Images haven't loaded yet. Please exit printing, wait for images to load, and try to print again.

Software Specialist, Google Developer Expert on Android, traveller, phographer, istanbul lover www.elif...
Jan 21, 2016 · 10 min read

18.

Android Application Performance— Step 4: Battery

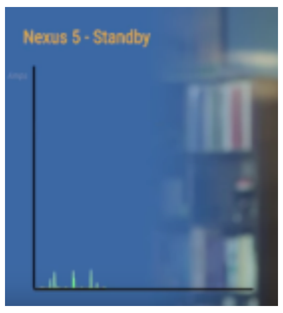
I am moving on with **Battery**. Battery is the last step that I talk about in this series. As taking the Udacity Performance class as reference, I have tried to explain main performance problems and how we can analyze them.

Hardware of our mobile device drain battery while executing the tasks or uploading our cat's pic. According to this, how much power will be drawn from the battery and how long the battery will be stood could change. And how we can develop less battery draining applications is related with how well we understand the background process.

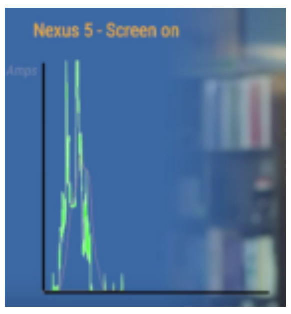
Lets take an Android device and keep it to airplane mode. It will nearly drain no battery. The device drains battery as long as being active. And we define the active here, we can that CPU's working jobs, radio's data transfer, the screen's refreshing itself.

So, which tasks do drain more battery?

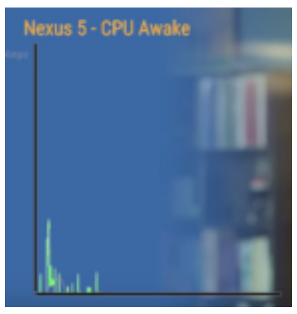
Actually, it is not an easy question to answer. It is very complicating to monitor the battery at the level of hardware, because monitoring itself also drains battery while trying to measure how much battery is drained. To be able to get that kind of information accurate, we need to integrate a third part hardware to the device. So we can measure the battery consumption without draining the device's battery.



The pic above shows the state of Nexus 5 on airplane mode.



If we wake up the device or open the screen vs. the state will like on above.



If the device is awakened by the APIs the state will be like above. (Alarm, scheduled tasks) We see a peak here at the very beginning of the device awakening, after that a series of tasks follow it.

The important point is here, after the job completed, the device backs to the sleep mode again. Doing a little work but keep the device awake for a long time can drain battery easily.

If you use Android L, there are lots of tools to analyze the battery usage.

Batterystats & Battery Historian

Batterystats takes data from our device bout battery. On the other hand Battery Historian converts this data to HTML format to be able to see it on Browser. Batterystats is a part of Android framework and Battery Historian is on Github as opensource.

https://github.com/google/battery-historian

What is it for?

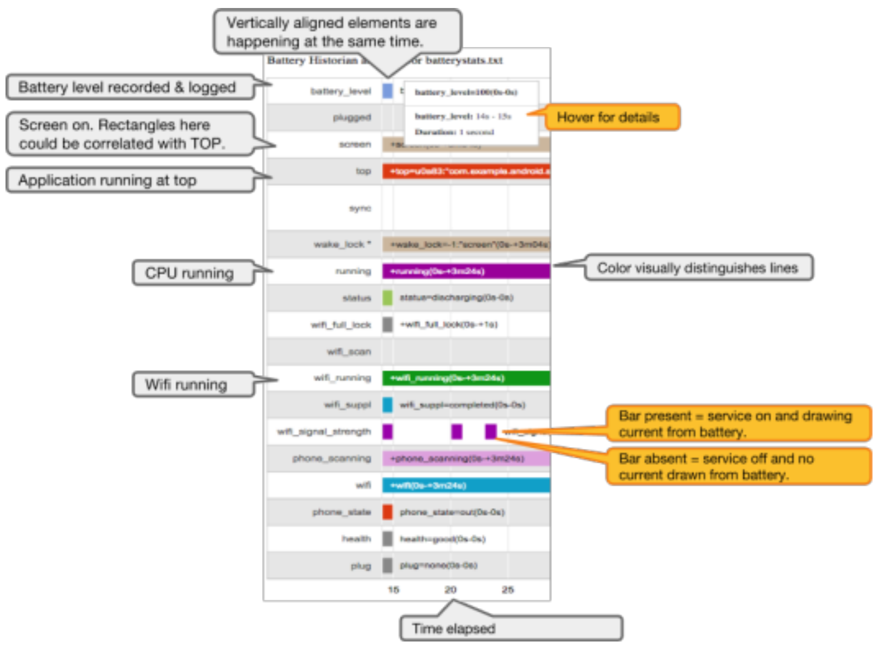
> It shows the immediate processes how and where work.
>
> It defines the task that could be postponed or removed to be able to extend the battery life.

