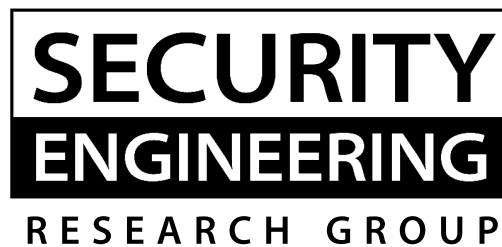


Technical Report

Analysis report on Android Application Framework and existing Security Architecture



Constrained Intents: Extending Android Security for Intent
Policies (EASIP)

Security Engineering Research Group,
Institute of Management Sciences Peshawar, Pakistan

<http://serg.imsciences.edu.pk>

Februry, 2010

Disclaimer

THIS REPORT IS PROVIDED “AS IS” WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY, NONINFRINGEMENT, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE. Without limitation, Security Engineering Research Group disclaims all liability, including liability for infringement of any proprietary rights, relating to use of information in this technical report and Security Engineering Research Group disclaims all liability for cost of procurement of substitute goods or services, lost profits, loss of use, loss of data or any incidental, consequential, direct, indirect, or special damages, whether under contract, tort, warranty or otherwise, arising in any way out of use or reliance upon this technical report or any information herein.

Any marks and brands contained herein are the property of their respective owners.

Contributors

Shahryar Khan (engrshahrs@gmail.com)

Syed Hammad Khalid Banuri (hammadbanuri@gmail.com)

Mir Nauman (mir.nauman@gmail.com)

Sohail Khan (sohail.khan@imsciences.edu.pk)

Masoom Alam (mmalam@imsciences.edu.pk)

Acknowledgements

The preparation of this technical report has been supported by Grant No. IC-TRDF/TR&D/2009/03 from the National ICT R&D Fund, Pakistan to Security Engineering Research Group, Institute of Management Sciences.

Table of Contents

Acknowledgements	iii
List of Tables	viii
List of Figures	ix
Glossary	x
1 Android Introduction	1
1.1 Android	1
1.2 Features of Android	2
1.2.1 Android Software Development Kit	2
1.2.2 Dalvik virtual machine	2
1.2.3 Graphics support	2
1.2.4 SQLite	3
1.2.5 Connectivity	3
1.2.6 Media Support	3
1.3 Android System Architecture	3
1.3.1 Linux Kernel	3
1.3.2 libraries	4
1.3.3 Android Runtime	5
1.3.4 Application Framework	5
2 Application Fundamentals	7
2.1 Application Components	7
2.1.1 Activities	7
2.1.2 Services	8
2.1.3 Broadcast Receivers	9

2.1.4	Content Providers	9
2.2	Activating Components	11
2.2.1	Activating Activity	11
2.2.2	Activating Service	12
2.2.3	Activating Broadcast Receiver	12
2.3	Shutting Down Components	12
2.4	The Manifest File	13
2.5	Intent and Intent Filters	14
3	Processes and Threads	18
3.1	Processes	18
3.2	Threads	19
3.3	Remote procedure calls	19
3.4	Thread-safe methods	21
4	COMPONENT LIFE CYCLE	23
4.1	Activity	23
4.1.1	Active	23
4.1.2	Paused	24
4.1.3	Stopped	24
4.1.4	Saving activity state	27
4.1.5	Coordination activities	27
4.2	Services	28
4.3	Broadcast Receivers	29
4.4	Life cycles and Processes	30
4.4.1	Foreground Process	31
4.4.2	Visible Process	31
4.4.3	Service Process	32
4.4.4	Background Process	32
5	Case Study	34
5.1	The code of activity	34
5.2	Code of TimeService class	37
5.3	Code of AddActivity class	38
5.4	The code of the MyBroadcastReceiver class	40
5.5	The code of the manifest file	41

5.5.1	Application Component used	41
6	Intent and Intent Filters	44
6.1	Intent objects	45
6.1.1	Component name	45
6.1.2	Action	46
6.1.3	Data	47
6.1.4	Category	47
6.1.5	Extras	48
6.1.6	Flags	48
6.2	Intent Resolution	49
6.2.1	Intent filters	50
6.2.2	Filters and security	50
6.2.3	Using intent matching	55
7	Android Security Architecture	56
7.1	Introduction	56
7.2	Application Level Security	57
7.2.1	Security implemented by Application Signing using Certificates mechanisms	57
7.2.2	Security enforcement using USER ID	57
7.2.3	Security enforced on FILE	57
7.3	Related Work	58
7.3.1	Semantically Rich Application-Centric Security in Android	58
7.3.2	Towards Formal Analysis of the Permission-based Security Model for Android	58
7.3.3	Developing Secure Mobile Applications for Android. An introduction to making secure Android applications	58
7.4	Security Flaws in Android	59
7.4.1	Vulnerability to image processing libraries	59
7.4.2	Security hole in web browser	59
7.4.3	Bugs that leads to Denial of Service attack	60
7.5	GOOGLE'S response to reported Security Threats	61
7.5.1	Patch For Android Security Flaw Released By Google And T-Mobile	61
7.5.2	Patch for Android Malformed SMS DoS	61

