

# Google Android - A Comprehensive Introduction

Technical Report: TUB-DAI 03/09-01

Hans-Gunther Schmidt, Karsten Raddatz, Aubrey-Derrick Schmidt, Ahmet Camtepe, and Sahin Albayrak

March 16, 2009













DAI-Labor der Technischen Universität Berlin

Prof. Dr.-Ing. habil. Sahin Albayrak

Technische Universität Berlin DAI-Labor

Institut für Wirtschaftsinformatik und Quantitative Methoden

Fachgebiet Agententechnologien in betrieblichen Anwendungen und der Telekommunikation

Sekretariat TEL 14 Ernst-Reuter-Platz 7

10587 Berlin

Telefon: (030) 314 74000 Telefax: (030) 314 74003

E-mail: Sekretariat@dai-labor.de

WWW: http://www.dai-labor.de

# **Disclaimer**

This Technical Report has been kindly supported by DAI-Labor Technische Universität Berlin, Prof. Dr.-Ing. habil. Sahin Albayrak and was produced as a result of the university project called course "Autonomous Security II".

## **Abstract**

Google Android, Google's new product and its first attempt to enter the mobile market, might have an equal impact on mobile users like Apple's hyped product, the iPhone. In this Technical report we are going to present the Google Android platform, what Android is, describe why it might be considered as a worthy rival to Apple's iPhone. We will describe parts of its internals, take a look "under the hood" while explaining components of the underlying operating system. We will show how to develop applications for this platform, which difficulties a developer might have to face, and how developers can possibly use other programming languages to develop for Android than the propagated language Java.

# **Contents**

1	Introduction									
2	Abo	out Android	8							
	2.1	What Is Google Android?	8							
	2.2	What Is the Google Android SDK?	8							
	2.3	The Google Android Architecture	9							
	2.4	Google's Motivation Behind Android	9							
3	The Android SDK									
	3.1	General Structure Of The SDK	10							
	3.2	Tools Within The SDK	10							
	3.3	The Android Emulator	11							
	3.4	Android Documentation	12							
	3.5	ADT - Android Development Tools For Eclipse	12							
4	The	Android OS	14							
	4.1	The Android Emulator On OS-level	14							
	4.2	The Kernel	16							
	4.3	Fileystem Layout	16							
		4.3.1 System Image	17							
		4.3.2 User Data Image	18							
		4.3.3 Cache Image	19							
		4.3.4 SD Card Image	19							
	4.4	Android-Specific Daemons	19							
5	Source Code Life Cycle On Android									
	5.1	From Java Source Code To A Compiled Dex File	21							
	5.2	Packaging All Project Files Into An APK	21							
	5.3	How To Create And Deploy APKs Without Eclipse	21							
6	Fac	ing Difficulties In Developing On Android	23							
	6.1	Difficulties In Developing Dndroid Applications	23							
	6.2	Difficulties Developing In C/C++	23							
	6.3	Difficulties Developing In Java	23							
7	Interconnect Daemon and GetProcessList 25									
	7.1	Motivation	25							
	7.2	Handling Interprocess Communication	25							
	7.3	$\epsilon$								
	7.4	Interconnect Daemon								
	7.5	5 GetProcessList								
	7.6	Out Of Scope	27							
		7.6.1 What Did We Not Implement? Why?	27							
		7.6.2 What Else Could Be Done Based On This Project?	29							

References								
List	of Figures							
1	The Google Android Architecture overview	9						
2	Greeting you with the welcome screen							
3	UML-diagram of GetProcessList							
4	UML-diagram of GetProcessList	27						
5	UML-diagram of GetProcessList	28						
List	of Tables							
1	SDK Flements	10						

## 1 Introduction

For some time now, two operating systems for mobile phones have been dominating the mobile market: Symbian OS and Windows Mobile.

In January 2007 [1], Apple, as third big company to enter the mobile market, introduced the *iPhone* running the in-house developed operating system Mac OS X. Leaving the generation of keypads behing, the *iPhone* was one of the first phones to offer a ground-breaking new touchpad interface which works very fast and sensitive. This, combined with its appearance of both physical device and operating system, resulted in a big hype for Apple's new cash cow *iPhone*. In combination with *iPhone market*, offering thousands of applications which can easily be downloaded and installed, *iPhone* has become a major success in Apple's product history.

As last contestant, Google, famous for its dominating web search portal, entered the market with a Linux-kernel-based product: Google Android. Google Android, a Linux-kernel-based operating system, comes along with a quite new credo within this market: it is supposed to be open! This, in particular, means all elements used shall be provided in source code form so that developers will have the possibility to take a closer look into the system to maybe alter it to their needs. In order to guarantee a great impact of Google Android on the market, Google organized several developer contests in order to create a rich set of ready-to-go applications for the Android plattform before the actual phone is even available on the market, hoping to enter the competition with Apple's *iPhone* market. As strategic partners Google picked HTC as provider for the first physical device running Android and T-Mobile as telephone provider who will offer the T-Mobile G1, the name of the first upcoming device, coupled with a contract, for a consumer-friendly price.