It works on 5.0+.

If we move step by step how we use batterystats;

> Firstly, we should download opensource Battery Historian script from Github. (https://github.com/google/battery-historian).
>
> Then the folder of Battery Historian should be unzipped. we should move the file which is name of historian.pf from that folder to desktop or any other place that we can easily edit.
>
> Connect your device to the computer.
>
> Open your terminal on your computer.
>
> go into the folder which historian.py file is saved in.
> örnek: cd ~/Desktop
>
> Kill adb server.
> \> adb kill-server
>
> Restart adb and control the connected devices.c
> \> adb devices
>
> Reset the battery data.
> \> adb shell dumpsys batterystats — reset
>
> Disconnect your device from the computer, so we could take only usage of the battery in that moment.
>
> Move around the application that you want to test.
>
> Reconnect your device to the computer.
>
> Be sure about your device is recognized.C > adb devices
>
> Dump all the battery:
> \> adb shell dumpsys batterystats > batterystats.txt
>
> Generate a HTML version of the dump for Battery Historian:
> \> python historian.py batterystats.txt > batterystats.html
>
> Open the batterystats.htm on Browser.

Battery Historian Charts


Vertically aligned elements are happening at the same time.
Battery level recorded & logged
Screen on. Rectangles here could be correlated with TOP.
Application running at top
Hover for details
CPU running
Color visually distinguishes lines
Wifi running
Bar present = service on and drawing current from battery.
Bar absent = service off and no current drawn from battery.
Time elapsed

As we see on the pic above, each line shows an active system component and battery draining.

Battery Usage Categories: (This part is completely taken from developer.android)

battery_level: When the battery level was recorded and logged. Reported in percent, where 093 is 93%. Provides an overall measure of how fast the battery is draining.

top: The application running at the top; usually, the application that is visible to the user. If you want to measure battery drain while your app is active, make sure it's the top app. If you want to measure battery drain while your app is in the background, make sure it's *not* the top app.

wifi_running: Shows that the Wi-Fi network connection was active.

screen: Screen is turned on.

phone_in_call: Recorded when the phone is in a call.

wake_lock: App wakes up, grabs a lock, does small work, then goes back to sleep. This is one of the most important pieces of information. Waking up the phone is expensive, so if you see lots of short bars here, that might be a problem.

running: Shows when the CPU is awake. Check whether it is awake and asleep when you expect it to be.

wake_reason: The last thing that caused the kernel to wake up. If it's your app, determine whether it was necessary.

mobile_radio: Shows when the radio was on. Starting the radio is battery expensive. Many narrow bars close to each other can indicate opportunities for batching and other optimizations.

gps: Indicates when the GPS was on. Make sure this is what you expect.

sync: Shows when an app was syncing with a backend. The sync bar also shows which app did the syncing. For users, this can show apps where they might turn syncing off to save battery. Developers should sync as little as possible and only as often as necessary.

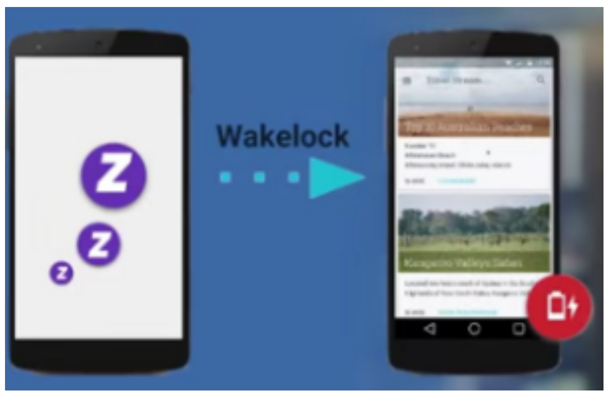
Filtering batterystats output:

If we save the batterystat.txt file from batterystats command, we can reach to more information.

