Android Internals and Telephony

A. J. Singh¹, Akshay Bhardwaj²

¹Professor, ²Student, Computer Science, HP University, Shimla, India

Abstract—There are so many books and paper publications about Android and related applications but only few are related to Android telephony. The topics which are covered here like Android internals, Logging and Debugging mechanism for Android powered devices are important with respect to Android since enough information is not available about them. Our effort here provides certain framework to these practically relevant topics and their use while debugging, analyzing and fixing anything related to Android. This research will be extremely helpful to foster understanding among new android developers about Android Internals as well as everybody else who desires a general understanding of the internal working of an Android powered Smartphone.

Keywords--android, telephony, logging, debugging.

I. INTRODUCTION

Android is an open-source Operating System created for mobile phones and other devices led by Google. In 2003, Andy Rubin and his team invented Android [1]. Later in 2005, Google acquired the Android operating system. Android is a software environment built for mobile devices. It is not a hardware platform. Android includes a Linux kernel-based OS, a rich UI, end-user applications, code libraries, application frameworks, multimedia support, and much more [1]. On the 5th of November in 2007, distribution of Google’s version was announced with the founding of the Open Handset Alliance. Open Handset Alliance is a group of device manufacturers, Chipset Manufacturers and Mobile Carriers. Device Manufacturers include companies like HTC, LG, Samsung, Motorola and Sony. Chipset makers are Qualcomm, Texas Instrumental and Intel. Mobile Carrier companies are represented by companies like Verizon, Sprint, AT&T, T-Mobile, MetroPCS and lots of other companies all over the world [2]. Most of the Android platform is released under Apache 2.0 license. Google Android released the entire source code under an Apache license [1]. With Apache license, a user can freely download and use Android for personal and commercial purposes. It allows user to make changes to original software without having to contribute to the open source community.

1.1 Boot Up Process In Android

The boot up process can be described as the process of powering on the device [3]. During boot up process device goes through these various stages [4].

1.1.1 Stage 1: The Android Boot Process from Power On

When an Android powered device boots up, Boot ROM code initialize the hardware and locate the boot media with the help of Central Processing Unit (CPU). When device hardware is initialized, ROM code finds the boot media. It also copies the boot loader into internal RAM. After this step, the process of boot loader is started. Figure 1.1 explains the first stage, when device is booting up [3].

1.1.2 Stage 2: The Boot Loader

Boot loaders (see Figure 3.2) are combinations of instructions to boot any operating system kernel, generally speaking a boot loader is a small program that loads the operating system into the computer’s memory when the system is booted, and also starts the operating system. In Android the BL (Boot loader) is a program which is special in its own ways. This is because Boot Loader is separate from Linux kernel, which is used to setup initial memories and load kernel to RAM [4].
Figure 1.2 explains the third stage of Boot loader. Details of each stage are explained below:

During step-1, Initialization of boot loader takes place. Boot loader sets up External RAM for further processing.

In step 2, Boot loader makes sure when external RAM is available and system is also ready to run specific process.

In step 3, the first major program will run. It includes important activities from setting up additional memory, to setting up file system. In an Android powered smartphone, it also supports loading code for the modem CPU and low level security aspects and memory protections as well.

In step 4, Boot loader will look for a Linux Kernel when a specific task is over for boot loader. As per requirements of device manufacturer, it will load this from the boot media and place it in the RAM. It will place certain boot parameters in memory for the kernel when it initializes.

Finally, Linux Kernel will start its operations.

1.1.3 Stage 3: The Linux Kernel

Linux Kernel is a heart of the Android operating system, the whole process remains the same, as the Linux kernel starts up, and it sets up all the necessary initializations which are needed by the system. For instances, it initializes the interrupt controllers, sets up memory protections, caches and schedule processes. In stage -4, the Linux kernel initializes all the memory units. Once this is done, the system is ready to execute the user space processes.

User space processes refer to the processes that are executed in the user space e.g. Applications, user data. To launch the user space processes, kernel must execute init process first. This is found in system/core/init. The init process initializes all the user space processes.

1.1.4 Stage 4: The Init Process

The init process is the "head of all the processes". This is because all other processes in the system are launched from this process or they are one of its descendants.

