Android internals: Services and Threads

TUB-mastercourse: Online Social Network Project - ST 2012

Philipp Herman
Technische Universität Berlin
Student - 336416
philipp.herman@campus.tu-berlin.de

June 22, 2012

I. BRIEF OVERVIEW OF ANDROID-ARCHITECTURE

Android is not just an operating system but also a software-platform for different mobile devices like smart phones, net-books and tablets. It is based on a linux-kernel, was created by the Open Handset Alliance (65 companies, e.g. HTC, Motorola, Qualcomm, Intel, Vodafone), under the leadership of Google and is available since Oct 2008. The current version is 4.0.4 named *Ice Cream Sandwich* released on March 28th 2012. The previous version for smart phones was version 2.3 called *Gingerbread*. Meanwhile there was Android 3 Honey-comb, a tablet optimzed version, realeased. This distinction has ended with the last version, where smart phones and tablet versions have merged. In chapter II are the main differences between the 2.3 and 4.0 are mentioned.

The core of the Android platform is a linux kernel, which is reponsible for memory and hardware management. The libraries of the Android framework are located one layer above. They are written in C/C++ and constitute the accesss to basic functionality of Android, e.g. I/O or telephony. The development of applications for Android is done in Java. It is not possible to program in C or C++. Every application is set up on the Android-framework.

II. MAIN DIFFERENCES BETWEEN ANDROID 2.3 (GINGERBREAD) AND ANDROID 4.0 (ICE CREAM SANDWICH)

Android 4.0 is the first Android version optimized for both tablets and smart phones while Android 2.3 is more suitable for smart phones. The navigation buttons such as back, home are available as soft keys on Android 4.0, where as Android 2.3 does not has soft keys for similar navigation. In devices with Android 2.3, hardware keys are available for back, home and settings (and sometimes search). The switching between applications has become more convenient in Android 4.0 because of the so called *Thumbnail Multi-Tasking*. With the help of so-called thumbnail view of recently used apps, switching between apps is now a little quicker and a lot more attractive. *Fine-Tuned Notification Control*: only Android 4.0 has the ability to dismiss individual notifications. This feature is not available in Android 2.3, where the user can only clear all notifications. The new voice input engine on Android 4.0 gives an open microphone experience and allows users to give voice commands any time, this was not implemented with Android 2.3. The *Face Unlock* feature, allowing users to unlock the home screen by face recognition is only available in Android 4.0. Both Android 2.3 and Android 4.0 support NFC (*Near Field Communication*) if the device has the capability.

III. THE PRINCIPLE OF THE SANDBOX

Android was created according to the principles of sandbo-xing. This means among other things that for each application installed, there is a new OS-user registered. This has all the rights only for his own processes and data. Android uses for this security approach (sandbox) the access concept of the fundamental Linux-OS. This also implies that each executed application is running in its own process, which is recognizable by its individual process-id (PID). Each app has its unique user (UID) and group ID (GID) and can only access its own directory. A sandbox itself is a limited runtime-environment, in which certain functionalities are prohibited; like e.g. certain accesses to the OS-layer or to data of other applications and processes. Because each process is protected no unauthorized access to the heap and memory of the individual Dalvik Virtual Machine (DVM) is ensured. The DVM is a virtual machine (VM) on Android which is optimized for systems that are constrained in terms of memory and processor speed. So prior to the installation on the Android device the Java Virtual Machine-compatible .class files need to be converted to Dalvik-compatible .dex (Dalvik Executable) files. But this is not our task, this is automatically done by the Android-SDK. So from the perspective of the Android-OS each ap-plication owns it own process, own OS-user, own Dalvik Virtual Machine (DVM), own heap-stack and its own area in the storage memory of the OS. So no other application is able to get access to other user's data and processes without authorization. Another feature of the DVM based concept is the multitasking-ability of Android. Due to the efficient and intelligent management of components and resources this is even a real multitasking-OS, not like e.g. Apples iOS. An aspect regarding its intelligent resource management is that only actually needed core libraries and frameworks are loaded by the DVM. If the application needs further information

and to go beyond the sandbox it needs authorizations. This authorizations are then explicitly requested from and assigned by the user. A problem is that the apps of an author can communicate freely because of same UID and GID. That means, if multiple apps from same author are installed on the device the different rights of each app accumulate through transitive property. An example: App A asks app B of the same author to perform an action, for which app A has no rights but app B. So app A gets in possession of rights which the user did not committed.

IV. Threads and background processing in Android

Android modifies the user interface via one thread: the UI-thread or main-thread. Furthermore, if no other threads or processes are launched during the executing of an application all of its work runs in this single thread. So if you perform a long lasting task, e.g. uploading a picture to the internet, the user interface of your application will be locked until this operation has finished. So to achieve a good user experience all slow running operations in an application should run asynchronously. Another reason is that the Android-OS enforces an ANR-dialog-message (Application Not Responding) if an activity does not react within 5 seconds to user input. The ANR-dialog-message then offers the user to choose if the application should be stopped.

