POLITECNICO DI MILANO

Facoltà di Ingegneria dell'Informazione

Corso di Laurea in Ingegneria Informatica

Technical Report

The MPower Sense Library

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

KVM Kernel-based Virtual Machine

LTE Long Term Evolution

MHE Mean Hourly Error

MVC Model-View-Controller

OS Operating System

SDK Software Development Kit

TTL Time-To-Live

VM Virtual Machine

Chapter 1

Sensing

Modern mobile devices have the availability of several hardware sensors, that can be used to sense both the internal and external device conditions. The operating system is then in charge of implementing a custom Hardware Abstraction Layer (HAL) for every hardware platform it is compatible with, in order to simplify their usage. Android itself provides a good hardware abstraction layer, but the implementation of a logging infrastructure like the one proposed in this work is far from trivial, since there are too many Android devices and too many operating system versions on the market.

Moreover, it is clear that from an energetic point of view these sensors have to be used consciously, since most or all of them require an energy consumption. The MPower app must be able to detect the current status of every hardware module present on the device, consuming the less power possible.

In order to develop a well organized and easily maintainable application code, we developed the *Sense libraries*. These Java classes, contained in the `org.morphone.sense` package, are meant to provide a single access point to the hardware of the device: they encapsulate all the logic needed to support different Android versions, following the Android best practices for an effective and efficient source code.

When possible, these libraries implement a kind of *cache layer* instead of just a hook

for the Android system Application Programming Interfaces (APIs). They record the status of the underlying hardware and update it only when it changes, providing a "cached" value to the caller instead of continuously polling the state of the device to gather state information.

This is made possible using Intents and BroadcastReceivers, two important concepts implemented by the Android Framework that we introduced in Section **??**. Applications can register Broadcast Receivers to listen for and react to one or more Intent types. In order to filter a particular class of events, Broadcast Receivers can be registered using specific Intent Filters. The Activity or the Service that instantiates this object (in our case, the LogService component) has to register the receiver in its onCreate method, unregistering it on its onDestroy method: this is a best practice suggested by the Android documentation.

The hardware features taken into consideration have been grouped into different categories. Table 1.2 has these categories in the header, lists all the data gathered from the devices in the underlying rows. Every category corresponds to a sub-package in the source code, with the same name. Every package contains:

- an *interface*, defining the information that the library has to provide to the caller. The use of an interface allowed us to develop different implementations of the same sensing library, in order to estimate which approach could be the most suitable to our purpose;
- the actual *"sense"* object, that implements the logic to provide those information;
- an *exception*, thrown when a particular information is not available or it has not been retrieved correctly. The caller will then be on charge of handling the exception, knowing that something went wrong and potentially warning the server with some information about the error.

At the time of writing, these libraries provides more information that those used

Table 1.1: Data gathered from the device

Screen	**Battery**	**CPU(s)**	**Mobile**
is_on	on_charge	max_frequency	state
brightness_mode	temperature	min_frequency	activity
brightness_value	voltage	current_freq	net_type
width	percentage	max_scaling_freq	signal_strenght
height	technology	min_scaling_freq	tx_bytes
refresh_rate	health	governor	rx_bytes
orientation		usage	call_state
		cpu_id	airplane_mode

Wifi	**Audio**	**Bluetooth**	**GPS**
is_on	music_active	is_on	is_on
is_connected	speaker_on	state	status
signal_strenght	music_volume		
link_speed	ring_volume		

to calculate the power model. This makes possible future works on the data logged during a long time period. A comprehensive description on how these measurements are performed is given in the following subsection.

ScreenSense

This package is meant to retrieve information about the device's screen. Two different implementations have been developed. The first one, SCREENSENSE, retrieves information invoking the Android APIs. The second one, SCREENSENSERECEIVER, extends the `BroadcastReceiver` class and is thought to prevent the LogService from doing useless polling requests.

