Karlsruher Institut für Technologie

Analysis of the Android Architecture

Studienarbeit
von

Stefan Brähler

an der Fakultät für Informatik

Erstgutachter: Prof. Dr. Frank Bellosa

Betreuende Mitarbeiter: Dr. Jan Stöß, Dipl.-Inform. Konrad Miller

Bearbeitungszeit: 2. Juni 2010– 6. Oktober 2010

Deutsche Zusammenfassung

Die vorliegende Studienarbeit beschäftigt sich mit dem Android Betriebsystem für mobile Systeme. Die Arbeit stellt das System grundsätzlich vor und zeigt an einigen Stellen Besonderheiten und spezielle Anpassungen von Android auf.

Als Betriebsystem für mobile Systeme wie Smartphones und Tablets liegen bei Android Schwerpunkte auf Energieeffizienz und Energieverwaltung. Diese Schwerpunkte treten an vielen Stellen in den Vordergrund bei Designentscheidungen und prägen daher das System sehr stark.

Android basiert auf Linux und verschiedenen typischen Bibliotheken und Diensten im Linux Umfeld auf, z.B. libSSL, FreeType und SQLite. Diese Infrastruktur bildet die Basis für eine Laufzeitumgebung in der Java Programme ausgeführt werden. Die Laufzeitumgebung Dalvik führt modifizierte Java class Dateien aus, wobei jedes Programm in einer eigenen Dalvik Instanz in einem eigenen Prozess läuft. Dalvik ist spezialisiert und optimiert auf die besonderen Gegebenheiten und Limitierungen von mobilen Systemen, wie begrenzter Speicherplatz und Energiebedarf.

Anwendungen in Android sind aus verschiedenen Komponenten aufgebaut, wobei die einzelnen Komponenten klar getrennte Aufgabenbereiche haben. Diese Modularisierung erlaubt es, daß sich Anwendungen bestimmte Komponenten teilen können und daß einzelne Komponenten im System ausgetauscht werden können. Alle Anwendungen sind voneinander getrennt und können im Normalfall nicht auf Daten anderer Anwendungen zugreifen.

In den Kernel sind einige Änderungen eingeflossen die den Android Kernel hauptsächlich um neue Funktionen in Bereichen der Energieverwaltung und Speicherverwaltung erweitern. Die weitreichenste Änderung betrifft die Energieverwaltung in Form der neu eingeführten *wake locks*, welche es erlauben, dem System bestimmte Schlaf- und Ruhezustände vorzuenthalten. Weiterhin sind neue Treiber und Anpassungen für die Android Infrastruktur in den Kernel gelangt.

Für die Anwendungsentwicklung stehen eine Vielzahl von Programmierschnittstellen bereit, welche viele Java Standard Schnittstellen abdecken und teilweise erweitern. Android bietet weitreichende Netzwerkunterstützung und ein erweiterbares Framework für verschiedenste Medienformate. Zur Datenspeicherung und

Datenverwaltung stehen sowohl normale Dateien als auch SQLite Datenbankdateien samt Mitteln zur Sicherung von Daten und Einstellungen zur Verfügung.

Um die Entwicklungsarbeit zu erleichtern, gibt es eine Vielzahl von Werkzeugen die in einem Software Development Kit (SDK) mitgeliefert werden. Zu diesen Werkzeugen gehören u.a. ein Emulator, ein Eclipse Plugin und eine Debug Shell samt Debug Monitor. Zum Testen und Debuggen gibt es sowohl Werkzeuge, als auch ein Instrumentation Framework und angepasste JUnit Tests.

Hiermit erkläre ich, die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Literaturhilfsmittel verwendet zu haben.

I hereby declare that this thesis is a work of my own, and that only cited sources have been used.

Karlsruhe, 6. Oktober 2010

_____

Stefan Brähler

Contents

Chapter 1

Introduction and overview of Android

As smartphones and tablets become more popular, the operating systems for those devices become more important. Android is such an operating system for low powered devices, that run on battery and are full of hardware like Global Positioning System (GPS) receivers, cameras, light and orientation sensors, WiFi and UMTS (3G telephony) connectivity and a touchscreen. Like all operating systems, Android enables applications to make use of the hardware features through abstraction and provide a defined environment for applications.

Unlike on other mobile operating systems like Apple's iOS, Palm's webOS or Symbian, Android applications are written in Java and run in virtual machines. For this purpose Android features the Dalvik virtual machine which executes it's own byte code. Dalvik is a core component, as all Android user applications and the application framework are written in Java and executed by Dalvik. Like on other platforms, applications for Android can be obtained from a central place called Android Market.

The platform was created by Android Inc. which was bought by Google and released as the Android Open Source Project (AOSP) in 2007. A group of 78 different companies formed the Open Handset Alliance (OHA) that is dedicated to develop and distribute Android. The software can be freely obtained from a central repository [12] and modified in terms of the license which is mostly BSD and Apache. [11, 8, 6, 7]

The development of Android takes place quickly, as a new major release happens every few months (see section 1.2). This leads to a situation where information about the platform becomes obsolete very quickly and sources like books and articles can hardly keep up with the development. Sources that keep up with the pace are foremost the extensive SDK documentation, documentation in and the source code itself as well as blogs.

1.1 Structural overview

The Android software stack as shown in figure 1.1 can be subdivided into five layers: The kernel and low level tools, native libraries, the Android Runtime, the framework layer and on top of all the applications.

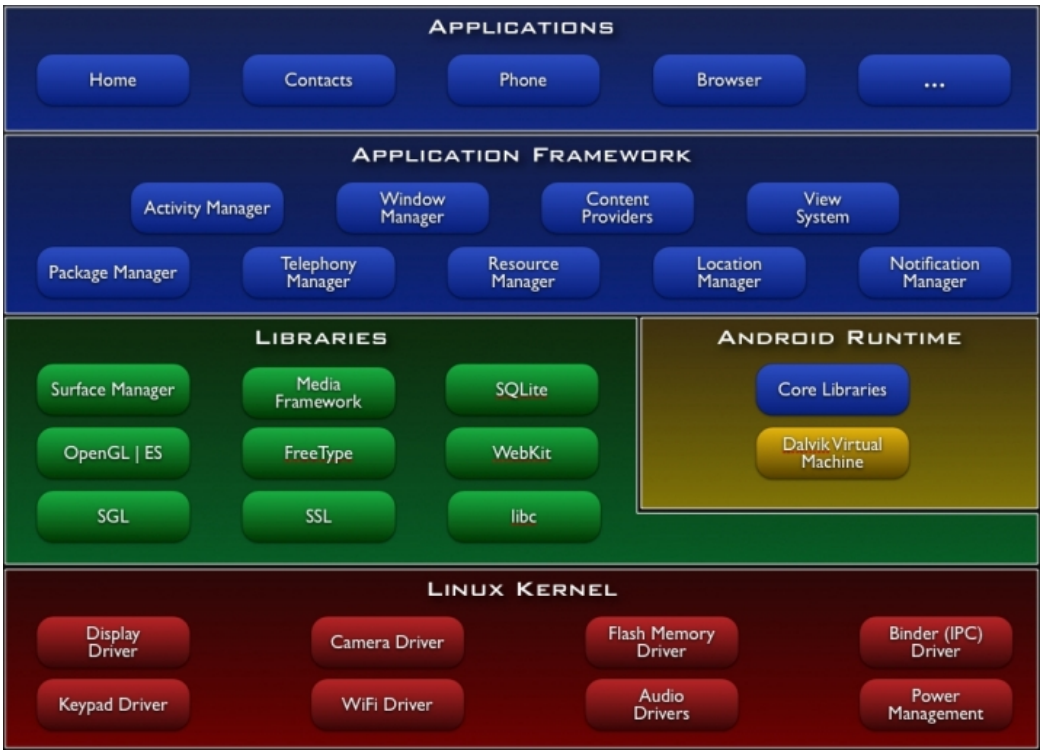

Figure 1.1: Android system architecture. Green items are written in C/C++, blue items are written in Java and run in the Dalvik VM. Image taken from [22, What is Android?].

The kernel in use is a Linux 2.6 series kernel, modified for special needs in power management, memory management and the runtime environment. Right above the kernel run some Linux typical daemons like *bluez* for Bluetooth support and *wpa_supplicant* for WiFi encryption.

As Android is supposed to run on devices with little main memory and low powered CPUs, the libraries for CPU and GPU intensive tasks are compiled to device optimized native code. Basic libraries like the libc or libm were developed especially for low memory consumption and because of licensing issues on Android. In this layer the surface manager handles screen access for the window manager from the framework layer. Opposing to other frameworks, the media framework

resides in this layer, as it includes audio and video codecs that have to be heavily optimized.

The Android Runtime consists of the Dalvik virtual machine and the Java core libraries. The Dalvik virtual machine is an interpreter for byte code that has been transformed from Java byte code to Dalvik byte code. Dalvik itself is compiled to native code whereas the the core libraries are written in Java, thus interpreted by Dalvik.

Frameworks in the Application Framework layer are written in Java and provide abstractions of the underlying native libraries and Dalvik capabilities to applications. Android applications run in their own sandboxed Dalvik VM and can consist of multiple components: Activities, services, broadcast receivers and content providers. Components can interact with other components of the same or a different application via intents.

1.2 Brief version history

Android is a young platform and the development is very rapid, as new major releases come out every few months. The following list shows the major Android versions and bigger changes in each version. More detailed information about the changes in each Android version can be found in [22, Android x.y Platform Highlights].