7.5.3	Patch for Dalvik API DoS	61
8	Android Permissions	62
8.1	Components Level Security	62
8.1.1	Security through Permission Mechanism	62
8.1.2	Permission enforcement during a Program operation	62
8.1.3	Declaration of Permissions in Android Manifest.xml file	62
8.1.4	Enforcing Permissions in Android manifest.xml	65
8.1.5	Checking Permissions against a Process	65
8.2	Data Level Security	66
8.2.1	Security through PER-URI permissions	66
8.3	Android Security Exceptions	67
9	The Android Application Life Cycle	70
9.1	Active Processes	73
9.2	Visible Processes	74
9.2.1	Started Service Processes	75
9.2.2	Background Processes	75
9.2.3	Empty Processes	75
10	Conclusion	76
	References	78

List of Tables

4.1	ENFORCING PERMISSIONS IN AndroidManifest.xml	33
6.1	String Table	46
6.2	String Table	48
8.1	ENFORCING PERMISSIONS IN AndroidManifest.xml	65

List of Figures

2.1	Shows the inter application communication	10
2.2	AndroidManifest.xml file of Media Player application	15
2.3	AndroidManifest.xml file of Media Player application with extension of intent filter	16
3.1	Figure showing RPC mechanism	20
4.1	Figure showing loops and the paths that an activity can take while switching between different states	26
4.2	Figure illustrating the callback methods for a service	30
5.1	The Figure shows the first activity is main activity which displays some text objects and a list view control	42
5.2	The Figure shows the second activity that is a simple form which is used to enter name, address, contact numbers of people	43
8.1	Install time permissions	63
8.2	Permission enforcement during a program operation	64
9.1	Priority tree used to determine the order of application termination .	74

Glossary

URI	Universal Resource Identifier
coq	Calculus of Inductive Constructions
DoS	Denial of Service
PoC	Proof of Concept
PDA	Personal Digital Assistant
OHA	Open handset Alliance
SDK	Software Development Kit
IDE	Integrated Development Environment
API	Application Program Interface
RDMS	Relational Database Management System
MSM	The Mobile Station Modem chipset and system software
SSL	Secure Sockets Layer
JNI	Java Native Interface
LED	light-emitting diode
MIME	Multipurpose Internet Mail Extensions

Abstract

Analysis report on Android Application Framework and existing Security Architecture

Security Engineering Research Group,
Institute of Management Sciences
<http://serg.imsciences.edu.pk>

Februry, 2010

This report is written as part of the research project EASIP (Extending Android Security for Intent Policies) funded by ICT R&D. It has mainly three objectives. Firstly, we present the application components and their life cycle. Secondly, we present the existing Android security mechanisms to elaborate what is going on within Android at the moment. Finally, this report will lead us to complete EASIP's next milestone, i.e. the implementation of a rudimentary 'Selective Permission Mechanism for Android'.

Android is the first comprehensive open source mobile software stack, destined towards consumer market. It consists of complete mobile operating system supported by Linux kernel, a newly built Dalvik virtual machine, and some smart mobile applications. Android system architecture is composed of applications, its framework, native *c/c++* libraries, Android runtime (which is further consist of Dalvik virtual machine and Android core libraries which reflects the functionality of core libraries written in java), and at last the Linux kernel use to manage the low level resources. The Android application architecture has four basic components; these are activities, services, broadcast receivers, and content providers. Every Android application may comprise of one or more such components. The purpose of this report is to discuss the application components and their life cycle in detail.

Moreover, the permission model and security architecture of Android are explained separately in this report. Android system implements security at process level. Variables, such as user IDs and group IDs are use to identify the applications, which in turn use to control access of that application. The components of one application could access the services provided by other application's components,

this inter component communication is controlled through permissions assigned in AndroidManifest.xml file. The URI based security permissions further refines the control access to any application's component. Some security holes were found in Android; in response Google has suggested some remedies to such security bugs. The report also incorporates a case study to elaborate the application's component life cycle.