Motivation of this Technical Report was to get a thorough understanding of the new arising platform *Google Android*, its framework, its underlying operating systems and possible differences to commonly known Linux operating systems. Furthermore, we wanted to get a feel for how *open* the proposed system really is (as stated by Google), meaning to identify difficulties in developing applications for Android, not only with the proposed core language Java but also with other well-known programming languages, e.g. C or C++. Once we identified and, finally, mastered those difficulties, we wanted to show and prove that other languages than Java might as well be used to successfully develop applications for the Android plattform. And, as an important point, we wanted to show that such applications can coexist with given Android Java applications and even communicate with those.

The layout of this Technical Report is as follows: While section 1 gives a brief introduction into the topic, section 2 will describe the Abouts of *Google Android*. Section 3 will cover the *Android SDK*, its structure, included tools, the emulator and so forth. Section 4 will take a deeper look, "under the hood", being specific, a description of the underlying operating system, the used kernel, filesystem layout and more. Interesting for developers, we will then give a basic overview of the source code life cycle in section 5. Section 6 will shortly discuss difficulties in developing applications for Android,

using any kind of programming language for this task. To prove that Java is not the only programming language that can be used on Android, we will, in section 6, give a brief introduction into our basic server-/client-architecture *Interconnect/GetProcessList*, a software set consisting of both Android-based Java-application and a unix service that has been implemented in the programming language C. In the same section, we will also show and discuss out-of-scope thoughts and what else could be done ontop of our server-/client-stack.

After having given a short introduction into the topic, we will continue with the Abouts of Google Android to get a common, basic understanding of what we are dealing with.

## 2 About Android

Google Android, as the new contestant on the market, gained great popularity after being first announced in November 2007. With its announcement, it also was accompanied by the founding of the Open Handset Alliance for which Google Android is said to be the flagship software. In following chapters, we will take a closer look at Google Android and describe its parts, functionalities, eventual problems and more. But first, we will start with a basic understanding of *Google Android*.

#### 2.1 What Is Google Android?

According to the Google Android website [2], Google describes its platform as

a software stack for mobile devices including an operating system, middleware and key applications.

Basically said, Android consists of a UNIX-like operating system based on a 2.6-series Linux kernel. The operating system has been enriched with all necessary elements that are required to provide basic functions like a networking stack, GSM/GPRS abilities, and more. On top of all this, Google offers a framework containing a rich set of Java methods enabling developers to create a wide range of software for mobile use on Google Android.

Taking a closer look at the operating system, we are dealing with an ARM-architecture based system (armv5tejl) which is quite common for small or handheld devices. The provided 2.6 kernel (2.6.25-00350-g40fff9a in SDK version 1.0\_r1) is slightly modified - specialized for handheld devices - and offers support for the most commonly used hardware devices in the embedded field of use (SD-Card, USB, and more).

## 2.2 What Is the Google Android SDK?

As no officially obtainable physical device is available at time of writing, developers will have to rely on the provided development kit: Google Android SDK.

A core element of the SDK is the actual Google Android Emulator which provides a graphical emulation of a possible handheld device running Google Android. Furthermore, the SDK not only provides the core classes of the Android framework packed into a Java Jar-file: it also includes the documentation in HTML-form and several tools that improve the usability and interaction with the emulator.

Once the SDK has been downloaded, developers can directly start creating Android-applications, compiling them and deploying them to the emulator. Nothing more than an editor and a working JDK (Sun JDK) is required.

Though, to make things easier, Google suggests Eclipse as the IDE to be used. For this, Google optionally provides an Android Plugin made for Eclipse that will take over a few basic tasks like creating the Android Project layout on disk, the integration of your SDK, eventually starting your emulator if not running yet and the deployment of you application on the emulator ... all this with just a few clicks.

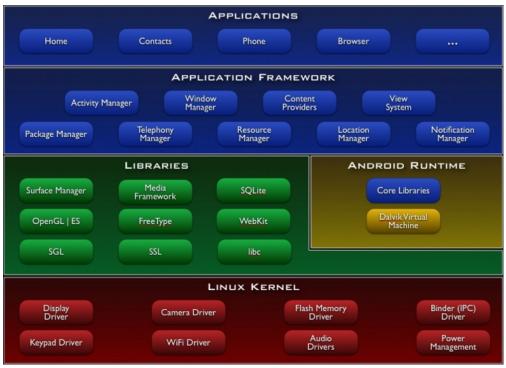


Figure 1: The Google Android Architecture overview

#### 2.3 The Google Android Architecture

Figure 1 shows the Architecture overview that can be found on the Google Android project website ([2]). The architecture basically consists of four sections: the Linux kernel (system) as underlying operating system interface, the libraries (e.g libc) as important part of the operating system; the Android framework providing all necessary classes and methods in order to write Android-compatible applications; and, as top section, the actual Android applications.