1.1 – February 2009 – Initial release

1.5 (Cupcake) – April 2009 – User Interface (UI) updates for all core elements, accelerometer-based application rotations, on-screen soft keyboard, video recording & playback, Bluetooth (A2DP and AVCRP profiles), based on kernel 2.6.27

1.6 (Donut) – September 2009 – Gesture support, support for higher screen resolutions (WVGA), text-to-speech engine, Virtual Private Network & 802.1x support, based on kernel 2.6.29

2.0 (Éclair) – October 2009 – Major UI update, Bluetooth 2.1 (new OPP and PBAP profiles), media framework improvements, Microsoft Exchange support, based on kernel 2.6.29

2.1 (Éclair) – January 2010 – Minor update, UI tweaks, based on kernel 2.6.29

2.2 (Froyo) – May 2010 – Performance optimizations, just in time compiler, tethering and WiFi hotspot capability, Adobe Flash support, enhanced Microsoft Exchange support, OpenGL ES 2.0 support, based on kernel 2.6.32

This study thesis is based on Android version 2.2. which is the latest release as of writing.

Chapter 2

Application anatomy

Running applications is a major goal of operating systems and Android provides several means on different layers to compose, execute and manage applications. For this purpose Android clearly differentiates the terms application, process, task and thread. This chapter explains each term by itself as well as the correlation between the terms.

2.1 Processes & threads

Five types of processes are distinguished in Android in order to control the behavior of the system and it's running programs. The various types have different importance levels which are strictly ordered. The resulting importance hierarchy for process classes looks like this (descending from highest importance, from [22, Application Fundamentals]):

Foreground A process that is running an *Activity*, a *Service* providing the *Activity*, a starting or stopping *Service* or a currently receiving *BroadcastReceiver*.

Visible If a process holds a paused but still visible *Activity* or a *Service* bound to a visible *Activity* and no foreground components, it is classified a visible process.

Service A process that executes an already started *Service*.

Background An *Activity* that is no longer visible is hold by a background process.

Empty These processes contain no active application components and exists only for caching purposes.

If the system is running low on memory, the importance of a process becomes a crucial part in the system's decision which process gets killed to free memory.

Therefore empty processes are killed most likely followed by background processes and so on. Usually only empty and background processes are killed so the user experience stays unaffected. The system is designed to leave everything untouched as long as possible that is associated with a visible component like an *Activity*. [24]

Processes can contain multiple threads, like it is usual on Linux based systems. Most Android applications consist of multiple threads to separate the UI from input handling and I/O operations or long running calculations, hence the underlying processes are multi-threaded. The threads used on application level are standard Java threads running in the Dalvik VM.

2.2 Applications & tasks

Android applications are run by processes and their included threads. The two terms task and application are linked together tightly, given that a task can be seen as an application by the user. In fact tasks are a series of activities of possibly multiple applications. Tasks basically are a logical history of user actions, e.g. the user opens a mail application in which he opens a specific mail with a link included which is opened in a browser. In this scenario the task would include two applications (mail and browser) whereat there are also two *Activity* components of the mail application and one from the browser included in the task. An advantage of the task concept is the opportunity to allow the user to go back step by step like a pop operation on a stack.

2.3 Application internals

The structure of an Android application is based on four different components, which are: *Activity*, *Service*, *BroadcastReceiver* and *ContentProvider*. An application does not necessarily consists of all four of these components, but to present a graphical user interface there has to be at least an *Activity*.

Applications can start other applications or specific components of other applications by sending an *Intent*. These intents contain among other things the name of desired executed action. The *IntentManager* resolves incoming intents and starts the proper application or component. The reception of an *Intent* can be filtered by an application.

Services and broadcast receivers allow applications to perform jobs in the background and provide additional functionality to other components. Broadcast receivers can be triggered by events and only run a short period of time whereas a service may run a long time.

The compiled code of the application components and additional resources

like libraries, images and other necessary data is packed into a single .apk file that forms the executable Android application.

2.3.1 AndroidManifest.xml

All Android Dalvik applications need to have a XML document in the application's root directory called AndroidManifest.xml. This document is used by various facilities in the system to obtain administrative and organizational information about the application.

In the manifest file 23 predefined element types are allowed to specify among other things the application name, the components of the application, permissions, needed libraries and filters for intents and broadcasts. During development the manifest file holds the control information for instrumentation support (see section 6.2).

Some of the elements in the manifest file are discussed in more detail in other chapters, where it matches the context. Detailed information on all elements can be found at [22, The AndroidManifest.xml File].

2.3.2 Activities

An *Activity* is a single screen of an application like a browser window or a settings page. It contains the visual elements that present data (like an image) or allow user interaction (like a button). Each application can have multiple activities whereat the transition between the different activities is initiated via intents.

All activities are subclasses from *android.app.Activity* and their life cycle is controlled by the *onXYZ()* methods. This concept is needed by Android's way of handling multitasking and helps dealing with low memory situations (see section 2.3.6 for more detailed information about the life cycle). The main functions are:

onCreate() The initial method to set up an *Activity*.

onDestroy() The counterpart to *onCreate()*.

onResume() This method is called if the *Activity* is visible in the foreground and ready to get and process user input.

onPause() The method has to quickly save uncommitted data and stop CPU intensive work to prepare the *Activity* to lose the focus and going to background.

onRestart() This method has to restore a previously saved state of the *Activity*, as it is called after an activity was completely stopped and is needed again.

2.3.3 Intents, Intent filters and receivers

Unlike *ContentProviders*, the other three component types of an application (activities, broadcast receivers and services) are activated through intents. An *Intent* is an asynchronously sent message object including the message that should be transported.

The contained message either holds the name of the action that should be performed, or the name of the action being announced. The former applies to activities and services, the latter to broadcast receivers. The *Intent* class has some actions like *ACTION_EDIT* and *ACTION_VIEW* or for broadcasts *ACTION_TIME_TICK* included already. In addition to the action, the message contains an Uniform Resource Identifier (URI) that specifies the data used for the given action. Optionally the *Intent* object can hold a category, a type, a component name, extra data and flags. [22, Intents and Intent Filters]

Android utilizes different hooks in the application components to deliver the intents. For an activity, it's *onNewIntent()* method is called, at a service the *onBind()* method is called. Broadcast actions can be announced using *Context.sendBroadcast()* or similar methods. Android sends the *Intent* to the *onReceive()* method of all matching registered receivers.

Intents can be filtered by an application to specify which intents can be processed by the application's components. The list of filters is set in the application's manifest file, thus Android can determine the allowed intents before starting an application.

2.3.4 Content provider

The data storage and retrieval in Android applications is done via content providers. These providers can also be used to share data between multiple applications, given that the involved applications own the correct permissions to access the data. Android already has default providers for e.g. images, videos, contacts and settings which can be found in the *android.provider* package.

An application queries a *ContentResolver* which returns the appropriate *ContentProvider*. All providers are accessed like databases with an URI to determine the provider, a field name and the data type of this field. Applications only access content providers via a *ContentResolver*, never directly. If an application wants to store data that does not have to be shared, it can use a local *SQLiteDataBase*. Other available storage facilities are explained in section 5.1.4.

2.3.5 Background activities

Applications may need to perform some supporting operations in the background or without a graphical interface at all. Android provides the two classes *BroadcastReceiver* and *Service* for these purposes. If only a short operation has to be performed, the *BroadcastReceiver* is preferred and for long running jobs a *Service* is preferred.

Both classes do not necessarily imply that the background component runs as a thread or even in it's own process, hence Android does not enforce this kind of behavior. In order not to freeze a running application, the service or broadcast receiver usually is running in it's own thread, otherwise Android tends to kill such an application as it seems not responsive.

The *BroadcastReceiver* is activated through the *onReceive()* method and gets invalidated on return from this method. This makes it necessary to only use synchronous methods in a receiver. A broadcast normally is unordered and sent to all matching receivers at the same time, but it can be ordered too. In this case Android sends the broadcast only to the one receiver at a time. This receiver runs as usual and can forward a result to the next receiver or it can even abort the whole broadcast.

A *Service* allows an application to perform long running tasks in background and provide some of the application's functionality to other applications in the system. A service can be used in two ways, it is either started with one command or started and controlled by an incoming connection that makes use of Remote Procedure Calls (RPC, see section 2.4).

Both receiver and service have to be announced in the application manifest file to allow Android to determine the service's or receiver's class even if the application is not running.

2.3.6 Application lifetime & states

The state an Android application is in, is determined by the state of it's components, most importantly it's activities. As the application components alter their states, the application's underlying process type is adjusted.

On application start the individual components get started and in case of an activity the following hooks are called sequential: *onCreate()*, *onStart()*, *onResume()*. The first hook is only called once in an activities lifetime, but the other two methods can get called more often. If an activity loses the focus, the *onPause()* method is called and if the activity is no longer visible, *onStop()* is called. Before deleting an activity, it's *onDestroy()* method is run which ends the activity lifetime.