The receiver listens for changes in the device screen state. When the device turns on, a local private variable is set to 1, otherwise it is set to 0. Every other measurement, like screen brightness mode or orientation information, is retrieved using the Android API only if the screen is on. Otherwise, a zero value is returned by the called method since the asked value is useless. This way, a huge amount of system calls is avoided when the screen is off: according to our measurements, this happens for the majority of the time and this policy has then a great impact in saving battery power.

Some information, like display height and width, are initialized when the object is first created, since they do not change during the device life cycle.

Particular attention had to be paid when retrieving screen brightness information. When the screen brightness mode is set to "manual", the screen brightness value can be retrieved through the Android Content Resolver API. Otherwise, if it is set to "automatic", the information provided by the API is no more valid and the current screen brightness is not measurable from any background component. A solution to this is to start a fake activity every time we need to know the actual brightness value, but this is not feasible since it is too power consuming. That's the reason why we have to look for the current brightness value in a file stored on the device filesystem, only when we

detect automatic brightness and the screen is on. The problem here is that this file is placed in a different path, depending on the hardware platform and the Android version. A recursive approach has finally been studied and implemented to look for this file when the ScreenSense is first created, saving its real path in a local private variable: the brightness value can then be read from that file when needed.

BatterySense

Information about the battery status is often coarse grained, since most devices do not include a physical measurer about electrical current or more precise hardware sensors. The only information exposed by the system to the developers are the battery percentage, voltage, temperature and an indicator of the battery health.

This package is meant to retrieve these information. The current implementation is actually a class that listens for battery status change events.

When a battery status change event occurs (e.g. the battery decreased its amount of charge, the device has been plugged into an electrical appliance, etc), the BatterySense is waken up and the local private variables of the class are updated. These variables records the current state of the device's battery: when a log operation is performed, there's no need to call the Android APIs, since BatterySense stored the actual state of the battery and change it only when a change event occurs.

A final consideration regards the Android APIs. It does not provide developers instruments to understand how much an application, a component or the whole system is consuming, nor it is possible to know this consumption *a priori*. Starting from version 2.3, Android has added some statistics for the final user, i.e., a battery usage report that can be accessed through the system settings menu. In this screen are listed all the applications and services that have consumed at least the 2% of the battery life, since last full battery recharge. A graph is is also displayed, showing the battery discharging over time and for each listed application some useful data are reported. However,

these data cannot be accessed programmatically by non-built-in-system applications, strongly limiting their usefulness to developers.

CPUSense

This package is meant to retrieve information about the device's processors. The current implementation relies on the framework API or on direct read operations of the Android system files.

This is the case of maximum and minimum working frequencies or the CPU governor, whose values are retrieved for every core when the CPUSense is first created: these are stored in local private data structures, in order not to perform future accesses to files when these information are requested.

Unfortunately for most Android devices, the APIs provided by the framework to obtain the number of virtual processors doesn't work properly. Even looking in /proc/stat does not always show the correct number of CPU cores. The only reliable method to determine the number of cores is to enumerate the list of virtual CPUs present in the system folder. Current working frequency is then provided reading directly system files that map CPU resources.

Finally, the current core usage estimation is performed reading from the file /proc/stat the amount of time a core has spent performing a particular kind of work.

MobileSense

This package is meant to retrieve information about the state of the mobile data module. The current implementations makes use of the Android TelephonyManager service, extracted from the application context used to initialize the class.

As explained for other sensing libraries, values that will not change during the device life cycle are initialized when the MobileSense object is created. An example is the International Mobile Equipment Identity (IMEI) value, a number that uniquely

identify a device with a GSM module.

MobileSense then registers a listener for changes in GSM signal strengths, mobile data activity, mobile connection and GSM call states: as already discussed, this behavior prevents the continuous polling of the status of the device, updating it only when it changes. The listener, called `MyPhoneStateListener`, extends the `PhoneStateListener` class. Local variables are then updated only when they really change.