Init.rc: The init.rc is a script that is looked up by the init process. The init.rc script is placed in system/core/rootdir in the Android open source project. This script describes all kinds of system services, file systems, and the parameters that need to be set up. The init process parses this script and launches the system service processes.

1.1.5 Stage 5: Zygote and Dalvik

Once the init process launches the system services, then it starts launching the zygote process. This means it will just start, initialize and execute the Dalvik Virtual Machine (DVM).

1.1.6 Stage 6: The System Server

The first java component is the system server that will run in system. It is responsible for starting Android services such as telephony manager and Bluetooth. In the Android operating system start up of each service is currently written directly into the run method of the system server. The system server source can be found in the file/frameworks/base/services/java/com.android/server/SystemServer.java in the open source project.

Figure 3.3 explains the Interaction of system server with the File System.

User space processes refer to the processes that are executed in the user space e.g. Applications, user data. To launch the user space processes, kernel must execute init process first. This is found in system/core/init. The init process initializes all the user space processes.

1.1.4 Stage 4: The Init Process

The init process is the "head of all the processes". This is because all other processes in the system are launched from this process or they are one of its descendants.

Init.rc: The init.rc is a script that is looked up by the init process. The init.rc script is placed in system/core/rootdir in the Android open source project. This script describes all kinds of system services, file systems, and the parameters that need to be set up. The init process parses this script and launches the system service processes.

1.1.5 Stage 5: Zygote and Dalvik

Once the init process launches the system services, then it starts launching the zygote process. This means it will just start, initialize and execute the Dalvik Virtual Machine (DVM).

1.1.6 Stage 6: The System Server

The first java component is the system server that will run in system. It is responsible for starting Android services such as telephony manager and Bluetooth. In the Android operating system start up of each service is currently written directly into the run method of the system server. The system server source can be found in the file/frameworks/base/services/java/com/android/server/SystemServer.java in the open source project.

Figure 3.3 explains the Interaction of system server with the File System.
1.2 Introduction To The Attention Commands (At Commands)

AT commands are strings of data which begin with the prefix AT and can be sent to a communication module for the purpose of programming [5]. In other words, to communicate with Radio Interface Layer (RIL) which is the modem or in simpler words the base band side of the phone we need the AT commands. AT commands are related to asynchronous serial interface and we can use Hyper Terminal tool/software in Windows to send it. If module is connected to Personal computer/machine using RS232 (Electric Standard), then also using RS232, can also send the message [5]. AT commands were introduced in 1977 by Hayes Communications Company for dial-up modems [5]. AT Commands should communicate with many different areas. Overall, AT commands are used to control Modem functionality. The supported categories include:

Some Important AT commands:

- **AT+CGMI**: Request Manufacturer Identification: execution command returns the device manufacturer identification code [6].
- **AT+CGMM**: Request Model Identification: execution command returns the device model identification code [6].
- **AT+CGMR**: Request Revision identification: Execution command returns device software revision number [6].
- **AT+CGSN**: Request Product Serial Number Identification: Execution command returns the product serial number, identified as the IMEI (Identity Mobile Equipment Identity) of the mobile [6].
- **AT+CIMI**: Request International Mobile Subscriber Identity: Execution command returns the value of the Internal Mobile subscriber Identity stores in the SIM (Subscriber Identity Module) [6].
- **AT+CHUP**: Hang Up Call: Execution commands cancel all active and held calls, also if a conference call is going on [6].
- **AT+CRS**: Cellular Result Code: set command controls whether or not the extended format of incoming call indication is used [6].
- **AT+CNUM**: Subscriber Number: Execution command returns the MSISDN (Mobile Subscriber Identity Number) only when the phone number of the device has been stored in the SIM card [6].
- **AT+CCFC**: Call Forwarding Number and Condition: Execution command controls the call forwarding supplementary service [6].
- **AT+CCWA**: Call Waiting: Set Command allows the control of the call waiting supplementary service like activation, deactivation and status query [6].
- **AT+CLCC**: List Current Calls: Execution command returns the list of current calls and their characteristics [6].
- **AT+CPAS**: Phone Activity Status: Execution command reports the device status [6].
- **AT+CPIN**: Enter PIN: Set command sends to the device a password which is necessary before it can be operated like SIM pin [6].
- **AT+CSQ**: Signal Quality: Execution command reports received signal quality indicators [6].
- **AT+CPBS**: Select Phonebook Memory Storage: Set command selects phonebook memory storage, which will be used by other phonebook commands [6].
- **AT+CBC**: Battery Charge: Execution command returns the current battery charge status [6].
- **AT+CSMS**: Select Message Service: Set command selects messaging service. It returns the type of message [6].
- **AT+CMGL**: List Messages: Execution commands reports the list of all the messages with status value stored into message storage [6].