## 2.4 Google's Motivation Behind Android

It is hard to say what Google's plans are for Android. The physical device T-Mobile G1, which will be the first phone running with Android OS, will be Google's first step into the mobile market. If Android is going to advance to a sophisticated operating system for smart mobile phones or if it is going to be stripped down to a very basic system for every mobile phone hardware, we can only guess.

But, as we all know Google, Google often sets foot into a certain area with a basic version of application or service and then simply waits what the actual user is going to make out of it. In Android's case, this strategy might just be like that: Offer the G1, offer well-documented SDKs and APIs for Java, offer and promise openness ... and watch what is going to happen next. The users will define Android. And as soon as Google knows in which direction we are going, they are most likely making something great out of it.

Let us wait and see how Android is going to develop.

Table 1: SDK Elements

android.jar	The Android classes				
docs	Android documentation (HTML)				
LICENSE.txt	License terms				
samples	Sample projects based on the Android framework				
tools	binaries, libraries, emulator images				

## 3 The Android SDK

The Google Android SDK can be freely downloaded on the project website:

http://code.google.com/android/download.html

Google provides the Android SDK for three different platforms: Windows, Apple's Mac OS X and Linux. All SDKs are quite similar in its structure, therefore we will, from now on, refer to the Linux-version of the SDK.

In following sections, we will take a glance at each part of the SDK.

#### 3.1 General Structure Of The SDK

Once the SDK-archive is unpacked on the host system, a quick glance at the filesystem structure reveals following:

The *tools* directory is, by far, the most interesting directory to inspect: in its root directory it offers several executables consisting of shell scripts, binaries, python scripts, basically everything that enables the developer to create Android projects and work with them. Furthermore, the *tools/lib* directory contains several Java-libraries in form of Jarfiles. And, last but not least, the *tools/lib/images* directory accommodates the emulator base images which are loaded upon emulator start.

#### 3.2 Tools Within The SDK

All important tools for successfully developing applications for the Android platform can be found in the *tools* directory. If the developer uses Eclipse as standard development IDE, he will most likely not be confronted with any of these: the optionally available Eclipse plugin provided on the Google Android project website takes over the responsibility for all necessary tasks. One can simply concentrate on writing the code, a simple click on the build button will automatically do the rest.

In case, you are not using any IDE or simply not using the plugin, following will give a short overview of each tool and its meaning:

- aapt Android Asset Packaging Tool This tool enables the developer to view, create and update Zip-compatible archives. It can also compile resources into binary assets.
- **activityCreator** The shell script is used to create basic filesystem structure for a new Android project. From then on, any IDE or Editor can be used to alter the code.
- **adb Android Debug Bridge** A very powerful service can basically enables all communication to and from the emulator. Connecting with the emulator via port 5555, it enables the user to execute remote shell commands or simply provides a login shell.
- **aidl Android Interface Definition Language** The Android Interface Definition Language tool is used to generate code that enables two processes on an Android-powered device to communicate with each other using IPC.
- **ddms Dalvik Debug Monitor Service** This tool provides port-forwarding services, screen capturing on the device, thread and heap information, logcat and much more (see [3]).
- **dmtracedump** Alternative to generate graphical call-stack diagrams from trace log files
- **dx** Generates bytecode from .class files. It converts the files into the .dex file format which can be run in the Android environment.
- **mksdcard** Creates a FAT32 disk image which can be loaded upon emulator start. Simulate the presence of an SD card.
- **sqlite3** The well-know sqlite database which saves its databases in plain files. Can be used for debugging issues (inspecting databases in Android).

**traceview** Traceview is used to view tracefiles which can be created on the device.

**emulator** Executable to start the Android Emulator.

#### 3.3 The Android Emulator

Based on Qemu (see [4]), the Google Android emulator works with several images which can be found as files in the SDK:

```
hgschmidt@toyo:~/Android/SDK/android-sdk/tools/lib/images$ file *
kernel-qemu: data
NOTICE.txt: ISO-8859 English text
ramdisk.img: gzip compressed data, from Unix, last modified: Mon Sep 22 22:37:59 2008
skins: directory
system.img: VMS Alpha executable
userdata.img: VMS Alpha executable
hgschmidt@toyo:~/Android/SDK/android-sdk/tools/lib/images$
```

Both system and user data images can be found here which represent two important parts within the operating system. We will take a more detailled look at all images in the next chapters so please hold on by now.

In order to start the emulator, the *tools* directory offers the emulator binary. The help page to the command reveals a quite powerful background:

```
hgschmidt@toyo:~/Android/SDK/android-sdk/tools ./emulator -help
Android Emulator usage: emulator [options] [-qemu args]
options:
-system <dir>
-datadir <dir>
-datadir <dir>
-kernel <file>
use specific emulated kernel
-ramdisk <file>
ramdisk image (default <system>/ramdisk.img
-initdata <file>
initial data image (default <system>/system.img
-data <file>
data image (default <datadir>/userdata.img
-data <file>
cache <file>
cache partition image (default is temporary file)
-nocache
-sdcard <file>
SD card image (default <system>/sdcard.img
[...]
```

As already stated, the emulator works ontop of Qemu, therefore the emulator offers nearly all parameters in order to tweak the emulated qemu system. As we can see here, the used images can be placed at different locations making it possible to start several instances of the emulator with all different user data and system images.