Chapter 1

Android Introduction

1.1 Android

Mobile users demand more functionalities and innovative mobile platform in order to customize their handset according to their personal desires. The mobile community supporters, whether they are application developers, service provider, or mobile device manufacturers, all are trying hard to fulfill the growing demands of customers. Mobile service providers want to offer value-added services to their customers in an organized and meaningful manner; Application developers want the freedom to develop more powerful, affordable, and innovative handset applications; similarly mobile device manufacturers want to produce reliable and affordable mobile devices. The rise in the usage of mobile devices (cell phones, PDAs, and smart phones) and rapid growth in demands of customers have convinced the different mobile communities to work together on a common platform to compete mobile market current challenges. Open Handset Alliance is formed to provide such common platform for different mobile manufacturers and developers to contend with aforementioned challenge. The resultant product they manage to work on is the Android, an open source mobile platform. The initial development of Android was done by a firm Android Inc; then it was supported by Google. OHA distributed the Android in the market for the first time.

Android is the first comprehensive open source mobile platform, equipped with operating system, a Dalvik middleware [3], Linux kernel and with some rich modern day handset applications. The open source licensing of Android, gives freedom to developers to develop applications without concerning the royalty and licensing cost. There is no such cost of membership, testing, and digital certification fees involved

in the development of Android application. The Android SDK provides the tools and APIs necessary to begin developing applications on the Android platform using the Java programming language. Developers mostly choose the popular Eclipse Integrated Development Environments for development. The innovative Android is positioned well to confront the current challenge of mobile market place.

1.2 Features of Android

1.2.1 Android Software Development Kit

The Android SDK includes an emulator, some tools for performance profiling and debugging. Eclipse IDE is natural choice for Android developers. Android Development tool (ADT) is a plugin use to enhance and boost the performance of Eclipse IDE. It provides faster and easier way of creation and debugging of Android application. Note that further plugins are also available to support other IDEs such as IntelliJ and NetBeans for the Android developers [9].

1.2.2 Dalvik virtual machine

It is specifically designed for Android platform and optimized for mobile devices, where resource constrains is an issue (like low memory, small size, and lower processing power). Dalvik is register based virtual machine and its interpreter is optimized for faster execution [15]. Dalvik is capable of executing programs written in Java. It does not understand the java code directly, rather a dx tool is use to convert java code into byte code (which is then executed by Dalvik) [16]. The purpose of conversion java code into byte code is to optimize the code to be easily compiled over the limited resourced mobile device. Android support the execution of multiple instances of Dalvik VM simultaneously.

1.2.3 Graphics support

Android have support for both 2D and high performance 3D graphics (the OpenGL API is use to provide support for 3D graphics). The classical 3D game Doom has already been developed for Android; some other powerful 3D application such as *Google Earth* and fantastic game *Second Life* likely to be implemented for Android mobile.

1.2.4 SQLite

Android use small sized SQLite as an RDMS (approximately equals to 500 Kb) for database storage. It has got single file to store all tables, definitions and other data. SQLite simple architecture and small size made it very suitable to limited resourced mobile devices.

1.2.5 Connectivity

Android is provided with modern day communication technologies. It supports Bluetooth, WiFi, UMTS, CDMA, EDGE (Enhanced Data GSM Environment [12] and 3G.

1.2.6 Media Support

Android has got support for different picture formats, including JPEG, BMP, GIF, PNG etc. H.263 and H.264 are video coding techniques supported by Android. H.263 is specialized for video conferencing; H.264 is basically MPEG-4 standard, use to offer high video compression. MP3, WAV, AMR, AMR-WB are use to support audio compression by Android. All audio and video coding technique mention here, are supported by MPEG-3 and 3GP multimedia container.

GPS, compass, and accelerometer are some other features supported by Android. The implementation of these features varies from hardware to hardware.

1.3 Android System Architecture

1.3.1 Linux Kernel

The Android system architecture comprise of four layers. The lowest of all is Linux kernel layer, used as an abstraction between hardware and the remaining software stack of Android. The basic reason to choose Linux 2.6 as kernel, as it is an open source and has proven driver model. It makes Android a robust operating system structure. Android rely on kernel for memory management, security model, network stack and process management. The Android current architecture relies on MSM7200A Qualcomm chipset for following features [11].