1.3 Radio Interface Layer In Android

Above we have a briefly described the AT commands, and now let’s have a look at the RIL. There is a Radio Layer Interface in Android and it is about telephony service.

Figure 1.4 [7] gives us a snapshot of the calling flow mechanism of Android. It displays vital components of RIL, which is the heart of any smart phone for substantial calling and Data activities.
Below I have tried to explain the general working of telephony framework when communicating with modem in the Android platform:

1. The modem receives all the inbound requests and processed data delivered by the framework through the RIL [8].
2. The RIL represents the Radio interface integrating modem and Android telephony [8].
3. Certain inbound events concerning network originating at the modem are also interfaced by the RIL with the Telephony framework [8].

The RIL has two parts known as RIL daemon and vendor RIL. RIL daemon is the interface between Android telephony framework and vendor RIL. All requests originating to and from the RIL are stored in one interface file RIL.java which is a part of the Android Telephony network. Customized implementations for the vendor RIL as implemented by each vendor are stored in a shared library. Modem specific vendor RIL are usually used by the vendor.

Basic Boot up sequence with reference to telephony is as under:

1. Android telephony framework is started as soon as the phone is powered on and the phone processes begin.
2. A socket connection is made between the RIL daemon and framework followed by the ril daemon search for the vendor ril path from the system property.
3. Vendor RIL is then loaded as the shared library.
4. The RIL_INIT method is employed by the vendor RIL by calling it during its initial start.
5. Telephony framework receives references for all vendor RIL functionalities from the RIL daemon, which does so, by calling the RIL_Register

**RIL Interaction:** There are two forms of communication that the RIL handles:

**Solicited commands:** Solicited commands comes from the upper application framework layer like dial up, sending SMS and read/write contact/SMS from SIM.

The RIL daemon fetches the all commands from application framework layer to vendor RIL. Then, with the help of inter process communication/AT commands, vendor RIL interacts with modem protocol stack. In respond, modem sends the commands to RIL vendor through IPC/AT commands. Then RIL vendor send the response to application framework [8].

**Unsolicited Commands:** The RIL can receive these types of commands/events like receiving SMS, and network state changes at any time from the radio hardware. These are dispatched by the RIL and sent back to the Android framework. For instance, receive an incoming call (from RIL-daemon to telephony service) [8].

The interesting part here to check is how does the incoming call ring? To understand this we need to understand two term, MO and MT. MO stands for mobile originated call, which means the call is originated by the mobile station, or in other words you make a call.

The MT call stands for mobile terminated call, which means the call is an incoming call from a different mobile station or a base station.

For an unsolicited call, once the RILd determines the state of the call, it uses the comprised set of API’s which is the AT command (AT+CLCC) to check for any incoming calls. The function of this AT command is to return the list of MT calls. All communication with radio hardware is processed by the radio specific RIL vendor. Unsolicited commands are then used to call RIL Daemon (RILd) by the radio specific RIL vendor. The vendor RILd receives unsolicited commands by the Radio (protocol stack) through Inter Process Communication (IPC).

2.1 **Introduction**

Issues need to be addressed and fixed for a technically sound software platform and a better user experience. Therefore, from the troubleshooting development perspective, logging is a very important tool. The Following paragraphs describe about the anatomy and severity of logs.
They also provide details of what information is contained within these logs, methods to pull relevant logs from the device, filtering required information from these logs and other such details.

Logging is the method to detect different kinds of issues. In its simplest form, logs allows developer to deduce the root cause of an issue based on the activity running in the background and see the exact time the issue take place. After getting all the required information, the developer fixes the issue for the next software version. This subsequently, produces a good user experience subsequently and a low priced product for the end user.