Each method gets called on a special event to allow the activity to preserve it's state or start and restart correctly. The following list describes the purpose of these

hooks and how the causing events change the application state. Figure 2.1 is the graphical representation of this list.

onCreate() This method is called for initialization and static set up purposes. It may get passed an older state for resuming. The next method is always *onStart()*.

onRestart() After an activity is stopped and about to be started again, this hook is

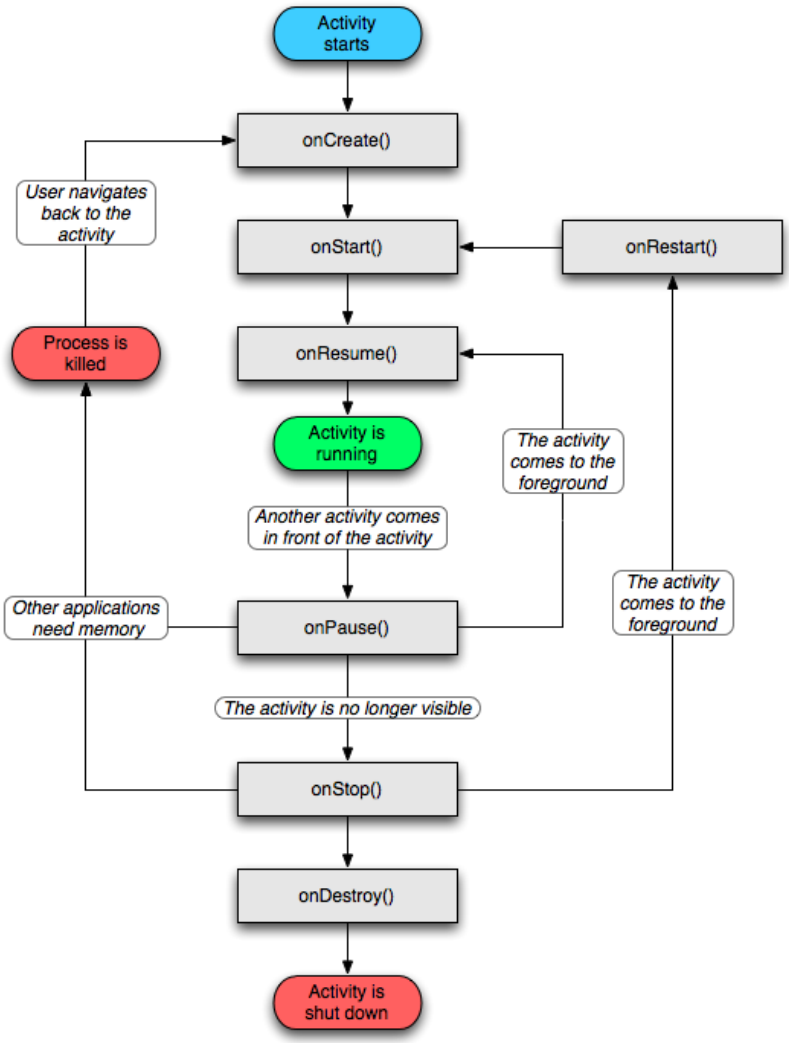

Figure 2.1: Life cycle graph of an *Activity*. The graph for a *Service* is similar. From [22, Application Fundamentals].

called and after it *onStart()*.

onStart() The application process type changes to visible and the activity is about to be visible to the user, but it's not in the foreground.

onResume() The activity has the focus and can get user input. The application process type is set to foreground.

onPause() If the application loses the focus or the device is going to sleep, this hook is called and the process type is set to visible. After running this hook, the system is allowed to kill the application at any time. All CPU consuming operations should be stopped and unsaved data should be saved. The activity may be resumed or get stopped.

onStop() The activity is no longer visible, the process type is set to background and the application may be killed at any time by the system to regain memory. The activity is either going to get destroyed, or restarted.

onDestroy() The last method that is called in an activity right before the system kills the application or the application deletes the activity. The application process may be set to empty if the system keeps the process for caching purposes.

For a service, the lifetime is simpler than the one of an activity, as the *onResume()*, *onPause()* and *onStop()* hooks do not exist. For interactive services, there are the *onBind()*, *onUnbind()* and *onRebind()* hooks that are called to start, stop and restart a service. The process type alters between *foreground* at creation and deletion time and *service* at run time. Broadcast receivers only have the *onReceive()* hook which runs at foreground process importance.

If an activity needs to save it's state to present the user the same exact state the activity was in when it was left, the *onSaveInstanceState()* method can be used for this purpose. This hook is called before the *onPause()* hook is called and allows to save the dynamic data as key-value pairs in a *Bundle*. The saved state object can be passed to *onCreate()* and *onRestoreInstanceState()* to restore the state of the activity. [24, 22, Application Fundamentals]

2.4 RPC

Android has a Common Object Request Broker Architecture (CORBA) and Component Object Model (COM) like lightweight RPC mechanism and brings it's own language, AIDL (Android Interface Definition Language). AIDL uses proxy classes to pass messages between the server and the client. The `aidl` tool creates

Java interfaces from the interface definition which have to be available at the client as well as at the server.

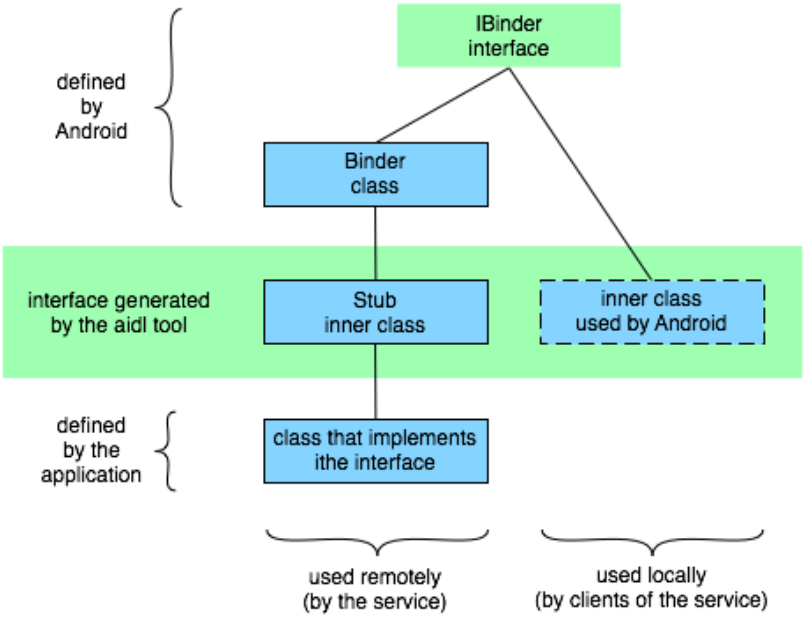

Figure 2.2: Android RPC class diagram showing the user space design of the mechanism. From [22, Application Fundamentals].

The created interface has an abstract inner class called *Stub* which has to be extended and implemented by the client and the server like shown in figure 2.2. If the server provides a *Service*, it has to return an instance of the interface implementation at it's *onBind()* method. Only methods are allowed in AIDL and all of them are synchronous. The mechanism is supported by the kernel's *binder* driver which is described in section 4.1.4.

2.5 Application security

The security model of Android heavily depends on the multi-user capabilities of the underlying Linux. Each application runs with it's own unique user id and in it's own process. All Dalvik applications run in a sandbox that by default prohibits e.g. communication with other processes, access to others data, access to hardware features like GPS or camera and network access.

Opposing to platforms with native binary executables, Android makes it easy to enforce a certain application behavior, as it's application VM Dalvik directly

controls code execution and resource access. Platforms like iOS, webOS, Symbian or MeeGo do not have this opportunity, thus their sandboxing systems are based on means on kernel-, filesystem- or process-level and some of these means are used by Android too.

The basic sandbox denies all requests from an application unless the permissions are explicitly granted. The permissions are set up in the application manifest file with the *<uses-permission>* tag. That allows the system to ask the user or a package manager upfront at install time for the application's wanted permissions. Once installed, an application can assume that all wanted permissions are granted.

During the installation process, an application is assigned with a unique and permanent user id. This user id is used to enforce permissions on process and file system level. An application can explicitly share files by setting the file mode to be world readable or writeable, otherwise all files are private. If two applications should share the same processes or use the same user id, both of the applications have to be signed with the same certificate and ask for the same *sharedUserId* in their manifest files.

The individual components of an application can also be restricted, to ensure that only certain sources are allowed to start an activity, a service or send a broadcast to the application's receiver. A service can enforce fine grained permissions with the *Context.checkCallingPermission()* method. Content providers can restrict overall read and write access as well as grant permissions on an URI basis.

A more detailed analysis of the security model can be found in [28, 10] and in [27] the opportunities of malware detection on Android are examined.

2.6 Native applications

Android applications that need more performance than the Dalvik VM can offer, can be partitioned. One part stays in the Dalvik VM to provide the application UI and some logic and the other part runs as native code. This way applications can take advantage of the device capabilities, even if Android or Dalvik do not offer a certain feature.

The native code parts of an application are shared libraries which are called through the Java Native Interface (JNI). The shared library has to be included in the applications .apk file and explicitly loaded. The code from the native library is loaded into the address space of the application's VM. This leads to a possible security hole, as the security means described in section 2.5 do not cover native code. [29, page 12 f.]

As of revision 4 of the Native Development Kit (NDK), the code can make use of the ARMv5TE and ARMv7-A instructions sets. In addition the Vector Floating Point (VFP) and Neon (Single Instruction Multiple Data instructions) extensions

can be used in ARMv7-A code.

Code used by native applications should only make use of a limited library set that amongst others includes the libc, libm, libz and some 2D and 3D graphics libraries (see [2]).

The performance speedup of native code over Dalvik interpreted code was examined in [16]. The results show that there is a huge difference between native and interpreted code, but the benchmarks were run on an early Android version that e.g. did not offer a just in time compiler.

