to include other management features such as package manager to uninstall packages or wipe their data/cache, and the ability to change system settings directly through our application. However, do to the limitations imposed by the android operating system the application had to be reduced to a read-only version of what was originally planned, hence the designation "resource *monitor*". The ability to free up the systems resources required permissions that are not available to applications since API 17. The scope was changed to displaying each of the main components of the system and their current status.

First the user needs to be greeted with the a screen that does not require any significant resources to run as a primary use case for the application is when the phone is struggling under an unknown load. Because of this, the decision was made to minimize the load on the first screen.

Implementation

MainActivity/Storage Information:

The first screen that the user sees has buttons to each of the other activities. The user is also shown a bar that shows how much of the secondary storage of the device is used. The bar is implemented as a large horizontal ProgressBar. This was chosen as the bar used to avoid having to draw any custom shapes and suited the screens purposes perfectly. The blocks free and block size are gotten via the core linux command statFs. The stats are then passed into the progress bar and the exact numbers are put into the TextViews. Additionally, the name of the user's device is

shown as the title for the internal storage. If an external path is then found to not be emulated the SD card's information is shown beneath the internal storage. This does not work for all OEM's however due to various different paths being used within the filesystem. While many manufactures use "/storage/sdcard", there is no established standard. Therefore, in some cases the information displayed for the internal and external storage will be the same. To detect when the external storage is emulated the getIsExternalEmulated() function is called. If it is emulated that means the directory given as external is actually internal.

RAM & CPU Information:

Then, an activity was created to provide the user with CPU and RAM usage statistics, which was determined to be the next most important set of information. The layout for the activity was set up similarly to the storage information area of the MainActivity, in that TextViews and horizontal ProgressBars were used to display these stats to the user. Since CPU and RAM usage are often in constant flux, the activity automatically updates this information every second. This is accomplished by setting up the variables and UI elements necessary for the activity in its onCreate() function and then executing a custom Runnable which is tied to a Handler. The Handler manages inserting the Runnable into the main thread while the Runnable calls a function named "updateStatus" repeatedly on a set interval (1000ms). All researched calculations, system class usage, and UI changes occur within updateStatus(). Additionally, it was important to halt the Runnable when the activity's onDestroy() method is called so that the usage statistics are not constantly polled in the background.

The first set of information shown to the user on this page is CPU statistics, which includes current utilization and max frequency. CPU usage is determined by reading a text file

present on every android device that contains statistics about CPU time spent on various processes. First the file is read, next the main thread of the activity is paused for 360ms, then the file is read again, and then the differences between how much of CPU time was allocated to each process type and how much time the CPU spent idling in each instance are used to calculate CPU utilization as a fractional percentage (0% to 100% usage). This information is then sent to the TextView and horizontal ProgressBar that both represent current CPU usage.

In order to increase the functionality and quality of the activity the current CPU usage is also constantly plotted over the last thirty seconds as a line graph, similar to Windows Task Manager. To simplify this task, and ensure the application did not take longer than allowed to develop, the open source "MPAndroidChart" library by PhilJay was used. This library is easily installed/added to any project by adding just two lines to the app's build.gradle file. All thirty points (thirty seconds, one sample every second) of the chart are stored in a Java List of Entries, a custom class provided with the library that stores an X and Y value. Various aspects of the charts style/settings are initialized, including starting each point at zero since CPU utilization is unknown until the activity is running, and then the chart and its point list are updated every second within the updateStatus() function. More information on the library and its usage can be found on its corresponding Github page as noted in the References section.

Additionally, the maximum frequency of the devices CPU is shown to the user. This statistic is obtained by reading another system file that records the time the CPU spends in each power state. The file is scanned line by line and the frequency of each state is compared. Once the highest frequency, which corresponds to the full power state, is determined it is displayed in the activity's corresponding TextView.

The only other information displayed within this activity is the device's RAM usage, which was much easier to obtain. An instance of Android's built in ActivityManager class is created which provides direct access to the systems total and currently available RAM using getMemoryInfo(). A simple difference is taken between the total and available RAM to determine RAM usage, and that value is then divided by the total RAM to calculate percent utilization. These statistics are then passed to their corresponding TextView and ProgressBar.

Process Information:

It was previously possible to also use the Android class ActivityManager and getRunningAppProcesses() to generate a list of the systems currently running processes. However, due to Google continually reducing access to system information for third-party applications, this now only provides information about the current process (which in this case is our app) since 5.0 (Lollipop). This would mean that most of the users we are targeting could not make use of this functionality. This issue is a commonly discussed topic on Android forums and Q&A sites, and while it seemed that a custom workaround was possible, there was little information on how to even get started and what we could find indicated it would be a lengthy a challenging process that could easily be a project on its own. Luckily, jaredrummler on Github created a free open source library (which surprisingly requires no permissions) that can generate a system wide process list and provide most of the information that ActivityManager used to. This library covers our minimum API of 18 and works up to API 23, and can be included by adding just one line to the app's build.gradle dependencies list.