Android based Smartphone contain Application side and the Baseband side. Application side supports components such as camera, messaging, call, etc., and Baseband related components include antenna, modem, etc. Android can only provide for Application side logging capabilities as the Baseband logging is usually proprietary to the respective manufacturing companies like Qualcomm INC., Intel, etc. To reduce the complexity of data and the logging nuances associated with Baseband logging we will keep our logging methods focused on basic Android logs and related basic logging tools.

2.2 Tools Used To Capture Logs

This section describes the Android tools and their variants to capture logs. Logs are real-time monitoring records running in the background which are stored in files on the phone while the user performs activities. When an unusual behavior occurs, it is recorded in real-time on the phone. After debugging and analysis of the logs, the developer can see the root cause of the issue, the time when the problem occurred, which application caused the issue, as well as what other processes were running at the time when issue occurred. There are several tools and debuggers used in business but the basic tools in Android are the Android Debug Bridge (ADB) and Dalvik Debug Monitor Service (DDMS).

2.2.1 Dalvik Debug Monitor Service (DDMS)

This is a tool which comes along with the Android Software Development Kit (SDK) Dalvik Debug Monitor Server (DDMS) is also integrated in eclipse. It provides screen capture service, thread information of device, logcat process, radio information, call and short message service spoofing and so many other functions. DDMS works with both emulator and a connected device. If both are connected and running simultaneously, DDMS defaults to emulator. In Android, each application runs its own process on its own virtual machine (VM). Android Debug Bridge (ADB) is connected to Dalvik Debug Monitor Server (DDMS), when it starts which basically means when a device is communicating to a machine, one type of service called as Virtual Monitor is created between ADB (Android Debug Bridge) and Dalvik Debug Monitor Service (DDMS). It is responsible to notify when application on the device is started or terminated [9].

2.2.2. Android Debug Bridge (ADB)

It is a powerful command based utility which allows communicating with an emulator instance or connected Android device. It has three commands. First is a client which runs on development machine, and second is a server which runs in a background. This manages the communication between client and ADB daemon and third is a daemon which runs as background process in each device instance. Android Debug Bridge (ADB) is also responsible for managing the current status of a device, running shell commands on it, managing port forwarding, and copy files to and from device [10].

Figure 2.1 shows us the can work flow of ADB. First, ADB client issues the command from the machine to start the ADB server. Then the ADB server set up the communication or connection with Android powered device by scanning the port. Finally, after the connection is established, ADB commands can run on the system.

Fig. 2.1 Workflow of ADB

2.2.3 General ADB Commands

Installing an Application:

adb install <path_to_apk> (this command install files to /system/app or /data/app) [10].
Copying Files to or from a Device Instance:

Unlike the install command, which only copies .apk file to a specific location, the pull and push commands let you copy arbitrary directories and files to any location in an emulator/device instance [10].

To copy a file or directory (recursively) from the device, use:

```
adb pull <remote> <local>
```

To copy a file or directory (recursively) to the device, use:

```
adb push <local> <remote>
```

Stopping the adb Server:

if adb does not respond to a command, you can terminate the server and restart it and that may resolve the problem [10].

To stop the adb server, use the adb kill-server. You can then restart the server by issuing any adb command [10].

Logcat:

The Android logging system provides a mechanism for collecting and viewing system debug output. Logs from various applications and portions of the system are collected in a series of circular buffers, which then can be viewed and filtered by the logcat command. You can use logcat from an ADB shell to view the log messages [10].

2.3 Methods To Capture The Logs

In this section we explain the methods to capture the logs from a device. There are two ways to capture the logs:

- Dalvik Debug Monitor Server (DDMS) ADB commands.

Android provides lots of facilities for debugging purposes. To capture the logs, we have to connect the mobile device to a laptop using USB Cable. Other thing we need to make sure is that the device has “USB Debugging” option enabled.

2.3.1 Capturing Logs Using DDMS

Two of the most used options in DDMS are Screen Capture and Logcat logging.

2.3.1.1 Screen Capture

We can capture the screen on the Mobile Device using screen capture option in Dalvik Debug Monitor Server (DDMS). This software is located in the Android SDK. As shown in Figure 5.1 (p. 33), it is under Device->Screen Capture. This will capture whatever is currently displayed on the Mobile Device. This can be saved as .png or .jpg file for debugging.

After looking at the saved image, a developer will know what problem was encountered, the exact error pop-up message, and UI distortions [9].

Figure 2.2 shows the screen capture option using the Dalvik Debug Monitor Server (DDMS).