Chapter 3

Dalvík VM

Android applications and the underlying frameworks are almost entirely written in Java. Instead of using a standard Java virtual machine, Android uses it's own VM. This virtual machine is not compatible to the standard Java virtual machine Java ME as it is specialized and optimized for small systems. These small systems usually only provide little RAM, a slow CPU and other than most PCs no swap space to compensate the small amount of memory.

At this point Android tells itself apart from other mobile operating systems like Symbian, Apple's iOS or Palm's webOS which use native compiled application code. The main programming languages used there are C, C++ and objective C, whereat e.g. webOS allows other mostly web based languages like JavaScript and HTML as well.

The necessary byte code interpreter – the virtual machine – is called Dalvik. Instead of using standard byte code, Dalvik has it's own byte code format which is adjusted to the needs of Android target devices. The byte code is more compact than usual Java byte code and the generated .dex files are small.

3.1 Design requirements

As a multitasking operating system, Android allows every application to be multi-threaded and also to be spread over multiple processes. For the sake of improved stability and enhanced security each application is separated from other running applications. Every application runs in a sandboxed environment in it's own Dalvik virtual machine instance. This requires Dalvik to be small and only add little overhead.

Dalvik is designed to run on devices with a minimum total memory of just 64 MB of RAM. For better performance actual devices have more than 64 MB installed. Of these 64 MB only about 40 MB remain for applications, libraries and

services. The used libraries are quite large and likely need 10 MB of RAM. Actual applications only have around 20 MB left of the 64 MB of RAM. This very limited amount of memory has to be used efficiently in order to run multiple applications at once. [17]

There are two major areas to consider for minimizing the memory usage. Firstly the application itself has to be as small as possible and secondly the memory allocation of each application has to be optimized. In addition to the reduced usage of valuable memory one gains faster application load times and less needed disk storage.

Most importantly the power supply by battery and the used CPUs confine the allowed and possible overhead for Dalvik. The typically used ARM CPUs only provide fairly limited computational power and small caches.

3.2 General & file optimizations

Java applications for Dalvik get compiled like other Java programs with the same compilers and mostly the same toolchain. Instead of compressing and packaging the resulting class files into a .jar file, they are translated into .dex files by the `dx` tool. These files include the Dalvik byte code of all Java classes of the application. Together with other resources like images, sound files or libraries the .dex files are packaged into .apk files.

In order to save storage space, .dex files only contain unique data. If multiple class files share the same string, this string would only exist once in the .dex file and the multiple occurrences are just pointers to this one string (see figure 3.1). The same mechanism is used for method names, constants and objects which results in smaller files with much internal “pointing”. The results of these means in terms of file size can be seen in table 3.1.

	System libraries	Browser app	Alarm clock app
Uncompressed	21445320 – 100%	470312 – 100%	119200 – 100%
Compressed Jar	10662048 – 50%	232065 – 49%	61658 – 52%
Dex file	10311972 – 48%	209248 – 44%	53020 – 44%

Table 3.1: Examples for file size reduction of .dex files. The file size is given in bytes and the .dex files are not compressed. The data is taken from [17].

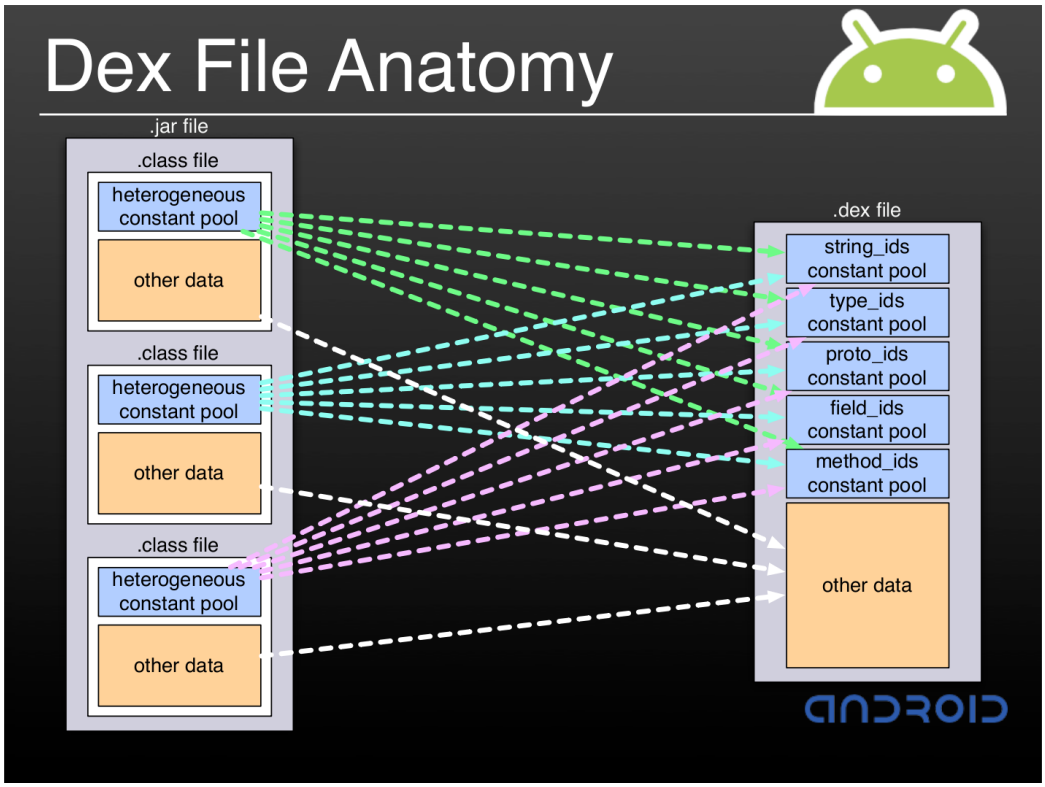

Figure 3.1: .dex file structure. Constant pools from classes get merged into one corresponded .dex file pool. From [17].

3.2.1 Byte code format

The Dalvik byte code is designed to reduce the number of necessary memory reads and writes and increased code density compared to Java byte code. For this purpose Dalvik uses it's own instruction set with a fixed instruction length of 16 bit. Furthermore Dalvik is a register based virtual machine which among other things reduces the needed code units and instructions. The given register width is 32 bit which allows each register to hold two instructions, 64 bit values are hold in adjacent registers. Instructions shorter than 16 bit zero-fill the unused bits and pad to 16 bit.

Dalvik knows 256 different op codes whereof 23 are unused in Android 2.2, leading to an actual total op code number of 233. Optimizations of the byte code is mostly done by the `dexopt` tool at installation time. [13, 15]

3.2.2 Install time work

For the start of an application, the associated .dex file has at least to be verified for structural integrity. This verification can take some time and is needed only once, as application files don't change unless the application is updated or uninstalled. Because of that, the verification process can be done either at the first startup or during the installation or update process. In Android this verification process is done at installation time, so that at all application startups later only e.g. a checksum of the .dex file is needed to verify it.

Verification

The verification and optimization procedure is performed by the `dexopt` program. To verify a .dex file, `dexopt` loads all classes of the file into a briefly initialized VM and runs through all instructions in all methods of each class. This way illegal instructions or instruction combinations can be found before the application actually runs for the first time. Classes in which the verification process succeeded get marked by a flag to avoid rechecking this class again. In order to check the .dex files integrity, a CRC-32 checksum is stored within the file. [13]

Optimization

After the successful verification of a .dex file, it gets optimized by `dexopt`. The optimizations aim at performance increase through reduced code size or reduced code complexity. The optimization mechanisms heavily depend on the target VM version and the host platform which makes it hard to run `dexopt` elsewhere than on the host. The resulting code is unoptimized Dalvik byte code mixed with optimized code using op codes not defined in the Dalvik byte code specification. If necessary for the processor architecture endianness, the code is byte swapped and aligned accordingly.

For code reduction `dexopt` prunes all empty methods and replaces them with a *no-operation* op code. Inlining some very often called methods like *String.length()* reduces the method call overhead and virtual method call indices get replaced by vtable indices. Furthermore field indices get replaced by byte offsets and short data types like char, byte and Boolean are merged into 32 bit form to save valuable CPU cache space. If it is possible to pre-compute .dex file data, `dexopt` appends the resulting data which reduces the CPU time needed by the executing VM later.

Both verification and optimization are limited to process only one .dex file at a time which leads to problems at handling dependencies. If a .dex file is optimized, it contains a list of dependencies which can be found in the bootstrap class path. In order to guarantee consistency in case of exchanged .dex files, only dependencies

to .dex files in the bootstrap class path are allowed. This way the verification of methods depending on external .dex files other than those in bootstrap will fail and the related class will not be optimized. [13]

3.3 Optimizations of memory allocation and usage

In Dalvik there are four different kinds of memory to distinguish that can be grouped to clean/dirty and shared/private. Typical data residing in either shared or private clean memory are libraries and application specific files like .dex files. Clean memory is backed up by files or other sources and can be pruned by the kernel without data loss. The private dirty memory usually consists of the applications heap and writeable control data structures like those needed in .dex files. These three categories of different memory are quite common and no specialty of Dalvik. [17]

3.3.1 Zygote

Shared dirty memory is possible through a facility of Dalvik called Zygote. It is a process which starts at boot time and is the parent of all Dalvik VMs in the system. The Zygote loads and initializes classes that are supposed to be used very often by applications into it's heap. In shared dirty memory resides e.g. the dex data structures of libraries.