The process list is generated using two simple functions provided with the library. A loop is then used to parse each process's name, ID, and corresponding app name from a Java List into

two arrays. Then, a ListView and ArrayAdapter are created to successfully display the list of processes and their related information to the user. The ArrayAdapter is overridden to add a line of sub-text for each ListView entry in addition the the default primary TextView. The primary text of each ListView entry displays the name of the app the process is a part of as well as the process ID, while the sub-text displays the process name itself. A check was implemented to prevent the application from throwing an exception in no process ID was found since many system process don't have one, and in this case the PID is simply displayed as "None". Additionally, a similar check was implemented if no app name was found since again, many system apps don't have one, and in this case the app name is simply displayed as "SYSTEM". More information on the library and its usage can be found on its corresponding Github page as noted in the References section.

After the above implementation this activity would normally have been complete; however, even on a fast android device generating and parsing the process list took and upwards of ten seconds, which left the app frozen and even caused a black screen for a few seconds, which is not user-friendly in the least. To circumvent this, Android's built in class AsyncTask (which stands for asynchronous-task) was used, which provides a simple way to place a slightly lengthy process on a thread separate from the activity's main thread. This allows the activity to launch almost instantly and show some UI elements instead of being stuck at the MainActivity screen for a long period. Once the activity's onCreate() function is finished the task is executed and the UI elements are allowed to be rendered while the task continues on a second thread. A custom class that extends AsyncTask was implemented and its four main functions were overridden to match the needs of our application. First, in onPreExecute() a class variable

representing the activity's ProgressBar is defined. Then, in doInBackground() all of the implementation described previously of generating and parsing the system process list is performed. Each time an iteration of the main loop within this function completes it calls the third AsyncTask function onProgressUpdate() which updates the appearance of the activity's ProgressBar. Lastly, once the lengthy task is completed onPostExecute() is called which sets the custom ArrayAdapter to process ListView, removes the progress bar, and sets a TextView that previously said "PLEASE WAIT" to a legend that notes the format of each ListView item. The list of processes, their IDs, and corresponding app names are then finally displayed to the user and the activity can be used as normal.

Wireless Information:

Lastly, an activity was created to give the user information about various radio-based functionalities in the device. The first element shown to the user is the WiFi connectivity information. This requires the permissions "Access Wifi State" and "Access Network State". The Connectivity Manager that will be used throughout the activity. The connectivity manager is a system service that allows the application to access various functions relating to the wireless (and ethernet when present. Although the application created will crash if the device has both a mobile network and an ethernet cable in use) connectivity on the device. This object is then queried for the Wifi Network info. If the Wifi is connected the network's SSID will be displayed as well as the theoretical maximum speed that the network is capable of based on 802.11 specifications. This speed is rarely achieved due to noise but it is nice to have displayed.

The second piece of information shown is the bluetooth status. This uses the BluetoothAdapter class which requires the "Bluetooth" permission. If the device does not have

bluetooth the TextView will read Unsupported. Otherwise, the text will read enabled or disabled. The settings button requires the “Bluetooth Admin” permission if running on some models of phones, dependent on the manufacturer.

The mobile network is then gotten from the connectivity manager. If it exists on the device, even if disconnected, the code continues. Otherwise, the code will hide the rest of the layouts. A Telephony manager is then created and queried for the network type. While most users are only familiar with 2G,3G or 4G, there are actually 15 types of networks. What’s worse is that they’re not perfectly in order with the common generational nomenclature. Thanks to extensive googling I believe the networks are all classified correctly, however testing of each network type was not done due to not having access to a device that employs all 15. There is also a default case that will work for any future network types that get added to android. The signal strength in dBm is then put into another TextView. This is implemented as a class that extends PhoneStateListener and overrides the onSignalStrengthsChanged system event. The signal strength then is read using either the Gsm or the Cdma functions. I don’t know why this is 2 separate functions or why getGsmSignalStrength doesn’t actually return the dBm. Because this function uses a listener this field will automatically update while the user sits on the screen.

The user’s SIM card status is then checked and displayed. If the user has a SIM card in the device and is running a version before marshmallow the IMEI of the SIM card is also displayed. This function does not work on the newer versions of android because a system was added where the user needs to be prompted to give permission to the application every time. This interrupts the user experience and requires a large amount of additional code. As such it was decided to inform the user why the feature doesn’t work for them but not to do anything about it.

References

http://stackoverflow.com/questions/33078003/android-6-0-permission-error
http://stackoverflow.com/questions/10527206/getting-signal-strength
http://stackoverflow.com/questions/30006488/android-provider-settings-action-bluetooth-settings-crashes-on-samsung
http://stackoverflow.com/questions/33350250/why-getexternalfilesdirs-doesnt-work-on-some-devices
http://stackoverflow.com/questions/31700842/which-intent-should-open-data-usage-screen-from-settings
http://stackoverflow.com/questions/3118234/get-memory-usage-in-android
https://developer.android.com/reference/packages.html
https://github.com/jaredrummler/AndroidProcesses
https://github.com/PhilJay/MPAndroidChart