Fig. 2.2 DDMS

2.3.1.2 LOGCAT

After connecting the android device we need to select device-> logcat option to see what processes are running at the time and, exactly what activities are going on at a certain time. When a Logcat process runs in Dalvik Virtual Monitor Server (DDMS), logs are running in the background of the device for different applications and are collected in a particular buffer.

Figure 2.3 shows sample logs captured by Logcat process. This is helpful to developers to understand the exact nature and root cause of the problem.

Fig. 2.3 Logcat samples

2.3.2 Capturing Logs Using Android Debug Bridge (ADB)

There are many types of logs but main log, radio log, kernel log, and event log are the basic logs which are very important for debugging and analysis of abnormal behaviour of the device [11]. We will see how to filter log output and use commands to capture the logs.
2.3.2.1 Filtering Log Output

Each Android log message has a tag and priority associated with it. The tag of a log message is a short string indicating the system component from which the message originates. The priority is indicated by one of the following character values, in order from the lowest to the highest priority:

V:Verbose (lowest priority): Zero level information that end user can identify
D:Debug
I:Info
W:Warning
E:Error
F:Fatal
S:Silent (highest priority, on which nothing is ever printed)

2.3.2.2 Commands To Capture The Logs

1. Main Log: The first basic log in the buffer memory is used to reflect activities that are running in the phone. Main log has information related to applications. It contains basic information only. Consider one example: a user switches to the camera and takes one photo, then press home key. Main logs contain all the information related to camera [11].

Command: adb logcat –b main>main.txt

2. Radio Log: This log contains the information related to telephony/radio messages. It will provide some more information for baseband/modem side. Radios log contains any network side issue or call related problems. Consider one example: A user is browsing websites on the device, and suddenly the device fails to open a web page. At that time Radio log contains the information about network loss [11].

Command: adb logcat –b radio>radio.txt

3. Kernel Logs: It provides the information related to kernel side activities. It contains important information about the communication between kernel layers, modem and application framework in Android device. Consider one example: navigation running on the device and it crashes unexpectedly. At that time kernel log is helpful to analyse the issue [11].

Command: adb logcat kernel>kernel.txt

4. Event Log: It provides information about launched or finished activities. It contains the information related to any event which occurs in the phone. Consider an example: a user change the mode of the device from landscape to portrait, but mode does not change. Event log contains all the information related to it [11].

Command: adb logcat –b event>event.txt

2.4 Commonly Seen Android Errors

In this section, I will list some issues that are encountered during our daily use of Smartphone. Our objective here is to answer the most commonly observed issues related to network, applications and other core internal systems like the kernel.

2.4.1 Call Drops

We always have call drop while in transit in urban or sub urban areas or even in stationary conditions. Now my objective here is to educate the user how these errors actually look when we extract a log. We will discuss more about log variants in our next section, but let’s focus now on the above issue.

Device using regular network:

```
09-19 04:39:17.327 D/RILJ (480): [UNSL]<
UNSOL_RESPONSE_CALL_STATE_CHANGED
network
09-19 04:39:17.334 D/RILC (153): rild: grabPartialWakeLock, s_wakeup_lock_flag -1
09-19 04:39:17.334 D/RILJ (480): [UNSL]<
UNSOL_RESPONSE_CALL_STATE_CHANGED
09-19 04:39:17.334 D/RILJ (480): [6677]>
LAST_CALL_FAIL_CAUSE
09-19 04:39:17.358 D/RILJ (480): getDlcRequestId
```

The above log gives me a description of the dropped call due to the network error.

The point that arises is how do we know about this network issue. AT commands, RIL, Rild (ril daemon), unsolicited, and solicited commands help in solving this matter. If we look at log snippet then we can observe that the call got disconnected from the network was in unsolicited state, and immediately the rild came in action and caused a wake to the device; by wake here we meant the backlight of the device turns on:
09-19 04:39:17.358 D/RILJ ( 480): DLC [2]: Tx [6677] sent = 77 answered = 76 Tout = 0
- - - LAST_CALL_FAIL_CAUSE: 13000ms 09-19 04:39:17.358 D/RILC ( 153): [6677]>
LAST_CALL_FAIL_CAUSE 09-19 04:39:17.358 D/RILC ( 153): lge-ril:
runRequestFunction(LAST_CALL_FAIL_CAUSE)
s_InitMutex LOCKED, dlc index - 2,
requestInterface(LAST_CALL_FAIL_CAUSE) invoking

The log shows that communication between Vendor RIL and Ril daemon got interrupted and created the call failure error. Above code in red color is generating the error report for temporary failure of network.