- Supports WCDMA/HSUPA and EGPRS networks
- Multimedia Broadcasting Multicast Service (MBMS)
- Integrated ARM11TM applications processor and ARM9TM modem
- 4000TM and QDSP5000TM high-performance digital signal processors (DSP)
- 528 MHz ARM11 JazelleTM Java[®] hardware acceleration
- Support for BREW[®] and Java applications
- QcameraTM: Up to 6.0 megapixel digital images
- QtvTM: Playback at 30 fps VGA
- QcamcorderTM: Record at 30 fps VGA
- Q3DimensionTM: Up to 4 million triangles per second, and 133 million depth-tested, textured 3D pixels per second fill rate
- gpsOne[®] position-location assisted-GPS (A-GPS) solution
- Support for Linux[®] and other third-party operating systems
- Digital audio support for MP3, aacPlusTM and Enhanced aacPlus
- Integrated Mobile Digital Display Interface (MDDI), Bluetooth[®] 1.2 baseband processor and WiFi support

1.3.2 libraries

The native libraries of Android are written in C/C++, used by various components of Android. *Surface Manager* support Android for composing various drawing surfaces on to the mobile screen. The seamless composition of 2D and 3D graphics layers through multiple applications is managed by surface manager. *OpenGL ES 1.0* APIs are use to implement 3D libraries, then 3D graphics are created (by using such libraries) either by hardware 3D accelerator or through software base rasterization, depending upon the availability of resource. *SGL* is core for 2d graphic libraries. The *Packet Video* libraries support multi-media services, such as playback and recording for all media files, including static images. Rendering bitmap images and vector fonts supported by *FreeType* libraries. The *SSL* libraries are available in Android's native libraries stack to ensure the privacy and integrity of data transmission between

web server and your mobile device browser. *SQLite* is relational database management system use to manage databases for each application of Android. *WebKit* is an integrated open source web browser, same as in safari and Apple with subtle modification.

1.3.3 Android Runtime

Along with native libraries, the Android runtime is on second layer right above the Linux kernel. The Android runtime consist of Dalvik virtual machine and some core libraries (it inherit almost all features provided by the core libraries of Java programming language).

Dalvik VM is specifically designed for Android platform and optimized for mobile devices, where resource constrains is an issue (like low memory, small size, and lower processing power). Dalvik is register based virtual machine and its interpreter is optimized for faster execution [15]. Dalvik is capable of executing programs written in Java. It does not understand the java code directly, rather a dx tool is use to convert java code into Dalvik understandable byte code [16]. The purpose of conversion java code into byte code is to optimize the code to be easily compiled over the limited resourced mobile device. Android support the execution of multiple instances of Dalvik VM simultaneously.

1.3.4 Application Framework

Application Framework is on third layer going from bottom to top. It is basically a built-in toolkit, use to provide different set of services to Android applications. All those services which utilizes by core applications are make available for the Android developers to build innovative and rich Android applications. The application architecture is designed to simplify the reuse of components; any application can publish its capabilities and any other application may then make use of those capabilities (subject to security constraints enforced by the framework) through underlying components of application framework layer.

Brief description of some components is discussed here [8]. The *Activity Manager* manages the lifecycle of the applications and provides a common navigation back-stack of applications that are running in different processes. The *Package Manager* maintain track of all applications that are installed in the device. The *Windows Manager* manages screen of mobile device and creates surfaces (an allocated memory

block) for all running applications. The *Telephony Manager* support applications to access the information regarding telephony services. Access to some telephony information is protected and some applications may not have permission to access such protected information (Constrain to permissions declared in its manifest file). *Content Providers* supports the sharing and accessing of data among applications; suppose the messaging service is an application that can access the data of other application contacts. In Android the term resources refers to the non code external assets. The *Resource Manager* provides access to such external resources to native Android application at built time. Android supports a number of different kinds of resource files, including XML (use to store anything, other than bitmaps and Raw), Bitmap (Use to store images), and Raw files (other resources such as sound, string, etc). The development of new Android applications requires access to views of previously built application. The *View System* provides access to some extensible set of views, such as lists, text boxes, grids, embedded browser, and button. *Notification Manager* allows application to notify users about events that take place. The notification can be inform of vibration, flashing LEDs of mobile device, playing specific sound against specific event, or could be an icon (persistently displayed into the status bar).

Chapter 2

Application Fundamentals

2.1 Application Components

Android architecture is flexible enough to allow an application to make use of features that have already built by other applications (constrain to the permission allowed by other applications). For example, if you want to build an application that creates and read pdf file; and there is another application that has pdf reader capability. Then you can call upon that pdf reader to do the job for you, rather to develop your own. This reusability certainly eliminates the redundancy in development of new applications. The developer do not have to incorporate the code of other application into his application, rather it just link their application and start that specific piece of code that do the job of pdf reader, whenever the need arises. Therefore the Android system is able to start an application process, when any part of it is needed. In order to achieve such thing, Android instantiate the java objects for that specific part which is required by other application. Therefore, Android applications does not have a single entry point for everything in the application; such as `main()` functions in applications of some other systems. The Android's application architecture composed of four components that can be instantiated whenever the need arise, these are briefly described below.

