POLITECNICO DI MILANO

Facoltà di Ingegneria dell'Informazione

Corso di Laurea in Ingegneria Informatica

Technical Report

The MPower LogService and the JSON logging file

Matteo Ferroni

ferroni@elet.polimi.it

Version 0.1 - Last update: N/A

NECST Laboratory

mpower.necst.it

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

API	Application Programming Interface
FSA	Finite State Automata
HAL	Hardware Abstraction Layer
IMEI	International Mobile Equipment Identity
IV	Inizialitation Vector
JSON	JavaScript Object Notation
LTE	Long Term Evolution
OS	Operating System
TTL	Time-To-Live
VM	Virtual Machine

Chapter 1

Logging Service

The LogService is the component in charge of retrieving the status of the device through the Sense Libraries. It extends the Android `Service` class, meant to execute long background operations. All its tasks are described in the following paragraphs.

Log operations

LogService performs a log operation with a certain *sample frequency*. On the one hand, the more this sampling frequency is high, the more accurate will be the information about the device status and its battery discharge curve. Then, a more accurate power model may be estimated, since more data are be available. On the other hand, a higher sample frequency would impact on the device battery life: a trade off has then to be defined.

We empirically chose to log the device status every *ten seconds*. This is a higher bound from the model perspective: choosing a higher time interval might cause missing information about system dynamics (e.g. the user turning on the screen for a limited amount of time, a small data transfer operation, etc.). Lower time intervals have been empirically tested: sampling every one, two or five seconds have lead the MPower application to be listed between the most power consuming application detected by the

Android framework. With the chosen sampling frequency, we made sure that MPower is consuming less than 1% of the total system power, according to the Android statistics.

The log operation itself is performed by an Android `Handler` object, called *mHandler*: this is used to schedule "runnable" objects to be executed at some point in the future. At creation time, the LogService component creates and starts a new handler. This is in charge of collecting and storing information about the device state, then setting a timeout to be called after ten seconds: the logging loop is so implemented. Another handler, *mHandlerGraph*, has been developed to wake itself up every ten minutes: this executes the same task of the previous handler, but it is meant to provide information to the aforementioned TabsCharts activity, so graphs can be plotted. These information do not need to have a accuracy as high as the one required for the power model estimation phase, since it is used just to display the graphs.

File vs. SQLite storage

An important design choice has been made regarding the two data storing operations just discussed. A first straightforward approach to save information is to append the log string to a text file, while an alternative may be to store it directly on the local SQLite database. We chose the first approach for the *mHandler* log operations, since these are meant to provide high frequency information that will not be used on the client side. We only need to append a line to an file output stream, without the overhead needed to support the functionalities provided by the SQLite tool, e.g., transaction commits, indexes, queries execution, etc. Moreover, the SQLite database is implemented as a single big file on the filesystem, so writing on it every ten seconds is as expensive as writing on a simple text file. However, the second approach is used by the *mHandlerGraph*: this solution allows the TabsCharts activity to query and show just the data it needs when the user asks to see graphs activities. Log data and visualization data are then stored differently, since they have different purposes.

Data representation

A standard representation for the sampled data has then been defined: it has to be flexible and extendable, since mobile device platforms may have different hardware features and are continuously evolving (e.g., multicore processors and new network interfaces like Long Term Evolution (LTE) are not present in older device).

For this reason, the log files structure has been designed to be a collection of adjacent JavaScript Object Notation (JSON) file: the first level keys point out the category of the information (e.g., wifi stats, screen information, etc.), while the internal object structure contains the actual information. The structure is explained more clearly in Listing 1.1. However, to keep the log file as lightweight as possibile, we decided to skim it of all the useless blank spaces and newlines, and to have a single log for every line. This means that for every log on the app, cadenced every 10 seconds, we will have a line on the log file, with a different timestamp of course. This structure is shown in Listing 1.2. The whole file is then compressed and encrypted before being sent to the Meth server: more details about this procedures will be given in Section 1.1.3.

```
{
        "timestamp": 1357601640,
        "screen":
                {
                        "height":1184,
                        "width":720
                },
        "wifi":
                {
                        "tx_bytes":1927104,
                        "connected":1,
                        "wifi_rx_bytes":3545954},
        "bluetooth":
                {
                        "bluetooth_on":0,
                        "bluetooth_state":10
                }
}
```

Listing 1.1: A small extract of a JSON log file

```
{"timestamp":1357601640,"wifi":{"tx_bytes":1927104,"wifi_rx_bytes":3514...
{"timestamp":1357601650,"wifi":{"tx_bytes":2017124,"wifi_rx_bytes":3514...
{"timestamp":1357601660,"wifi":{"tx_bytes":2028804,"wifi_rx_bytes":3514...
{"timestamp":1357601670,"wifi":{"tx_bytes":2037104,"wifi_rx_bytes":3514...
{"timestamp":1357601680,"wifi":{"tx_bytes":2059904,"wifi_rx_bytes":3515...
```

Listing 1.2: An example of the real "one-liner" log file

Life cycle and wake locks

Another consideration regards the capability of the service to react to the unpredictable policies of the Android run time manager: as already discussed in Section **??**, the application must take into consideration the possibility of being killed in every moment of its life cycle, in order to free resources for applications with a higher priority.

A way to prevent the Android framework from killing the LogService would have been the acquisition of a *wake lock* during the whole logging period. The problem of this solution is that it greatly influences the system behavior, since it forces the Operating System (OS) to stay always on and waste a lot of battery power. We decided not to use wake locks except for the right instant of logging, releasing it as soon as the operation has finished. As a consequence, the information we log is not continuous in time because no data is being logged when the system goes in a *deep sleep* state: this is not a critical issue, since in that energy-saving state the power consumed by the device is really small and the power model can be computed ignoring this missing data.

The service is automatically restarted if it is killed by the Android framework, and it is registered to start when the device boots up.

Table 1.1: Events detected and corresponding messages sent

MESSAGE SENT	MESSAGE DESCRIPTION
gps_on	The GPS module has just been turned on
blue_on	The Bluetooth module has just been turned on
wifi_on	The Wi-Fi module has just been turned on
gps_off	The GPS module has just been turned off
blue_off	The Bluetooth module has just been turned off
wifi_off	The Wi-Fi module has just been turned off
msg_plugged	The device has been plugged into a power source
msg_wifi_connected	The device has just connected to a Wi-Fi network
msg_check_file	The log file has not been checked for a long time

State change detection

The LogService component is finally in charge of monitoring state changes, since it is continuously sampling the device status. As soon as a significant change in the current device status is being detected, one of the messages reported in Table 1.1 is broadcasted using the Intent message passing mechanism. Almost all the messages are self explanatory, while a brief consideration has to be done on the *"msg_check_file"* message. This is sent by the LogService about every hour in order to ask for a check of the file's size: as soon as the log file becomes too big, it is split in multiple parts, in order not to incur in additional overheads due to its excessive size.

The EventHandler will be in charge of handling them correctly, as explained in Section 1.0.2.