2.4.2 Force Close on Gmail Application

This issue relates to Gmail force close. It shows that FATAL exception has occurred in main thread. Also, Runtime Exception fails to destroy the background activity and enough memory is not available. It causes the Gmail Force Close. The issue clearly tells us by looking at the logs that the Gmail widget has gone through a fatal exception in the java.lang package which in turn caused the exception to arrive for java.lang.NullPointerException.

{com.google.android.gm/com.google.android.gm.GmailActivity:
java.lang.NullPointerException 07-23 21:45:48.726 E22165 AndroidRuntimeCaused by:
android.widget.AbsListView.setItemChecked(AbsListView.java:1411) 07-23 21:45:48.726 E22165 AndroidRuntime
com.google.android.gms.ConversationListFragment.onDestroyed(ViewConversationListF
ragment.java:266)

2.4.3 Black Screen on Device

Sometimes while using the Smartphone, we get the black screen on the device. There are certain reasons for black screen like multiple activities are running at a time, or heavy application is running in the background of the device. Below logs contain the information about black screen issue while using the phone and playing Pandora radio at same time.

05-12 12:49:57.579 I/ActivityManager (245): process name to start: com.android.vending
- > shows that the device is in low memory state that the LMK(Low Memory Killer) kills the processes with low adjs down to 4, which tells that the low memory state is quite severe
<br>29770.487224] select 24727 (equicksearchbox), adj 7, size 4122, to kill
<br>29770.487254] send sigkill to 24727 (equicksearchbox), adj 7, size 4122
<br>29771.923986] select 24708 (vvm.application), adj 7, size 3967, to kill
<br>29771.924016] send sigkill to 24708 (vvm.application), adj 7, size 3967
<br>29778.548599] select 957 (obile.bonusapps), adj 4, size 3934, to kill
<br>29778.548629] select 966 (angable.weather), adj 4, size 7840, to kill

The log shows that the device was in low memory state, and it caused Low memory killer to kill the running processes frequently, which caused the black screen symptom in this issue. Since the device has limited resource, and the Pandora radio uses quite a lot of memory when it's running, the device may get to low memory state more easily when it has the Pandora radio app on its background. All one can do is extra apps to ensure this issue doesn’t persist.

2.4.4 Application not Responding on Facebook (ANR)

Application not responding is the common issue seen on every smartphone. When any application is not giving proper response within about 5-7 seconds, ANR pop up is generated on that application. Please see below an example of Application not responding error on Facebook:

08-25 12:16:50.797 E/ActivityManager (436): ANR in com.facebook.katana
08-25 12:16:50.797 E/ActivityManager (436): Reason: Executing service
com.facebook.katana/.service.FacebookService
08-25 12:16:50.797 W/ActivityManager (436): Killing ProcessRecord(45309190)
28175:com.facebook.katana/10022): background ANR
08-25 12:16:50.847 I/ActivityManager( 436): Process com.facebook.katana (pid 28175) has died.

We can see ANR is generated in com.facebook.katana in first line. Android memory management gives the priority to all the processes. As shown in above snippet, Activity manager kills the process to prioritize, and ANR occurs on the device.

II. CONCLUSION

The Telecom industry is a mercurial entity. The smartphone era started with a loud bang, and it has made the users of formerly used feature phone devices consider switching to this advanced technology. Android started with a 2% market share in 2008, and now it has more than 60% smartphone users. Android provides an extremely user friendly experience with a very supportive ecosystem of applications being continually updated and enhanced by developers from the parent company as well as other extension developers all around the world. There are certain inadequacies which I have come across as a user and researcher, but they could be subjective. There are certain features and enhancements which I would expect to be included in future versions, and new ones might be added as they become available or, rather invented.

My research covers the basics of Android, its functioning as a system and logging and debugging mechanism. In conclusion, Android’s future is happening now, and further enhancements to this already diverse and this multifunctional platform will only take software engineering to the next level.

REFERENCES