V. The framework

The framework controls the behavior of applications and the communication between the applications. The framework can be devided into four main components:

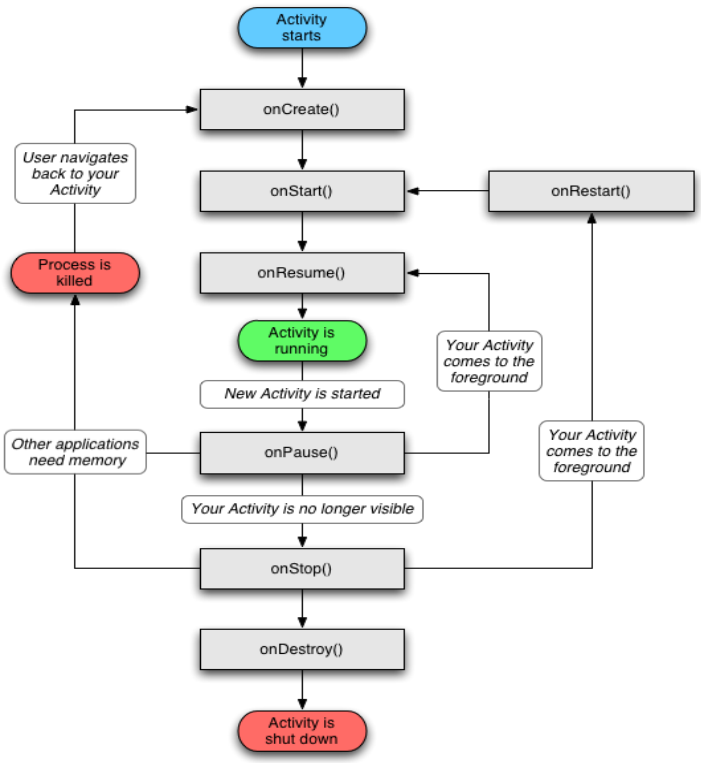
1) *activities*: An activity represents an UI, which reacts to user interaction. Activities are in a life cycle and own a lifetime which is determined by the Android-framework.
2) *services*: Services run in the background to fulfill certain tasks. Services do not own an UI and are therefore invisible for users.
3) *intents*: Intents are used to navigate between activities. They can be used for launching a certain activity and transmit some information from one activity to another.
4) *broadcast-receiver*: A broadcast-receiver listens inside a application for certain global events of the system, like e.g. incoming calls.

A. ACTIVITIES

Activities can be seen as the coreelements of an Android-application. An activity contains a layout, which again manages other views and thus constitute the basic structure of an UI. Each activity is controlled by the activity-manager of the framework which also controls its lifecycle. This is from outstanding importance, because the system is able to kill activities for system resource management. Activities are managed by following the LIFO (last in last out) principle, that means if a new activity is launched, the current activity will be put at the activity stack. If the current activity is been killed, then the previous application used is being shown. The lifecycle of an activity consist of seven states:

1) *onCreate*
2) *onStart*
3) *onRestart*
4) *onResume*
5) *onPause*
6) *onStop*
7) *onDestroy*

Every activity owns seven functions corresponding to these states, which can be overloaded by the programmer. So if the function has been overloaded and the state occurs the written code will be excuted.



B. INTENTS

The communication between activities is realized by intents. Intents can be sent between the following components: Activities, Services and Broadcast Receivers. Intents can be used to start Activities; start, stop and bind Services; and broadcast information to Broadcast Receivers. In other words: you don't need a hardcoded path to an application to use its functions and exchange data with it. Data can be passed in both directions using Intent objects.

C. SERVICES

For long duration purposes Android offers the service component. Services are working in the background of an activity without any user interface. A service also owns a lifecycle like an activity, but it is not represented visually. They have a higher processing priority than other invisible

activities. It is even possible to set its priority to the highest; that means equal to those of visible activities. Sometimes an activity may need to do a long-running operation that exists independently of the activity lifecycle itself. An example may be a camera application that allows you to upload a picture. The upload may take a long time and the user should be able to continue interacting with the application meanwhile, in this case a service is strongly recommended. Furthermore it is guaranteed that the upload is accomplished, even prioritised no matter if the origin application is paused, stopped or even finished. Intents are used to start and bind to Services. Services are differentiated into remote- and local-service.

1) Local-Service: A local-service is a service which runs in the same process like the activity which executes it. To share the already existing process is a quick and resource-saving approach to proceed the service. The drawback with this approach is that only the executing activity has access to the started service. The benefit is that the communication is much easier because it can be realized by Remote Procedure Calls (RPC). Local-services not only run in the same process like the activity but also in the same main thread (the UI-Thread).

2) Remote-Service: The remote service is a kind of Android service which runs in it own process. Due to the launch in its own separate process the stability of the entire application profits. One reason is that there is a new heap memory created for the new process. This also means that if more extensive operations are executed in that service, there is no risk to reach the end of the shared heap memory of the origin process. Furthermore a remote service can not only be used within the calling actvity but also by others. In terms of resource management a remote-service is compared to a local-service more expensive. The communication is realized by Inter Process Communication (IPC).