If we open the file and search the keywords below: **(This part is completely taken from developer.android)**

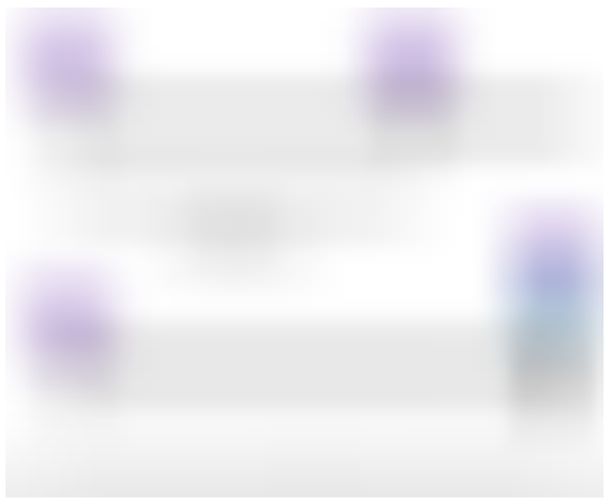
> **Battery History**: A time series of power-relevant events, such as screen, Wi-Fi, and app launch. These are also visible through Battery Historian.
>
> **Per-PID Stats**: How long each process ran.
>
> **Statistics since last charge**: System-wide statistics, such as cell signal levels and screen brightness. Provides an overall picture of what's happening with the device. This information is especially useful to make sure no external events are affecting your experiment.
>
> **Estimated power use (mAh)** by UID and peripheral: This is currently an extremely rough estimate and should not be considered experiment data.
>
> **Per-app mobile ms per packet**: Radio-awake-time divided by packets sent. An efficient app will transfer all its traffic in batches, so the lower this number the better.
>
> **All partial wake locks**: All app-held wakelocks, by aggregate duration and count.

Sometimes, problems occur when the device is in sleep mode. Awakening the device seems to be a good idea to be able to do the job, but we can face with serious battery problems in that kind of situations. For example, if there is an application that checks the scoreboards continuously, uploading photos or downloading data with a specific hashtag, device continues to drain battery even in sleep mode. The reason of this is device's being awakened by the application continuously.



Actually, Android closes many parts of the hardware to save the battery life. Firstly, the screen darkens and then closes itself. And finally, CPU goes into sleep mode. All these things are done to be able to save the battery life. On the other hand, most of the apps goes to work even in inactive mode.

Although the device is on sleep mode and even applications are not active, most of the application try to work and to be able to do this, they have to awake the device. The simplest way of awaking the application is using **powermanager.wakelock** API. This API prevents screen's blackout or closing, while GPU is going on to work. This provides application's awakening in a future time, doing the job and returning back to sleep mode. The tricky part is the last one at here. While when the device will be awakened to do a job is the easy part to decide, when the device will gone into sleep mode is not simple that much. For example, while 10 sec is enough to load the images, what happens if it takes 60 min? Or server crashes, can not take answer and wait to the end? As a result, the device waits to get response until the battery is drained totally. Because of all these reasons, using wakelock.acquire for timeout is a good choice. It provides **wakelock's** being **released** and preventing the device's battery being drained totally when that of situations happen. But this does not solve the problem completely. It sets the timeout at these situations, but there is another concern. If it takes error, how long will it wait for next try?



As using **inexact timer**, we can programme the job being executed in a future time. But if the system realizes that this job could be done sooner or later to save the battery life, it will wait that time. For example, if another process awakens the device, our process could wait 10 sec for these processes instead of working on the programmed time. This kind of situations can happen. But if our application needs to awaken the device, we should give extra attention to save the battery life. This is actually why **job schedular API** exists. This API is suitable for some kind of future time programmed jobs being done as saving battery life.

The pic above is battery storing of a hyper-active application. (It is taken from Google IO 2014 Project Volta.)