After the startup of the Zygote, it listens to commands on a socket. If a new application starts, a command is sent to the Zygote which performs a standard *fork()*. The newly forked process becomes a full Dalvik VM running the started application. The shared dirty memory is "copy-on-write" memory to minimize the memory consumption. [17]

3.3.2 Garbage collection

The garbage collector (GC) in Android runs in each VM separately, therefore each VMs heap is garbage collected independently. The Zygote process and the concept of shared dirty memory requires the GC data structures ("mark bits") to not be tied to the objects on the heap, but kept separate. If the mark bits would lie next to the objects on the heap, a run of the GC would touch these bits and turn the shared dirty memory into private dirty memory. In order to minimize the private dirty memory the needed mark bits are allocated just before a GC run and freed afterwards.

3.4 JIT

Dalvik was designed to be a simple interpreter without the capability to perform Just In Time (JIT) compilations. Android 1.0 was announced explicitly without a JIT, because it was seen as being too memory consuming and with a relatively small performance boost. With the recent release of Android 2.2 the platform got a JIT for ARM processors which should have a low memory footprint and provide a notable performance boost.

Android applications often call native compiled and optimized libraries to perform performance critical tasks which leads to only about $\frac{1}{3}$ of all executed code to be interpreted by the virtual machine. Therefore the achievable speedup is limited and also depends heavily on the benchmarked application. [19]

3.4.1 Types of JITs & Android's JIT

In the wide field of just in time compilers the method based and the trace based JITs are especially interesting to Android. Method based JITs compile a whole method to native code whereas trace based JITs only compile one call path. The advantages and disadvantages of these two compiler categories are listed below:

Method based JIT

+ large optimization window

+ easy to sync JIT and interpreter at method boundaries

- waste of space and time on code that is not run often or not run at all

- high memory usage for compilation

- takes long time to benefit

Trace based JIT

+ no method boundaries for optimization

+ only "hot" code is compiled

+ at exceptions just rollback and start interpreter

- limited optimization window of trace

- much overhead if the transition between JIT and interpreter is expensive

- sharing code between CPUs or Threads is hard

Android's JIT

Due to memory constrains the JIT used in Android is trace based and the trace length is only 100 op codes short. [14] The JIT and the traditional interpreter are tightly geared. In case of an exception in the code from the JIT, the interpreter takes over, resets the state to the beginning of the trace and runs the code sequence again without the JIT. This mechanism is extended to allow parallel execution of JIT code and interpreted code to verify the JIT against the interpreter.

The Android JIT uses trace caches per process so that threads in the same application share the trace cache. The cache size is configurable during build time while it's default is about 100 KB. In order to reduce overhead and increase performance, the JIT is able to chain consecutive traces to avoid the need of switching back to the interpreter.

The code generated by the JIT is either normal ARM code, Thumb or Thumb-2 code and in special cases the ARM VFP extensions are used for floating point operations. The code model used by the JIT can be configured at build time and is adjusted to the capabilities of the target device CPU. The Thumb-2EE instructions are not used even though they are designed to accelerate code generated by JITs. Android takes advantage of interleaving and rearranging byte code instructions to optimize the performance which contradicts Thumb-2EE's concept of accelerating just single instructions. [19]

3.4.2 Future of Android's JIT

The Android 2.2 release only offers a basic trace JIT which is bound to one architecture and it is limited in it's optimization options. In future versions of Android the architecture support will be extended to e.g. x86 and the scope of optimizations will be enlarged. The existing options will be fine tuned like e.g. the trace length will be enlarged and whole methods might be in-lined.

Apart from platform specific enhancements the JIT itself could be supported by persistent profiling information stored in the applications. Traces of frequently used methods in system libraries could be stored permanently. These pieces of code could also be generated by a method based JIT to obtain faster code. Optimizations like this could be done in background while the device is not in use and charging. [19]

Chapter 4

Power management & kernel

The kernel used in Android is a 2.6 series Linux kernel modified to fulfill some special needs of the platform. The kernel is mostly extended by drivers, power management facilities and adjustments to the limited capabilities of Android's target platforms. The power management capabilities are crucial on mobile devices, thus the most important changes can be found in this area. [31]

Like the rest of Android, the kernel is freely available and the development process is visible through the public Android source repository. [12] There are multiple kernels available in the repository like an architecture unspecific common kernel and an experimental kernel. Some hardware specific kernels for platforms like MSM7xxx, Open Multimedia Application Platform (OMAP) and Tegra exist in the repository too.

4.1 Differences to mainline

The changes to the mainline kernel can be categorized into: bug fixes, facilities to enhance user space (lowmemorykiller, binder, ashmem, logger, etc.), new infrastructure (esp. wake locks) and support for new SoCs (msm7k, msm8k, etc.) and boards/devices. [31]

The Android specific kernels and the mainline Linux kernel are supposed to get merged in the future, but this process is slow and will take some time. [21]

4.1.1 Wake locks

Android allows user space applications and therefore applications running in the Dalvik VM to prevent the system from entering a sleep or suspend state. This is important because by default, Android tries to to put the system into a sleep or better a suspend mode as soon as possible. Applications can assure e.g. that the

screen stays on or the CPU stays awake to react quickly to interrupts. The means Android provides for this task are wake locks.

Wake locks can be obtained by kernel components or by user space processes. The user space interface to create a wake lock is the file */sys/power/wake_lock* in which the name of the new wake lock is written. To release a wake lock, the holding process writes the name in */sys/power/wake_unlock*. The wake lock can be furnished with a timeout to specify the time until the wake lock will be released automatically. All by the system currently used wake locks are listed in */proc/wake locks*.

The kernel interface for wake locks allows to specify whether the wake lock should prevent low power states or system suspend. A wake lock is created by *wake_lock_init()* and deleted by *wake_lock_destroy()*. The created wake lock can be acquired with *wake_lock()* and released with *wake_unlock()*. Like in user space it is possible to define a timeout for a wake lock. [23, Power Management]

The concept of wake locks is deeply integrated into Android as drivers and many applications make heavy use of them. This is a huge stumbling block for the Android kernel code to get merged into the Linux mainline code. [21]

4.1.2 Power manager

The *PowerManager* is a service class in the application framework that gives Dalvik VM applications access to the *WakeLock* capabilities of the kernel power management driver. Four different kinds of wake locks are provided by the *PowerManager*:

PARTIAL_WAKE_LOCK The CPU stays awake, even if the device's power button is pressed

SCREEN_DIM_WAKE_LOCK The screen stays on, but is dimmed

SCREEN_BRIGHT_WAKE_LOCK The screen stays on with normal brightness

FULL_WAKE_LOCK Keyboard and screen stay on with normal back light

Only the *PARTIAL_WAKE_LOCK* assures that the CPU is fully on, the other three types allow the CPU to sleep after the device's power button is pressed. Like kernel space wake locks, the locks provided by the *PowerManager* can be combined with a timeout. It is also possible to wake up the device when the wake lock is acquired or turn on the screen at release. [22, PowerManager class overview]

4.1.3 Memory management

The memory management related changes of the kernel aim for improved memory usage in systems with a small amount of RAM. Both *ashmem* and *pmem* add a new way of allocating memory to the kernel. Ashmem can be used for allocations of shared memory and pmem allows allocations of contiguous memory.

ASHMEM The Anonymous/Android Shared Memory provides a named memory block that can be shared across multiple processes. Other than the usual shared memory, the anonymous shared memory can be freed by the kernel. To use ashmem, a process opens */dev/ashmem* and performs *mmap()* on it.

PMEM Physical Memory enables e.g. drivers and libraries to allocate named physically contiguous memory blocks. This driver was written to compensate hardware limitations of a specific SoC – the MSM7201A. [30]

Low memory killer The standard Linux kernel out of memory killer (oom killer) utilizes heuristics and compute the process' "badness" to be able to terminate the process with the highest score in a low memory situation. This behavior can be inconvenient to the user as the oom killer may close the user's current application when there are other processes in the system, that do not affect the user.

Android's *lowmemory* driver starts early before a critical low memory situation occurs and informs processes to save their state. If the low memory situation worsens, *lowmemory* starts to terminate processes with low importance who's state was saved. The used importance levels can be found in section 2.1.

4.1.4 Other changes

In addition to the already described kernel changes, there are various other changes that touch miscellaneous areas of the kernel. A few minor changes are described in this section. [3]

binder Unlike in the standard Linux kernel, the IPC mechanism in Android's kernel is not System V compatible. The description in section 2.4 shows the user space side of the IPC mechanism and the underlying concept. The kernel side of the mechanism is based on OpenBinder (see [9]), thus focused on being light weight. In order to enhance performance, the *binder* driver uses shared memory to pass the messages between threads and processes. [4, 18]

logger Extended kernel logging facility with the four logging classes: main, system, event and radio. The application framework uses the system class for it's log entries. [4]

early suspend Drivers can make use of this capability to do necessary work, if a user space program wants the system to enter or leave a suspend state. [23, Power Management]

alarm "The alarm interface is similar to the hrtimer interface but adds support for wake up from suspend. It also adds an elapsed realtime clock that can be used for periodic timers that need to keep running while the system is suspended and not be disrupted when the wall time is set." [1]

4.2 Device & platform support

Android's native platform is ARM, but there exist ports to other platforms like MIPS and x86 as well. The number of ports grows as Android becomes more and more popular. The list of supported devices grows steadily too, as there are more and more Android powered devices on the market, especially smartphones with ARM SoCs. Some of those device configurations can be found in the main repository, while others are made available by manufacturers like HTC or Motorola.