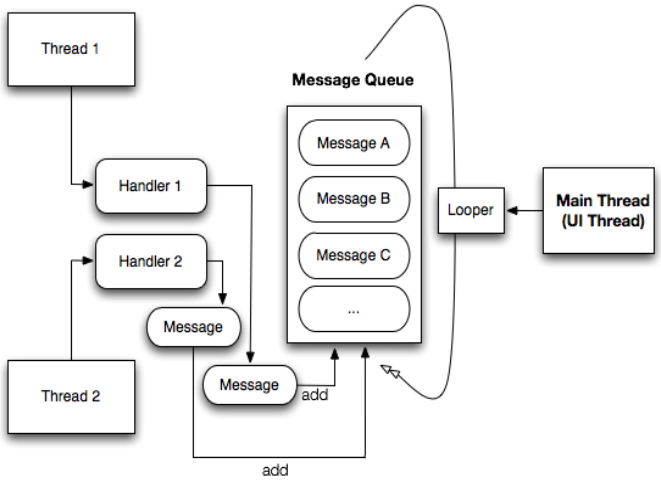
3) Android Interface Definition Language (AIDL): The Android Interface Definition Language is used for the defintion of RPC-interfaces (for IPC). IPC is used for communication from one process to another, so data stored in memory has to be moved across process boundaries. Thats were AIDL gets important. In Android it is corrupt to update UI components directly from threads other than main thread . It is a feature of AIDL that an application does not need to know if a service is running in a server process or in the local process. So if for example there are two apps where the first contains a Service and the second an Activity following is possible via AIDL: The Activity will connect to the Service using the Android RPC mechanism and will request a String from that Service. Therefor a remote interface is been transfered by using the bindService-functionality, which contains the signature of the function that will be bound. All signatures are collected in an AIDL-file. The format of the AIDL is quite similar to the java interface syntax and all parameters can be either basic java data types or *Parcelables*. So if a client wants to call methods in a remote service it can just call the *bindService* method which is available in Activities and Services.

D. LOOPER, HANDLER and MESSAGEQUEUE

As already mentioned it is not possible to update UI-components in the main thread (UI-Thread) directly from other threads started. The prime example is: we do an extensive operation and want to update a certain UI-component afterwards. In this case we do following actions: we create a Handler object associated with the main thread and post a Runnable when its needed, which is then invoked on the main thread. Therefor we make use of the already existing *Looper* and Handler classes.

The *Looper* class contains a *MessageQueue*, with a message list. Very important to know is that only one Looper can be attached to a main thread. Furthermore this connection between the Looper and the main thread is kept forever, because it can neither be broken nor changed. For a better understanding: the Looper is stored in the thread-local storage, and it can not be created via its constructor directly. To instantiate a *Looper* we use the *Prepare*-method of the *Looper* class, which then checks with the help of the *ThreadLocal*-class if there is no other Looper already associated with the main thread. In the end a new Looper is created and saved in *ThreadLocal*. We now benefit from the possibility to call the *Loop*-method of the Looper to check for new Messages and to refer them to the UI-Thread. Handlers are mainly needed for adding, removing, dispatching messages of current thread's MessageQueue. One Handler is always bounded just to one thread. The binding of the Handler to the main thread is indirectly done with the Looper and its MessageQueue. So because several Handlers can be associated to one Looper, several Handlers can be attached indirectly to one main thread. To add a message to a certain MessageQueue we call the *Post*-method of one Handler associated with that MessageQueue.

So in result we have the following concept: A queue of messages. Each message represents a job to be handled. Threads can add messages. Only a single thread pulls messages one by one from the queue to the main thread.



E. BROADCAST-RECEIVER

The broadcast receiver is for receiving system wide messages, e.g. the message that a new SMS has come in or the battery low level warning is provided to all subscribers. They can react to general system-calls or to individually defined calls by other intents. They are also able to send individual calls. Every app can specify an intent filter, a white list mechanism, that defines the types of intents the components of the application should receive. In consequence intents that are not listed will be filtered out.

F. CONTENT-PROVIDER

Content Providers are databases addressable by their application-defined URIs. They are used for persistent internal data storage and for sharing information between applications.

G. REFERENCES

(1) Android 2: Grundlagen und Programmierung, Dpunkt Verlag; Auflage: 2.,Mai 2010, Arno Becker, Marcus Pant

(2) Professional Android 2 Application Development, John Wiley and Sons; Februar 2010, Reto Meier

(3) Analyzing Inter-Application Communication in Android, a) Erika Chin b) Adrienne Porter Felt c) Kate Greenwood d) David Wagner, University of California, Berkeley

(4) Teaching Operating Systems Using Android, a) Jeremy Andrus b) Jason, Nieh, Columbia University New York