#### 3.4 Android Documentation

No development without appropriate documentation. Google is aware of this and offers two valuable sets of information resources.

**SDK documentation** Each SDK includes an offline documentation set including the Android Java API and code examples for most basic issues when developing for Android. The documentation can be easily observed with any given browser.

**Online Resources** Google offers an easy to understand API, enriched with various code examples, guidelines for naming conventions, extensive descriptions to all parts of the Android platform on the project website:

```
http://code.google.com/android/
```

The website will most obviously be the most valuable resource for Android developers.

## 3.5 ADT - Android Development Tools For Eclipse

Google provides developers with a rich framework to start developing applications for Android. The SDK is one part of this framework and offers almost anything a developer needs to start, given an editor to write the code. But, additionally, Google also offers another valuable piece in the Android application development life-cycle: ADT, the Android Development Tools, a plugin for the Eclipse IDE. Once installed into Eclipse, Eclipse will from then on

- know how to create Android projects,
- start up the emulator on compilation and deployment automatically if not running yet,
- offer interfaces to some mechanisms in Android, e.g. logcat, heap monitoring, power management, etc. for extensive debugging.

All steps that would have to be accomplished in a given shell can be handled with this plugin. Eclipse becomes the only interface developers will ever need to code, compile and deploy Android applications.

#### 4 The Android OS

Google announced Google Android to be the first open operating systems for handheld devices. In order to make that possible, Google decided to use the open and freely available operating system Linux. Linux has become a more and more used operating system, commonly used in the server and data center area. But, at the same time, is also quite often used as the running system for small, embedded devices.

Running on a handheld device, the Linux system is highly optimized for its environment: only elementary parts of Linux have been used like the basic and necessary kernel providing the most common drivers that will be required during the lifetime of the handheld device, a filesystem with a very own interpretation of a filesystem layout, basic libraries that seem to be stripped down to the bare minimum to run the whole system, and more. In following chapters, we will take a closer look at all of these elementary parts in Google Android. But before that, we will just take a short look at the maybe most important part that is provided with the Android SDK: the Emulator.

#### 4.1 The Android Emulator On OS-level

Being shipped with the SDK, Google provides an basis for the development tasks: an emulator which emulates a possible GPhone with graphical user interface. Furthermore, several scripts and daemons form the whole package of the provided emulator, simply to make certain things easier or being able to actually communicate with the emulated system.

First of all, to start the emulated Android system, one either has to deploy an existing, already created Android project in Eclipse which will automatically cause the emulator to start, given an installed Android plugin. Or, simply start the emulator executable from the *tools* directory of the SDK

```
\label{linux_x86-1.0_r1/tools} $$.\emulator \& [1] 15015 $$ hgschmidt@toyo:~/Android/SDK/android-sdk-linux_x86-1.0_r1/tools$$
```

After a few seconds it will greet you with a graphical interface (see figure 2) of a small handheld which is (almost) fully functional: outgoing and incoming calls are only simulated.

After having clicked through the interface for a while in order to see that everything seems to be working and after having found out that Google Android just looks great, we get bored and start trying to find out what is beneath the shiny interface. In order to do that, we will need the *Android Debug bridge* [5], a special services Google provides in order to communicate with the emulator. *adb* is quite powerful enabling the developer to do several tasks like:

- "push" files into the emulator
- "pull" files from the emulator
- view device log
- forward socket connections



Figure 2: Greeting you with the welcome screen

and more ...

To get into the system, *adb* provides the *shell* parameter. After executing *adb shell* the developer will find himself in a login shell within the emulator.

```
hgschmidt@toyo:~/Android/SDK/android-sdk-linux_x86-1.0_r1/tools$ ./adb shell * daemon not running. starting it now * daemon started successfully * \#
```

One can go ahead and drop off a few first linux shell commands:

```
# ls -1
                                   2008-10-20 18:21 sqlite_stmt_journals
drwxrwxrwt root
                 root.
d----- system cache
drwxrwxrwx root root
-rwxr-x--- root
                                   2008-10-20 18:21 cache
                                   2008-10-20 18:21 sdcard
2008-10-20 18:21 etc -> /system/etc
drwxr-xr-x root root
                                   2008-09-22 22:41 system
drwxr-xr-x root
                  root
                                   1970-01-01 01:00 sys
dr-xr-xr-x root root
                                  1970-01-01 01:00 proc
drwxr-x--- root root
-rw-r--r- root root
drwx----- root root
                                   1970-01-01 01:00 sbin
                                93 1970-01-01 01:00 default.prop
                                  1970-01-01 01:00 root
drwxr-xr-x root
                                   2008-10-20 18:21 dev
                 root
```

Once this has been accomplished, we have a common basis to take a closer look at the specific parts of the operating system.