Porting Android to a new architecture mostly means porting Dalvik, as the Linux kernel is probably already running on the new platform. The core libraries of Dalvik rely on other libraries like OpenSSL and zlib which have to be ported in the first place. After that, the JNI Call Bridge has to be ported according to the C calling conventions of the new architecture. The last step is porting the interpreter which means implementing all Dalvik op codes. [23, Porting Dalvik]

Chapter 5

Application Framework APIs

The application framework of Android provides APIs in various areas like networking, multimedia, graphical user interface, power management and storage access. The libraries in the framework are written in Java and run on top of the core libraries of the Android Runtime (see figure 1.1). These core libraries utilize and encapsulate optimized native system libraries like libc, libssl and FreeType.

The Application Framework provides managers for different purposes like power management, resource handling, system wide notification and window management. Applications are supposed to use the services of the managers and not use the underlying libraries directly. This way it is possible for the managers to enforce application permissions through the means of the sandbox permission system (see section 2.5). The managers can ensure that an application is allowed to e.g. initiate a phone call or send data over the network.

5.1 APIs

Given the vast number of APIs in Android, only a few parts are covered in this section and some miscellaneous APIs are pictured briefly at the end of the section. A comprehensive list of Android's APIs can be found at [22, Package Index]

API levels

As the development of Android continues, new APIs are added and old APIs get marked obsolete and are eventually erased. Each Android version has it's own API level and applications can define a minimum, maximum and preferred API level in their manifest file. The API level changes between major as well as minor releases of Android, as shown in table 5.1.

Platform Version	API Level
Android 2.2	8
Android 2.1	7
Android 2.0.1	6
Android 2.0	5
Android 1.6	4
Android 1.5	3
Android 1.1	2
Android 1.0	1

Table 5.1: The API levels of all Android versions. Data taken from [22, Android API Levels].

5.1.1 User interface

The basis of a graphical user interface in Android is the *View* class. All visible elements of a user interface and some invisible items are derived from this class. Android provides standardized UI elements like buttons, text and video views or a date picker. View items can be partitioned to a *ViewGroup* which allows applying a layout to the views in the group.

In order to interact with the UI, the *View* class provides hook methods like *onClick()*, *onTouch()* and *onKey()* which are called by the underlying framework. UI events can also be received by listeners that provide the *onXYZ()* method and are registered like e.g. the *OnKeyListener*.

In addition to the described UI elements, the framework provides menus and dialogs. Menus can either be options menus that can be accessed via the menu key, or they are context menus of a view. To inform the user or to obtain input, an application can present a dialog. Android provides dialogs for date and time picking, progress notification and a customizable general dialog with buttons.

For user notification exist three different mechanisms in Android. The most intrusive one is the notification with a dialog which moves the focus to the dialog, leaving the application in the background. The toast notification displays a text on top of the current application that allows no interaction and disappears after a given timeout. The status bar notification leaves the current application appearance unaffected and puts an icon in the status bar. The toast notification as well as the status bar notification can be utilized by background services. [22, User Interface]

5.1.2 Media framework

The media framework supports multiple audio, video and image data formats and has a media player and an encoder built into Android. The number of data formats can be extended, but as a minimum the following decodeable formats are supported [22, Android Supported Media Formats]:

Image JPEG (encoder provided), PNG, GIF and BMP

Audio MP3, OGG Vorbis, MIDI, PCM/WAVE, AAC, AAC+ and AMR (encoder provided)

Video H.263 (encoder provided), H.264 and MPEG-4

The *MediaPlayer* and *MediaRecorder* classes can be used to play back and record the supported multimedia data. In addition to the player and recorder, the *android.media* package includes specialized classes to play alarms and ring tones or generate image thumbnails and set up the camera. [22, Audio and Video]

5.1.3 Network

Access to networks and especially the Internet is crucial for Android devices, as many applications depend on network access. Most Android devices have multiple technologies on board to gain network access. Smartphones at least have Internet access via GPRS (2G telephony) or UMTS and usually WiFi is available too.

The *ConnectivityManager* allows applications to check the status of the different access technologies and it informs applications via broadcasted intents about connectivity changes. To make use of the network connection, Android provides a broad range of packages and classes. [22, ConnectivityManager Class Overview]

java.net.* The standard Java network classes like sockets, simple HTTP and plain packets

android.net.* Extended java.net capabilities

android.net.http.* SSL certificate handling

org.apache.* Specialized HTTP

android.telephony.* GSM & CDMA specific classes, send text message, signal strength and status information

android.net.wifi WiFi configuration and status

Bluetooth

Android's Bluetooth APIs allows applications to search for other devices, pair with them and exchange data. In order to use the Bluetooth capabilities, an application needs to have the *BLUETOOTH* or *BLUETOOTH_ADMIN* permission in it's manifest file granted. The Bluetooth API is located in the *android.bluetooth* package.

The connection via a RFCOMM (RS-232 serial line via Bluetooth) compatible *BluetoothSocket* is initiated by a local *BluetoothAdapter* and targeted to a remote *BluetoothDevice*. Making the device visible for scans and pairing devices need user interaction as the permission system asks for granting the needed permissions. [22, 23, Bluetooth]

5.1.4 Storage & backup

Applications can store and retrieve their data in various ways [22, Data Storage]:

SharedPreferences A class that provides storage for key/value pairs of primitive data types. The data can be shared between multiple clients in the same process. All data can only be modified through an *Editor* object to ensure consistency even if the application is terminated.

Internal/external storage If an application wants to make sure that no other application or even the user can access saved data, it can write this data to the internal storage. Files saved to the internal storage can be set up to allow others to access the files. Files that are supposed to be shared and/or user visible, can be stored to the external storage. Files in this storage area are always publicly visible and accessible. Android allows the external storage to be turned into an USB mass storage with full access to all files in this storage area.

Database storage The SQLite database files behind all content providers are available to applications as an application private database back end.

Network storage With network access available, Android applications can obtain and store their data via sockets, HTTP connections and other means from *java.net.** and *android.net.**.

Application data backup

From version 2.2 on Android provides a mechanism to backup preferences and application data on a remote site – the cloud. Applications can request a backup and Android automatically restores the data on an application reinstall, if the

application is installed on a new device or by request of the application. The mechanism consists of three parts on the device: The backup agent, transport and manager.

Applications can announce a *backup agent* in the manifest file and implement it to provide hooks for the *backup manager*. If application data changes, the application can inform the backup manager which may schedule a backup and call the according backup agent hook. The backup mechanism does not allow on demand read and write access to the backed up data as it's always the backup manager's decision when to perform a restore or a backup.

The *backup transport* is responsible to transfer the data from or to the remote site. There is no guarantee that the backup capability is available on a device as the backup transport depends on the device manufacturer and the service provider. Furthermore there is no security assurance for the data in the backup. [22, Data Backup]

5.1.5 Other APIs

Supplementary to the already mentioned APIs, Android provides a wide range of other APIs for many different areas. Some of those packages are listed below:

Location & Maps Applications can use the *LocationManager* and classes from *android.location* to obtain the device's location directly on demand or by a broadcast. The external *com.google.android.maps* package provides mapping facilities for applications. [25]

Search The system wide search on Android devices includes not only contacts, web search and such, but can also be extended by applications to make their content provider data searchable. The *SearchManager* provides unified dialogs and extra functionality like voice search for all applications.

WebKit For browsing web pages, Android provides among other things a WebKit based HTML renderer, a JavaScript engine (V8) and a cookie manager. These facilities can be used by applications to provide browser functionality inside the application.

Speech The *android.speech* package puts applications in the position to make use of server side speech recognition services. A text-to-speech API provides the means to turn text into audio files or play back the result directly.

C2DM Cloud to Device Messaging – An new and in Android 2.2 fairly limited capability to send data like URLs or even intents to the device. [25]

Chapter 6

Testing & debug

Android's SDK provides a rich set of tools and facilities for developing and debugging new applications. There is an Eclipse plugin that integrates some of the supplied tools into the IDE. The Android emulator allows testing and debugging an application without the need of an actual Android device. The Android Debug Bridge (ADB) allows to log in to the emulator or an actual device and start a shell, view logs or control and manipulate the connected system in various ways.

Additional to the tools there is a testing and instrumentation framework available. This framework is partly based on JUnit tests and extends them to cooperate with the Android infrastructure. Android 2.2 allows the end user to send a crash report to the developer which contains among other things a stack trace. This feature is available through the Android Market, thus it's only an option for applications available at the Android Market.

6.1 Tools

The tools in the SDK include maintenance programs as well as debugging and testing tools. Some of the tools are described in this section while others like the `dx` and the `zipalign` tool are too specific or described elsewhere e.g. `dx` in the Dalvik chapter 3.

ADB The Android Debug Bridge is an important tool which supports a variety of commands in different categories. ADB consists of three parts, the `adb` server and client on the development computer and daemon processes on the target machines like an emulator or an Android device. The client can send commands to the server and depending on the type, the command is passed to the daemons or is executed locally. ADB also has the ability to handle multiple devices at the same time.

Commands are available in different categories and for multiple purposes like data exchange, system control, maintenance or information retrieval. Some of the available commands are: `logcat` (view system log), `shell` (open a shell on the target device), `install` (install an application) and `bugreport` (list information for a bug report). The Eclipse plugin makes use of ADB and integrates some of the commands in the IDE. [22, Android Debug Bridge]

DDMS For debugging the Dalvik Debug Monitor Server (DDMS) provides a graphical interface which combines a part of the ADB functionality with detailed information about the currently running Dalvik applications. Furthermore DDMS allows to change the state of the targeted system.