Finally, information about data transferred using a mobile network are retrieved using the `android.net.TrafficStats` class.

WifiSense

This package is meant to retrieve information about the device's WiFi module. Two different implementations have been developed. The first one, called WIFISENSE, retrieves information doing simple polling requests to the Android API. The second one, called WIFISENSERECEIVER, is a Broadcast Receiver.

The behavior is similar to the one implemented in the ScreenSense class: the receiver listens for changes in the connection state and updates local private variables when needed. If the device is connected to a WiFi network, the other WifiSense methods perform their measurements, otherwise they return not valid values. Again, the idea is to avoid API requests as soon as the measurements are not needed, in order to save battery power.

A trivial trick has to be implemented to get the WiFi MAC address: this code uniquely identify the hardware module and it is retrieved and stored in a local private variable when the WifiSense is created. This is not available when the WiFi module is not enabled: the WifiSense has then to enable the WiFi module at its creation in order to get that address, then disabling the module instantly

AudioSense

This package is meant to retrieve information about the audio state of the device. The current implementations makes use of the Android AudioManager service, extracted from the application context used to initialize the class. It is a simple wrapper component and no additional considerations have to be done.

BluetoothSense

This package is meant to retrieve information about the device's Bluetooth module. Two different implementations have been developed. The first one, named BluetoothSense, makes use of the BluetoothAdapter class, provided by the Android framework: this is probably the most straightforward way to access Bluetooth information, but has the drawback of being a time-consuming (and potentially power-consuming) operation.

A second implementation, named BluetoothSenseReceiver, has then been developed using the aforementioned Broadcast Receiver approach. A status change event (e.g. the Bluetooth module has been turned on, a peer device has been connected, etc) will wake up the BluetoothSense component, that will update the private variables of the class, meant to store the actual state of the bluetooth module. No more calls on the BluetoothAdapter class are then needed.

LocationSense

This package is meant to retrieve information about the state of the GPS module. The current implementations makes use of the Android LocationManager service, extracted from the application context used to initialize the class.

At its creation, LocationSense checks if the GPS provider (and the corresponding module) is currently enabled and initializes its local variables. Then register a listener

for GpsStatus change events: as already discussed for the other libraries implementations, this behavior prevents the continuous polling of the status of the device, updating it only when it changes.

DeviceSense

This final package is meant to provide generic information about the device.

System properties like kernel version and device's architecture are retrieved from the System class, while other information like device model, brand and SDK are obtained using the android.os.Build class. These are static information send just at the first interaction with the Meth server.

An important consideration has to be made on the getDeviceId method: it is meant to provide a string that can uniquely identify the device hardware platform. The first attempt is to identify the device with its 15-digits IMEI number, but some mobile devices (i.e. some low cost tablets) do not have a GSM module installed. In such situations, the 12-digits MAC address of the WiFi module is then used, with the "MAC" word put in the front of it in order to have identifiers with the same length. This solution is reasonable, since every Android device on the market has at least a WiFi module.

1.0.1 Actuators Libraries

The Actuators Libraries are a thin layer of code which translates the Android API calls into simpler methods. The `org.morphone.actuators` package contains a Java class for every hardware module whose state can be changed.

The list of the actuator classes with their methods is presented in Table 1.3. For every actuator, a constructor method is in charge of initializing a local private variable with the Android API object needed to perform the actuation. The other methods are self explicative. No particular design choices have been made, since these are just wrapper classes.

Table 1.2: Actuator classes and methods description

CLASS NAME	METHODS
BluetoothActuator	BluetoothActuator()
	switchBluetoothOn()
	switchBluetoothOff()
DataActuator	DataActuator(android.net.ConnectivityManager)
	switch3GOn()
	switch3GOff()
LuminosityActuator	LuminosityActuator(android.content.ContentResolver)
	changeBrightness(brightnessValue)
WifiActuator	WifiActuator(android.net.wifi.WifiManager)
	switchWifiOn()
	switchWifiOff()

A little trick has been studied to force the change of the screen brightness. A simple hidden `Refresh` activity has been developed: its only task is to close itself after its creation. The activity is started just after the `changeBrightness` method is called: in this way, Android force the refresh of the screen brightness, without the user even noticing.