#### 4.2 The Kernel

As has already been stated, Google Android provides a 2.6 kernel (output from SDK version 1.0\_r1):

```
Linux localhost 2.6.25-00350-g40fff9a #1 Wed Jul 23 18:10:44 PDT 2008 armv5tejl unknown
```

The kernel configuration can be found in the proc filesystem as zipped archive:

```
# zcat /proc/config.gz
#
# Automatically generated make config: don't edit
# Linux kernel version: 2.6.25
# Wed Jul 23 18:09:51 2008
#
CONFIG_ARM=y
CONFIG_SYS_SUPPORTS_APM_EMULATION=y
# CONFIG_GENERIC_GPIO is not set
CONFIG_GENERIC_TIME=y
CONFIG_GENERIC_CLOCKEVENTS=y
CONFIG_MMU=y
# CONFIG_NO_IOPORT is not set
[...]
```

The kernel can be rebuilt, the sources are available on the project website ([6]). Once downloaded and extracted, the above mentioned configuration can be used as basis for further configurations. After having configured all necessary parts, having successfully compiled the new kernel, it can be loaded via an emulator-parameter:

```
$ emulator -kernel <file>
```

## 4.3 Fileystem Layout

Google Android has been designed to offer, both, a very stable basis every developer can rely on and, at the same time, enable developers to experiment and develop freely without any significant limitations. This is why Google obviously seperated distinguishable parts into different images that are loaded and mounted upon emulator start:

- system image
- user data image
- cache image

Eventually, more filesystems are mounted upon start but do not require further investigation as they are quite standard and common on other Linux-variants, e.g. the proc filesystem.

Here is a complete overview of the mounted fileystems within Google Android:

```
# /data/busybox/df -h
Filesystem Size Used Available Use% Mounted on
tmpfs 46.2M 0 46.2M 0% /dev
tmpfs 4.0M 0 4.0M 0% /sqlite_stmt_journals
/dev/block/mtdblock0 64.0M 41.3M 22.7M 65% /system
/dev/block/mtdblock1 64.0M 26.3M 37.7M 41% /data
/dev/block/mtdblock2 64.0M 1.1M 62.9M 2% /cache
```

#### 4.3.1 System Image

The system image contains the core parts of the operating system, for example:

- system binaries
- system libraries
- configuration files
- and more ...

Furthermore, the system image includes the actual Android framework with all necessary JAR-files:

```
# ls /system/framework
framework-res.apk
core.jar
ext.jar
framework.jar
am.jar
android.awt.jar
android.policy.jar
\verb|com.google.android.gtalkservice.jar| \\
android.test.runner.jar
com.google.android.maps.jar
input.jar
monkey.jar
pm.jar
ssltest.jar
svc.jar
framework-tests.jar
itr.jar
services.jar
```

It also includes a basic set of Android applications:

- Alarmclock
- Calculator
- Contacs
- Camera
- Music player
- Phone
- Google Maps
- and more ...

The system image is locked, so read-only access is granted but not more. Obviously, this will be the place for Google to provide updates to the system without any interference by the user.

As it is locked, importing new applications, either written ontop of the Android Framework or in any other programming language, is not possible at the moment. If it would be possible we would still have to face the problem of very limited filesystem size: for smaller applications it might be sufficient. For bigger applications with eventually big databases will easily exceed the available 21MBytes left over in the image.

#### 4.3.2 User Data Image

While the system image contains the fairly basic parts of Google Android, the user data image provides the main location to basically put files into the system. New applications that have been either downloaded or simply self-written will be put here, if correctly installed by the provided handlers of the emulator.

In its basic state, it already provides a quite complex filesystem structure:

On installation of new applications, obviously several things happen in order to integrate the new application into the fileystem:

- The apk (Android package) is placed into /data/app
- application-centric libraries are installed into /data/data/<application name>
- application-relevant databases are setup and eventually updated

The user data image is, most obviously, a writeable filesystem. New applications which might not have been created upon the Android framework but eventually in a different higher-level programming language like C or C++ will most likely find its way into user data image.

Problems of the user data image is that, quite similar to the system image, Google only provides a very limited area to import new applications: approx. 40Mbytes have to be shared for all Java-based applications, configuration files and databases, and applications written in other programming languages than Java.

#### 4.3.3 Cache Image

The cache image offers the exact same size to interact as both system and user data image: 65 MBytes. Except for being used for the typical *lost+found* directory, we could not observe any other usage than that ... rendering it a good place for temporarily saving files that will easily exceed the limitations of the other two images.

#### 4.3.4 SD Card Image

Having such great limitations on both system and user data image, the SD Card seems to be the perfect alternative to provide large applications within Android!

In order to use a simulated SD Card, one first has to create one with the provided SDK-tool *mksdcard*. Providing a label and a size, the image is created and can be used almost instantly. Once the emulator is started with appropriate parameter *-sdcard <location of SD Card image>* the emulator will start and offer the mounted image at */sdcard*.