As DDMS sits on top of ADB, it too allows data exchange, port forwarding and listing and filtering the system log. The main functionality of DDMS lies in the detailed live analysis of running Dalvik VMs and the capability of mocking system states and events like changes in radio reception, incoming phone calls or setting a location. [22, Using the Dalvik Debug Monitor]

Running Dalvik VMs can be analyzed as of their heap, thread status, memory allocation and garbage collection. DDMS provides detailed heap object statistics (shown in figure 6.1) and allows exporting the data to HPROF files. Furthermore DDMS can be used to start traces and gather trace information used by the `traceview` tool.

Traceview / dmtracedump The graphical tool `traceview` displays recorded application traces and offers navigation through the time lines of multiple application threads (see figure 6.2). Traces for this tool can be invoked either by DDMS or by the application itself utilizing *startMethodTracing()* of the *android.os.Debug* class. The command line program `dmtracedump` takes the same traces as `traceview` and generates a call stack graph from them. [22, Traceview: A Graphical Log Viewer]

6.2 Instrumentation & JUnit tests

Android's capabilities for automated testing cover unit tests and instrumentation tests. The unit tests are based on Java JUnit tests and for instrumentation exists a framework which allows detailed tests of activities. The unit tests can test the whole application behavior or isolate components like a *ContentProvider*. With instrumentation it is possible to perform similar tests for a single *Activity* as for application tests with unit testing. All these tests can be found in the *android.test* package.

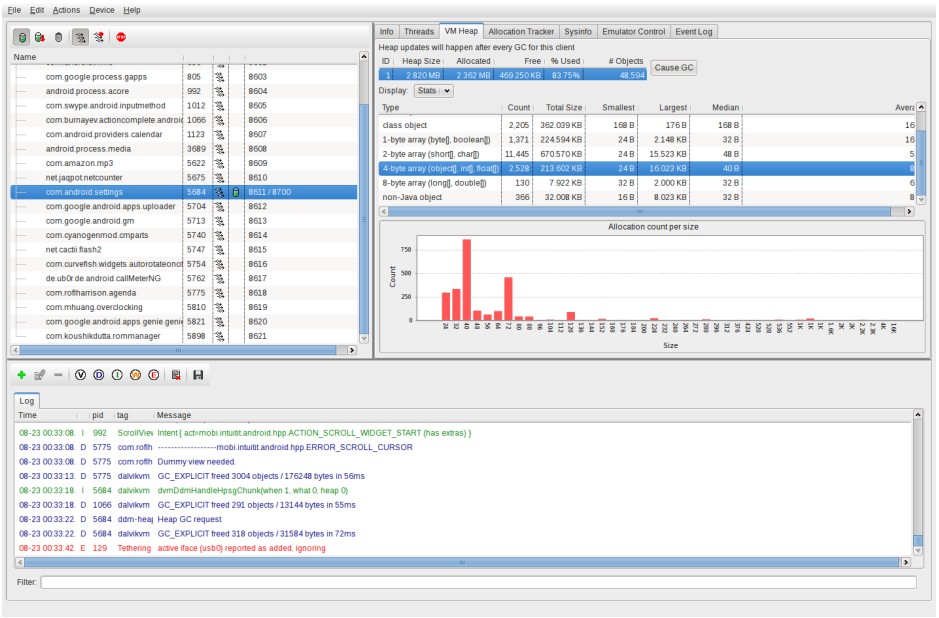

Figure 6.1: DDMS main view showing statistics from heap garbage collection of a running device.

Like the tested application, the test itself is an Android application. The test is linked to the application by an <*instrumentation*> entry in the application manifest file. Starting a test can be done by command line, an `adb` command, an Ant build target or the Eclipse plugin. The test will start before the application starts to allow the setup of mocked environments. The test is started and controlled by the *InstrumentationTestRunner* class which starts the application and it's related test in the same process. This facility is used even if a test does not explicitly uses instrumentation. [22, Testing and Instrumentation]

6.2.1 JUnit tests

The unit tests allow to change the *Context* of a whole application and thereby e.g. isolate certain parts of the application for detailed testing. The test classes are derived from the base class *AndroidTestCase*.

ApplicationTestCase A class to test a whole application in a definable environment. The application's *onCreate()* is not called until the test application calls *createApplication()*. The test cases' *tearDown()* method is called automatically which will call the application's *onDestroy()* method. This class allows to mock the *Context* of the application before calling *createApplication()*.

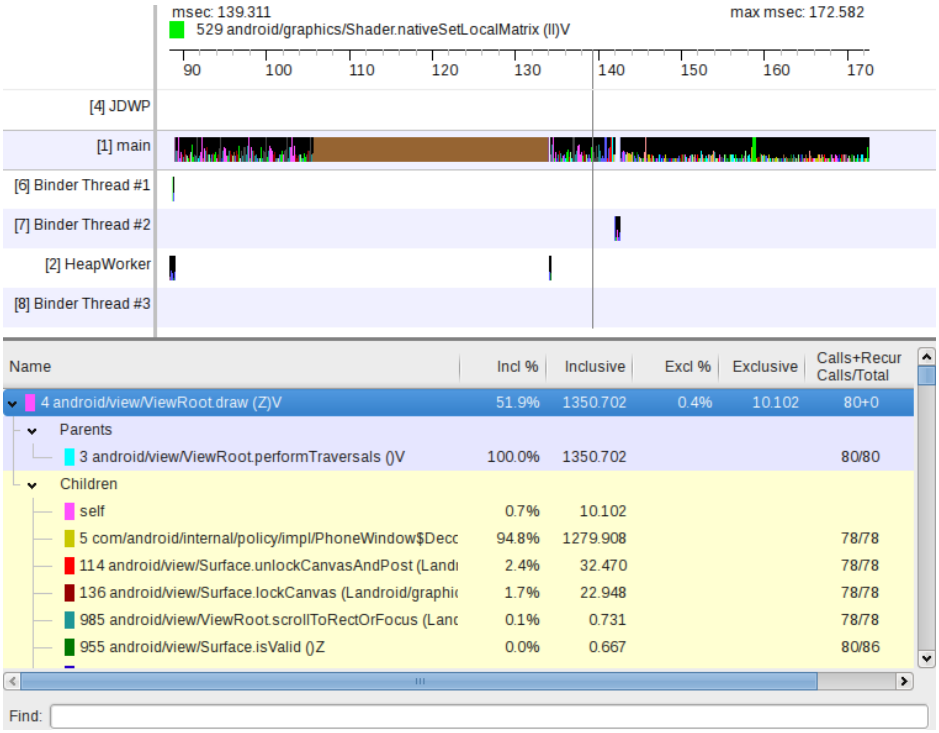

Figure 6.2: Traceview displaying an application trace. In the upper panel the thread activity is shown and in the lower panel the corresponding function calls and their statistics are listed.

ProviderTestCase2 A test class to test a single *ContextProvider* by picking a provider and using a mocked *Context* to isolate it.

ServiceTestCase A service test class similar to *ApplicationTestCase* which allows mocking or modifying the *Context*. The application's *onCreate()* is called after invocation of the *startService()* or *bindService()* method.

6.2.2 Instrumentation tests

Tests in this category can be used to test an *Activity* of a given application. To achieve this, the tests send mocked UI events and intents to the tested application while monitoring it. The test classes are derived from *ActivityTestCase* which is a child class of *InstrumentationTestCase*. The instrumentation framework design is shown in figure 6.3.

SingleLaunchActivityTestCase A class that can be used if there is only one *Activity* which covers all methods that should be tested, so that *setUp()* and

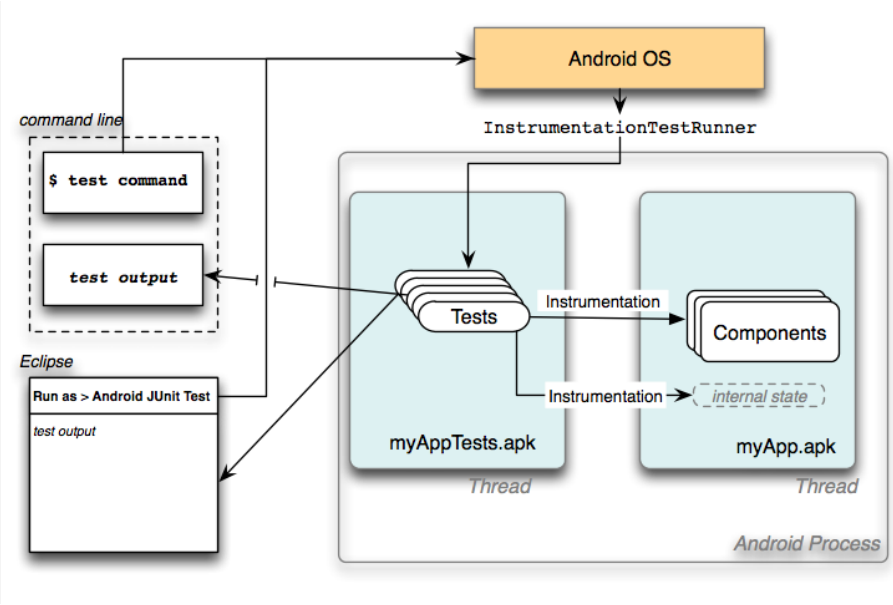

Figure 6.3: Instrumentation framework structure with it's different components on- and off-site the device. From [22, Testing and Instrumentation].

tearDown() is only called once.