If we interpret the graph, when we look at the wakelock activity, lots of gaps mean the device's being awakened so many times. When we look the network activity, all of these means battery drain. (You can get more information as watching Project Volta talk.) So what kind of code causes a graph like this?

I will move on the application which is on Github added by Udacity Performance Class. We implement a wake lock, if we want the device to awaken as traditionally.

As using the code sample on Github, the method on pic below uses wakelock. **Firstly**, it acquires wakelock to awaken the device. So network update could be done anymore. **And then**, it should be controlled that if the network is connected or not. If it is not connected, it will move on to try until being connected. If it is connected, it will take the new data from server in an Asynctask independently from UI thread. When work is completed, it goes to AsyncTask's postExecute and it can be released here.

```
private void pollServer() {
    mWakeLockMsg.setText("Polling the server! This day sure
went by fast.");
    for (int i=0; i<10; i++) {
        mWakeLock.acquire();
        mWakeLockMsg.append("Connection attempt, take " + i
+ ":\n");

mWakeLockMsg.append(getString(R.string.wakelock_acquired));


        // Always check that the network is available
before trying to connect. You don't want
        // to break things and embarrass yourself.
        if (isNetworkConnected()) {
            new SimpleDownloadTask().execute();
        } else {
            mWakeLockMsg.append("No connection on job " + i
+ "; SAD FACE");
        }
    }
}

@Override
protected void onPostExecute(String result) {
    mWakeLockMsg.append("\n" + result + "\n");
    releaseWakeLock();
}
```

The main problem with wakelocks here is keeping and releasing by developer as manual. I mean, if we keeping the wakelocks for a long time, we can drain all the battery of the user.

So, how we can optimize this situation as using **Job Scheduler API**? You can examine the sample code that is shared on Udacity class from the link. As a difference, we need to create a service endpoint to be able to use Job Scheduler. This is actually the service that will be called to the job by the system when it is suitable. We handle this as extending JobService.

```
mServiceComponent = new ComponentName(this,
MyJobService.class);
```

The important point on JobService is implementing the onStartJob and onStopJob callbacks.

On the onStartJob method, we implement the logic of downloading data from network will be working on background.

```
@Override
public boolean onStartJob(JobParameters params) {
    // This is where you would implement all of the logic
for your job. Note that this runs
    // on the main thread, so you will want to use a
separate thread for asynchronous work
    // (as we demonstrate below to establish a network
connection).
    // If you use a separate thread, return true to
indicate that you need a "reschedule" to
    // return to the job at some point in the future to
finish processing the work. Otherwise,
    // return false when finished.
    Log.i(LOG_TAG, "Totally and completely working on job "
+ params.getJobId());
    // First, check the network, and then attempt to
connect.
    if (isNetworkConnected()) {
        new SimpleDownloadTask() .execute(params);
        return true;
    } else {
        Log.i(LOG_TAG, "No connection on job " +
params.getJobId() + "; sad face");
    }
    return false;
}
```

It is not a specific work should be done on onStopJob method. But if our job is stopped before expected then this is called. For example, disconnection of wi-fi unexpectedly. And another important point is to call the job finish method when our work is completed.

```
@Override
protected void onPostExecute(String result) {
    jobFinished(mJobParam, false);
    Log.i(LOG_TAG, result);
}
```

The code sample below shows the new version of pollServer() method.

```
public void pollServer() {
    JobScheduler scheduler = (JobScheduler)
getSystemService(Context.JOB_SCHEDULER_SERVICE);
    for (int i=0; i<10; i++) {
        JobInfo jobInfo = new JobInfo.Builder(i,
mServiceComponent)
                .setMinimumLatency(5000) // 5 seconds
                .setOverrideDeadline(60000) // 60 seconds
(for brevity in the sample)

.setRequiredNetworkType(JobInfo.NETWORK_TYPE_ANY) // WiFi
or data connections
                .build();


        mWakeLockMsg.append("Scheduling job " + i + "!\n");
        scheduler.schedule(jobInfo);
    }
}
```

References:

https://www.udacity.com/course/viewer#!/c-ud825/l-3758168642/m-3731358949

http://developer.android.com/tools/performance/batterystats-battery-historian/index.html

http://developer.android.com/tools/performance/batterystats-battery-historian/charts.html

https://developer.android.com/reference/android/app/job/JobScheduler.html