Major drawback when using the SD Card image in the emulator is: although, the image can be created with no obvious size limitations, it cannot be used to execute provided shell scripts or applications written in e.g. C/C++. The problem is that certain permissions have to be set to applications that are supposed to be run within the operating system. All files in the SD Card image will receive a certain user id, group id and file permissions which cannot be altered in any way: no execution bit can be set and the files cannot be allocated to user *shell* and group *shell* (the user/group for files to be executed).

If this is going to change in future (when the first devices are actually available on the market), is not known yet. Hopefully, Google will change this offering developers more space for their applications.

## 4.4 Android-Specific Daemons

Apart from the standard server daemons that are available on Linux-based systems running kernel version 2.6.xx, Google added own daemons, supposedly in order to be able to interact with the underlying operating system. Taken from the output of the *ps* command (list all system-wide processes), following *additional*, *non-standard* daemons will be up and running in a running instance of the Google Android Emulator:

USER	PID	PPID	VSIZE	RSS	WCHAN	PC		NAME
[]								
system	19	1	796	248	c017fb5c	afe0b74c	S	/system/bin/servicemanager
root	20	1	1820	328	ffffffff	afe0c0bc	S	/system/bin/mountd
root	21	1	652	248	c018a19c	afe0c0bc	S	/system/bin/debuggerd
radio	22	1	5308	636	ffffffff	afe0bdbc	S	/system/bin/rild
root	24	1	69508	18672	c008be9c	afe0b874	S	zygote
media	25	1	16032	3048	ffffffff	afe0b74c	S	/system/bin/mediaserver
root	28	1	788	296	c01eb8b0	afe0b50c	S	/system/bin/installd
root	31	1	812	308	c00aabf0	afe0c61c	S	/system/bin/qemud
system	55	24	19657	2 2361:	2 fffffff	f afe0b74c	2 1	S system_server
[]								

Following list will give a short description of each daemon and, supposedly, its purpose:

**servicemanager** Manages and provides access to services (a special application form) on the Android platform

**mountd** The mount daemon providing access to the YAFFS2 images (system, data, cache).

**debuggerd** Offers functions to handle crashes, logs, etc.

rild Radio daemon

**zygote** Started by /system/bin/app\_process. Father process of each new application process.

mediaserver Handle media libraries on the phone

**installd** As the name states, this daemon controls the install mechanism for Android packages on the phone (Emulator).

**qemud** Qemu daemon. The SDK runs within a virtual environment provided by Qemu.

**system\_server** Part in the dalvik VM application execution chain. Will require more analysis to identify the exact functionalities

## 5 Source Code Life Cycle On Android

In this section, we will cover the source code life cycle of an Android application, starting with the initial Android project layout, the development of the project written in Java, the compilation of the Android package and, finally, the deployment of such a project on the emulator.

#### 5.1 From Java Source Code To A Compiled Dex File

Android introduces a new virtual environment and concept that is based on the Java programming lanugage and a new file type: *dex*. Most parts of Android are open, source code is available excluding the virtual environment. So far, we can say that when Java sources are compiled, the final result is *dex* file which can then be run on the given device with appropriate tools. The user will not obviously have to cope with *dex* files as those are simply parts of the application package which a user will work with.

## 5.2 Packaging All Project Files Into An APK

An Android project consists of a given directory structure with given code sources. Once the project is compiled and built, all files are packaged into one single file, an *apk* (Android package). Compiled and packaged APKs can be found within the *bin* directory of the project. An APK basically is a zip file, including all resource files, compiled sources as *dex* files (Google's own file format for running Java bytecode on Androi), and the AndroidManifest file which defines the application and its activities.

APKs are required to successfully deploy applications onto the destination device: Android's mechanism will take a given APK, deploy it by unpacking all relevant files in defined locations in filesystem, update the application database of the Android-based device and finally start up the new application.

## 5.3 How To Create And Deploy APKs Without Eclipse

ADT in Eclipse takes over quite a few tasks in the code life-cycle. Not using ADT will require to manually run through all necessary steps:

- 1. The base project layout needs to be created with with *activitycreator* which is part of the SDK and can be found in the *tools* directory of the SDK.
- 2. One important product of the *activity creator* is the *build.xml* file holding all important project definitions (where to find the SDK, name of the project, etc).
- 3. *ant*, a Java build tool from the famous Apache Ant project, will require the *build.xml* to successfully compile a given Android project. The product of the *ant* run is a ready-to-deploy APK.
- 4. If using the emulator, the developer has to manually start the emulator from the *tools* directory of the SDK.

- 5. With *adb Android Debug Bridge* and appropriate parameters a given APK can easily be deployed on any Google Android-based target device.
- 6. Once installed on the target device, the application can be selected and started from the Android device menu.

## 6 Facing Difficulties In Developing On Android

Google offers a comprehensive API to help out in situations where the developer needs help. In this section we will shortly discuss where Google cannot help, where developers will have to face problems and difficulties.

#### 6.1 Difficulties In Developing Dndroid Applications

All SDKs for each supported operating system platform are quite similar. Using the officially supported development IDE Eclipse abstracts all underlying steps automatically taken on compilation and deployment of a given Android project on the emulator.