SyncBaseInstrumentation This class allows testing the synchronization of a *ContentProvider*.

ActivityUnitTestCase The equivalent class to the unit test class for applications. It can mock contexts and even an application, but no intents.

ActivityInstrumentationTestCase2 A class which allows testing of a single *Activity* with mocked intents and mocked UI events, but with an otherwise normal environment.

6.2.3 Assert classes

Android provides extended JUnit Assert classes which are more powerful than the standard Assert class or provide special functionality.

MoreAsserts An extended standard class with improved support for regular expressions.

ViewAsserts A class specialized for UI testing with methods for alignment and position testing.

6.2.4 Mock object classes

There are some classes in the testing framework which allow easy setup of mocked contexts, resources, intents and such. These classes can be found in the *android.test* and *android.test.mock* package.

IsolatedContext A Class for unit testing by isolating an application.

RenamingDelegatingContext A class delegating to the provided context and redirected database and file operations to renamed counterparts.

Other mockable classes MockApplication, MockContentResolver, MockContext, MockDialogInterface, MockPackageManager, MockResources.

Chapter 7

Summary

With the size and the change rate of a platform like Android, it is almost impossible to present all aspects of such a system, hence this study thesis can only provide a first glance at Android.

It looks like Android is just another mobile operating system, but the wide support from large companies and especially Google have made Android one of the important contestants in the mobile sector. The openness and extensibility allow manufacturers to modify the system to fit their needs, both in hardware as well as in software. This leads to a significant number of devices from various manufacturers and each manufacturer covers a different range of customers.

The entry barrier for application developers is lower than on other platforms, as the Java programming language, as well as the Eclipse IDE are wide spread and often known already. The API documentation and the developer guide (both can be found at [2, 22]) are detailed, enriched with examples and continuous extended. Like for webOS applications there is a browser based application development environment that helps creating applications easily and without the need of programming skills. [5]

One controversial aspect of Android is the version fragmentation of the platform. Device manufacturers take a while to adopt a new Android version to already released devices or don't provide an update at all. This leads to a situation in which multiple Android versions become the the latest version for different devices, thus multiple Android versions are "current" and need to be supported.

The fact that the platform development is mainly driven by Google may raise concerns about being biased by the companies commercial interests. In extension to the possible fragmentation, the question of the benefits and the degree of Android being free and open is raised. [20]

7.1 Outlook

In the relatively short period of it's existence, Android made a lot of progress in order to adopt and adjust the feature set to those of other available platforms. The interval between new releases is about to be enlarged and fixed to a roughly six month release cycle and later once a year. [26]

New features like the cloud to device messaging and the backup management are most likely to be extended and used in a wider range of applications. Other features might increase hardware support for new device classes with larger screens like tablets. The number of devices operating Android is most likely going to increase notedly, as the range of devices is extended virtually on a weekly basis.

The development itself may broaden as more hardware and software companies get involved in the project. Additionally the development may get more support by people from outside the Android project like it would happen on a merge with the mainline Linux kernel.

Bibliography

[1] Google Inc.: *Alarm header file*. http://android.git.kernel.org/?p=kernel/common.git;a=blob;f=include/linux/android_alarm.h;h=f8f14e793dbf635cea8ebb8674d8f6c0d5c9d918;hb=android-2.6.32

[2] Google Inc.: *Android – An Open Handset Alliance Project*. http://developer.android.com

[3] eLinux.org - Embedded Linux Wiki: *Android Kernel Features*. http://elinux.org/Android_Kernel_Features

[4] Google Inc.: *Android staging driver*. http://android.git.kernel.org/?p=kernel/common.git;a=tree;f=drivers/staging/android;h=2f7d01f2a4b322e941b7fedd6c9c6d13fb59fa90;hb=android-2.6.32

[5] Google Inc.: *App inventor for Android*. http://appinventor.googlelabs.com/about/

[6] Open Handset Alliance: *Industry Leaders Announce Open Platform for Mobile Devices*. Press release. http://www.openhandsetalliance.com/press_110507.html

[7] Open Handset Alliance: *Open Handset Alliance Releases Android SDK*. Press release. http://www.openhandsetalliance.com/press_111207.html

[8] Open Handset Alliance: *Open Handset Alliance website*. http://www.openhandsetalliance.com

[9] PalmSource Inc.: *OpenBinder*. http://www.angryredplanet.com/~hackbod/openbinder/docs/html/index.html. Version: 2005

[10] INSTITUTE OF MANAGEMENT SCIENCES PESHAWAR – SECURITY ENGINEERING RESEARCH GROUP: Analysis report on Android Application Framework and existing Security Architecture. 2010. – Forschungsbericht. – http://imsciences.edu.pk/serg/wp-content/uploads/2010/05/Android-application-framework-and-security-architecture.pdf

[11] Google Inc.: *Android Open Source Project*. http://source.android.com. Version: 2010

[12] Google Inc.: *Android Repository*. http://android.git.kernel.org. Version: 2010

[13] THE ANDROID OPEN SOURCE PROJECT (Hrsg.): *Dalvik documentation in git repository*. The Android Open Source Project, http://android.git.kernel.org/?p=platform/dalvik.git;a=tree;f=docs. – Used version is "froyo-release"

[14] THE ANDROID OPEN SOURCE PROJECT (Hrsg.): *JIT Header file*. The Android Open Source Project, http://android.git.kernel.org/?p=platform/dalvik.git;a=blob;f=vm/interp/Jit.h;h=9d17a52687efaab5100eae082b5a885aa0391462;hb=froyo-release

[15] THE ANDROID OPEN SOURCE PROJECT (Hrsg.): *Opcode header file*. The Android Open Source Project, http://android.git.kernel.org/?p=platform/dalvik.git;a=blob;f=libdex/OpCode.h;h=58d17026bb8418fbd49111cc9de99d775815369e;hb=froyo-release

[16] BATYUK, Leonid ; SCHMIDT, Aubrey-Derrick ; SCHMIDT, Hans-Gunther ; CAMTEPE, Ahmet ; ALBAYRAK, Sahin: Developing and Benchmarking Native Linux Applications on Android. In: *MobileWireless Middleware, Operating Systems, and Applications*, 2009, 381–392

[17] BORNSTEIN, Dan: *Dalvik VM Internals*. Google I/O conference 2008 presentation video and slides. http://sites.google.com/site/io/dalvik-vm-internals

[18] BRADY, Patrick: *Anatomy & Physiology of an Android*. Google I/O conference 2008 presentation video and slides. http://sites.google.com/site/io/anatomy--physiology-of-an-android

[19] CHENG, Ben ; BUZBEE, Bill: *A JIT Compiler for Android's Dalvik VM*. Google I/O conference 2010 presentation video and slides. http://code.google.com/events/io/2010/sessions/jit-compiler-androids-dalvik-vm.html

[20] CORBET, Jonathan: *Android: the return of the Unix wars?* http://lwn.net/Articles/401527/

[21] CORBET, Jonathan: *What comes after suspend blockers*. http://lwn.net/Articles/390369/

[22] GOOGLE INC. (Hrsg.): *Android Software Development Kit (SDK)*. Google Inc., http://developer.android.com/sdk/index.html. – Android 2.2, Release 2

[23] GOOGLE INC. (Hrsg.): *Android Platform Developer's Guide*. Google Inc., 2010. http://pdk.android.com

[24] HACKBORN, Dianne: *Android Developers Blog – Multitasking the Android Way*. http://android-developers.blogspot.com/2010/04/multitasking-android-way.html

[25] INC., Google: *Google Projects for Android*. http://code.google.com/android/

[26] KINCAID, Jason: *TechCrunch – Android Chief Andy Rubin: Updates Will Eventually Come Once A Year*. http://techcrunch.com/2010/06/01/android-chief-andy-rubin-updates-will-eventually-come-once-a-year/

[27] SCHMIDT, Aubrey-Derrick ; BYE, Rainer ; SCHMIDT, Hans-Gunther ; CLAUSEN, Jan H. ; KIRAZ, Osman ; YÜKSEL, Kamer A. ; ÇAMTEPE, Seyit A. ; ALBAYRAK, Sahin: Static Analysis of Executables for Collaborative Malware Detection on Android. In: *ICC*, 2009, S. 1–5

[28] SCHMIDT, Aubrey-Derrick ; SCHMIDT, Hans-Gunther ; CLAUSEN, Jan ; YÜKSEL, Kamer A. ; KIRAZ, Osman ; CAMTEPE, Ahmet ; ALBAYRAK, Sahin: Enhancing Security of Linux-based Android Devices. In: *in Proceedings of 15th International Linux Kongress*, Lehmann, October 2008

[29] SHABTAI, A. ; FLEDEL, Y. ; KANONOV, U. ; ELOVICI, Y. ; DOLEV, S.: Google Android: A State-of-the-Art Review of Security Mechanisms. In: *Arxiv preprint arXiv:0912.5101* (2009)

[30] SWETLAND, Brian: *Android Linux Kernel Development – PMEM*. http://groups.google.com/group/android-kernel/msg/6b8f9866f4d23795. – Android development mailing list

[31] SWETLAND, Brian: *Some clarification on "the Android Kernel"*. http://lwn.net/Articles/373374/

All URLs were last accessed September 2010.