A *GPS actuator* had been developed but toggling GPS on/off is no longer possible as of Android 2.3.3. The same happened for *Airplane mode*: that operation was allowed only for Android versions before Android 4.2, unless the device has been rooted.

1.0.2 Data helpers

The last components presented in this chapter are those related to data management. The first one, called **DATABASEHELPER**, extends the `SQLiteOpenHelper` and it is meant to manage the creation and the accesses to the MPower SQLite database on the device. Inspired by the name of the first, we then called **FILEHELPER** the component in charge of handling the operations on the log files.

DatabaseHelper

In addition to the standard methods that handle the creation and the management of the database structure, the DatabaseHelper provides two main functionalities to the upper layers.

Time-To-Lives (TTLs) table access — As already told in Section 1.2.2, TTL information is stored on the device with the form of a lookup table. Its header and a trivial example of content is given in Table 1.4.

The first six columns contain information about states of the hardware module, i.e. the Wi-Fi, GPS, mobile network and the Bluetooth state, the screen brightness level and the airplane mode. The following one hundred columns contain the estimated TTLs (in

Table 1.3: TTL lookup table example

wifi	**gps**	**screen**	**mobile**	**bluetooth**	**airplane**	**1%**	**2%**	...	**99%**	**100%**
off	off	low	on	off	off	4	12	...	987	1031
...	...	...	...	...	...	...	...	...	...	...
on	on	medium	off	off	off	1	4	...	598	632
...	...	...	...	...	...	...	...	...	...	...
on	off	medium	off	off	off	1	4	...	798	815

minutes), one for every battery level (in percentage). We then have one row for every observed configuration.

As a consequence, given a certain device configuration, the TTL computation task is nothing more than a SQL query that selects the content of the column corresponding to the current battery level (e.g., the highlighted column), with some constraints on the states of the hardware modules (e.g., the highlighted row). In this way, no additional computation is performed on the device.

Given a certain battery level, the maximum and the minimum TTLs are the maximum and the minimum values in the column corresponding to the current battery level. As soon as the user specifies more hardware constraints in the PowerModelActivity, these take part in the *where* clause of the query and the compatible configurations will be the one contained in the result set.

Generic information tables — As already told in Section 1.3.1, high detailed log data for model computation is stored on a log file, while less detailed information is stored on the SQLite database in order to be shown by the TabsCharts activity. A simple table stores information about battery levels and the hardware activity of the GPS, Bluetooth, Wi-Fi and Mobile network modules. Every row corresponds to a log sample in a

specific instant. Service methods are then provided to insert a sample and extract data of interest: these implement simple queries and extract from the table just the data needed to be shown, i.e., the data related to a certain hardware module in a certain period of time.

Two more tables are then used to store reports information (e.g., last time data was sent to the server or the device fallen into a deep sleep state, etc.) and the suggestions history, presented in Section 1.2.3. Again, service method are provided in order to insert and query data from these tables.

FileHelper

This wrapper component provides two main functionalities.

The first one is the *gunzip* compression [**?**], that is applied on the log files. Gunzip, as reported in their website "*finds duplicated strings in the input data. The second occurrence of a string is replaced by a pointer to the previous string, in the form of a pair (distance, length)*" [**?**]. Our log files are composed of multiple identical occurrences, since all the JavaScript Object Notation (JSON) objects written in the file have the same keys. This operation is done before encrypting the files, to gain the maximum compression ratio.

The second functionality provided by this component is the file encryption. This is done using the AES-256(CBC) symmetric algorithm using the previously exchanged token as the common secret key and an Inizialitation Vector (IV) randomly generated for every file, to ensure the best level of security.