As soon as a developer refuses to use IDE (due to own preference) and instead use a different editor, a few things have to be beared in mind:

- Android projects have to be created manually with the *activitycreator*.
- *ant* has to be used to compile the project and create the APK, and *adb* has to be used to install the APK on the target device.
- If the developer wishes to manually compile his Android project with *ant*, *ant* will require the *build.xml*. This file is ONLY created when the project was initially created via *activitycreator*. Eclipse with ADT will not create this important configuration file and therefore a manual *ant* compilation run will fail.

## 6.2 Difficulties Developing In C/C++

A major issue when developing applications for Android in C/C++ is that the developer will have to face the problem that he or she will be developing for a different platform: ARM. This might require advanced knowledge of this platform type and also knowledge in how to appropriately compile code for ARM architecture.

The Android Sources include a complete cross-compile toolchain for ARM. But, the developer will have to face learning the Android Makefile system in order to successfully compile code on basis of the Android Sources' cross-compile toolchain. Especially getting all required includes correct in order to compile against the Android libraries and header files will appear to be a major problem in the code life cycle.

If this gets a major problem, there are other solutions: CodeSourcery offers a cross-compilation ARM toolchain which can also be used to compile the sources. But, most likely the includes of the header files will be different, the libraries might differ in its functionalities. A way out is to statically compile and link all applications in order to statically include the necessary library functions. The developer receives one big, all-in-one static binary which can then be pushed to the device and run. This might be a sensible solution for smaller projects. But, most obviously, not desired in situations where applications get quite big and full-blow.

## 6.3 Difficulties Developing In Java

The official API is very well documented. Any problem going beyond the capabilities of the API can be discussed on several mailing lists that are available for the Android

project. Ranging from the beginners mailing list up to porting other existing applications to Android: Google made sure that everybody can find an appropriate answer. Using ADT with Eclipse makes development very easy and straigh-forward. Other IDEs, e.g. Netbeans can also be used as the community starting developing a pendant to ADT, simply for Netbeans. Furthermore, Google did not restrict developers to keep information for themselves. This resulted in various new websites dealing with Android development giving hundreds of new ideas, code examples and more.

#### 7 Interconnect Daemon and GetProcessList

As a first demonstration of what is possible on Google Android, we decided to write a small application stack, which both includes a Java-based part *GetProcessList*, using the official Google Android Java APIs, and a helper application *Interconnect Daemon*, written in C, which operates on OS-level.

Before going into the architecture description in detail, we will first cover the motivation behind this application stack and how to implement the common communication channel between both distinct parts.

#### 7.1 Motivation

Main intentions for this application stack were to demonstrate ...

- the inability of the Java-level to cope with certain OS-level tasks.
- that programming languages other than Java can be used to develop applications for Google Android.
- how such applications can communicate with each other and therefore legitimate their existence.

Various tasks can be mastered via the Android Java API but still though, a few remain impossible to be done on Java-level. While the Java API offers a fairly big number of methods and classes, it still cannot overcome the problem that each application is run with the application's user permissions. Thus, certain files or ressources are simply unavailable on a Linux system (e.g. some log files on standard Linux distributions). In Android, this counts for a rare number of files within the /sys and the /proc directories.

Altering important data on OS-level is an almost impossible task: e.g. network information data can be *read* but not easily *altered*.

## 7.2 Handling Interprocess Communication

The application stack works, as already mentioned, in two tiers: the stack is split up into a Java-application, using the Android Java APIs, and a small Linux server daemon, the *Interconnect Daemon*, written in C, which operates on OS-level.

As both parts work fairly independent, a common communication channel had to be established. In order to implement such a common communication channel, several concepts have to be taken under consideration:

**JNI - Java Native Interface** *JNI* offers a possibility to connect Java-applications to shared libraries which are implemented in C/C++. The shared libraries offer functions to call *libc* system and library calls. The Java-application has to be the instance in control as it triggers each function from higher level. Once mastered the page alignment issue, *JNI* is a very promising concept to quickly implement application stacks with almost full access to the underlying OS-level.

For our usage, *JNI* did not fit as we opted for an independent server daemon on OS-level which can communicate to various independent Java-applications.

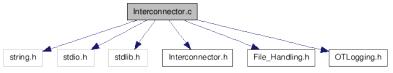


Figure 3: UML-diagram of GetProcessList

**Sockets** Sockets are a very commonly used concept for communication, especially *IPC* (Interprocess Communication). While not digging to deep into this concept, the following concept *Named Pipes* has some minor advantages over *Sockets* due to non-blocking-issues in certain cases.

**Named Pipes** Named Pipes is also a very commonly used concept for IPC and it is very simple to implement. The libc offers a simple function to create such FIFOs (First In, First Out). A huge number of these can already found in the /dev filesystem, e.g. in Google Android's case for the logging system logcat.

#### 7.3 Architecture Design And UML-Diagramm

Taking a top-down-view on the architecture design, *GetProcessList* is the application that interacts with the user. It offers a graphical user interface based on the Android Java API, which presents all results from the application stack on the screen.

The *Interconnect Daemon* runs as a regular server daemon on Linux-level from where it gathers relevant system (process) data and passes this on to *GetProcessList*.

#### 7.4 Interconnect Daemon

The *Interconnect Daemon* is a small server daemon, compiled for ARM architecture. For the sake of easier handling, the application has been statically compiled and linked, rendering it a single ELF-compatible binary file which can be transferred to the Android Emulator or the actual physical device G1 and executed (Please note: There is not root-access on G1!)

Its main tasks are to:

- identify all currently available processes in the system, taken from the /proc filesystem. This filesystem offers a common information base to exchange data with the Linux kernel.
- record all processes and collect detailled information on each process.
- provide GetProcessList with this information.

The data is read from /proc and directly written to the named pipe.

Figure 3 will display the basic dependencies to given libraries.

In figure 4 you will see the call graph for the main function which handles all necessary steps to establish the communication via named pipes and send the process status information to the receiver on Java-level.



Figure 4: UML-diagram of GetProcessList

#### 7.5 GetProcessList

GetProcessList is an application written in Java. It is responsible for following tasks:

- Receive all information sent via named pipe (the interprocess communication channel between *Interconnect* and *GetProcessList*)
- Process each line sent via named pipe
- Save the data to its database
- Present the list of all processes in a graphical user interface

In figure 5 you will find the UML class diagram of GetProcessList.

## 7.6 Out Of Scope

Both *Interconnect* and *GetProcessList* are to be seen as a demonstration of what is possible on Google Android. Especially developing applications not based on the officially supported language Java is, disregarding a few minor hurdles to take, possible and, in some cases, a sensible choide.

As the complete application stack has to be considered as a demonstration software, a few things have not been implemented due to time constraints, out-of-scope-issues or the prototype character of this demonstrator. We will briefly cover what has not been implemented but may be in future in order to make this application stack a useful and usable product for end-users.

Furthermore, we will also cover eventual problems during development, problems that aroused due to programming language characteristics or simply anything that formed a given obstacle.

#### 7.6.1 What Did We Not Implement? Why?

Following points were disregarded during development of the demonstrator application stack:

**Using Standard protocols For IPC** *Interconnect* reads all process status files and simply pipes its output, which basically represent strings (char-arrays in C), into the established named pipe. No formatting of the strings sent over the pipe is done, requiring the receiver application *GetProcessList* to know which string is coming when and how each string has to be formed.

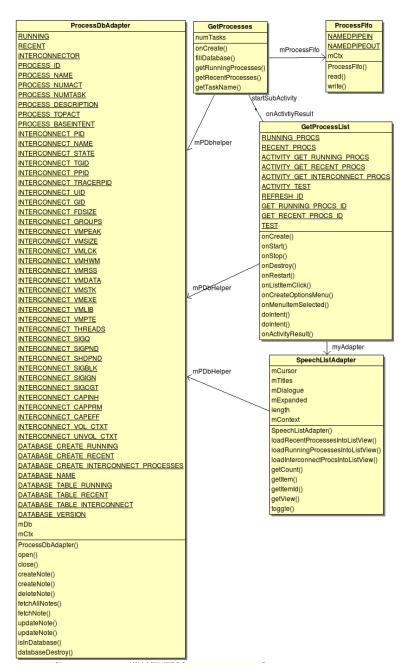


Figure 5: UML-diagram of GetProcessList

In a more advanced version of *Interconnect*, which has been used in following project, this communication is based on sending standardized JSON Javascript Object Notation [7] objects. Using JSON, several major advantages arise:

- Each object is a complete set of data including all necessary information. In our case this would mean that each object includes all data concerning a single process.
- Each object is one single string. Due to its optimal structure, all data concerning one process is sent "in one go".
- Receiver has more freedom on how to parse the received object and how to postprocess it.

#### 7.6.2 What Else Could Be Done Based On This Project?

Both *Interconnect* and *GetProcessList* could be extended to a useful *ps*-clonse (*ps* displays the process status on common UNIX- and Linux-based systems) including a graphical user interface for Android. For end-users who want to know what exactly is happening on their device, this tool could be very good solution.

# **Acknowledgement**

We hereby want to thank those people who supported us, always had an open ear whenever we had problems; simply all who have made all this possible. So special thanks go to Aubrey-Derrick Schmidt, the project manager of this project, and Seyhit Ahmet Camtepe, leader of our competence center CC-Security. Furthermore we would like to thank DAI Labor, especially Prof. Dr. Sahin Albayrak for giving us this possibility to work in this, in our opinion, very interesting field of mobile telecommunication, UNIX-based operating systems and security.

# References

- [1] MacWorld.com, "Apple unveils iphone."
- [2] G. Inc., "Google android project website."
- [3] G. Inc., "Android dalvik debug monitor service."
- [4] bellard.org, "Qemu project website."
- [5] G. Inc., "Android debug bridge."
- [6] G. Inc., "Android code downloads."
- [7] "Json javascript object notation."