2.1.1 Activities

It is a visual screen through which user interact with mobile device and its applications. An activity could be list of objects; For example Labels into top up menu, or multiple images displayed into dialog box on your mobile screen etc). An application

may consist of one or more activities depend on their architecture and design. Suppose, Text messaging is an application that composed of multiple activities, such as write message (activity for writing new message), Inbox (activity to see received messages), Outbox (activity that provide the information of sent messages), Settings (activity that provide the user some options related to set the message priorities), etc. All these activities work together and contained in a single application, however all these are independent and each one is independently created as a separate subclass of the base class activity. Usually an activity is draw into default window, which cover up the whole mobile screen, however some of the activities could also be contained into smaller window (which normally presented on to the top of another window of another activity). As an application may contain one or more activities in it, similarly an activity may consist of one or more windows.

The visual content of the window is provided by a hierarchy of views - objects derived from the base View class. Each view controls a particular rectangular space within the window. Parent views contain and organize the layout of their children. Leaf views (those at the bottom of the hierarchy) draw in the rectangles they control and respond to user actions directed at that space. Thus, views are where the activity's interaction with the user takes place. For example, a view might display a small image and initiate an action when the user taps that image. Android has a number of ready-made views that you can use - including buttons, text fields, scroll bars, menu items, check boxes, and more.

2.1.2 Services

Service refers to the application components that could not be seen, rather it could be realized. Service basically runs into the background of an activity or application. For example play back music, fetching of the data over network, calculation of an expression (whose return result will be shown through an activity, as service is nothing to do with visual contents), etc. The interaction of user with service is possible through an interface (presented by an activity) which the service posses. Once user activates any service (suppose playback music) through an activity of media player application, user may can continue other work while later activity may get close (who activated the playback music). In such case user must have to restart appropriate activity in order to stop playback music. activity may also provide some other control over service. Suppose in case of music, it may allow the user to pause, reverse, or forward the media file. However, activity does not have persistent control over the service, as user expects the service to continue when he leaves the

activity (which starts the service) and start doing something else. Once activity start playback music to run into the background, then system would be responsible to keep the playback music service continue. Each service extends the Service base class.

Like activities and the other components, services run in the main thread of the application process. So that they won't block other components or the user interface, they often spawn another thread for time-consuming tasks (like music playback). In the upcoming chapter Process and Threads are discussed in detail.

2.1.3 Broadcast Receivers

Broadcast receiver is another basic component of Android application. It is use to receive the broadcast announcements and react according to the arise situation. Normally system makes broadcast announcements, such as time zone has changed, Picture has been captured, language preferences have changed, or battery power is low. Same as system code, application can also initiate broadcast and broadcast receivers (contained by applications) may react to it, if needed so. Suppose, browser generate broadcast announcement that the requested download has been complete and it's now available for other applications to use. Application can have more than one broadcast receivers to receive multiple broadcast announcements simultaneously. All broadcast receivers extend the BroadcastReceiver base class. Unlike service component the broadcast receiver doesn't posses any sort of user interface; rather it may start a new activity in response to the message it receives or it may alert the user through Notification Manager. Notification Manager allows application to notify users about events that take place. The notification can be inform of vibration, flashing LEDs of mobile device, playing specific sound against specific event, or could be an icon (persistently displayed into the status bar).

2.1.4 Content Providers

Content providers supports the sharing and accessing of data among applications. Suppose the messaging service is an application that can access the data of other application contacts. In Android there is not any sort of centralized database storage that is available for all Android Packages. Content providers are the only way to store, retrieve, and share information among applications. Each content provider stores specific set of data, such as there are different content providers for different data types like images, audio, videos, or any user defined data type for personal

information that make sense. You can make your own data public into two ways; create a new content provider that extends the content provider base class, or use the existing content provider that is use to control the same data type you are going to publicize. Content Provider cannot be accessed directly by any application, there is another component Content Resolver that is used by application to access the information hold by content provider. All content providers basically have common interface that allows applications to interact with each other. For this, every application has to create an object of Content Resolver Class, which is then use to access that common interface provided by all content providers. Following diagram illustrate the communication process among the applications of Android (normally refers as inter process communication).

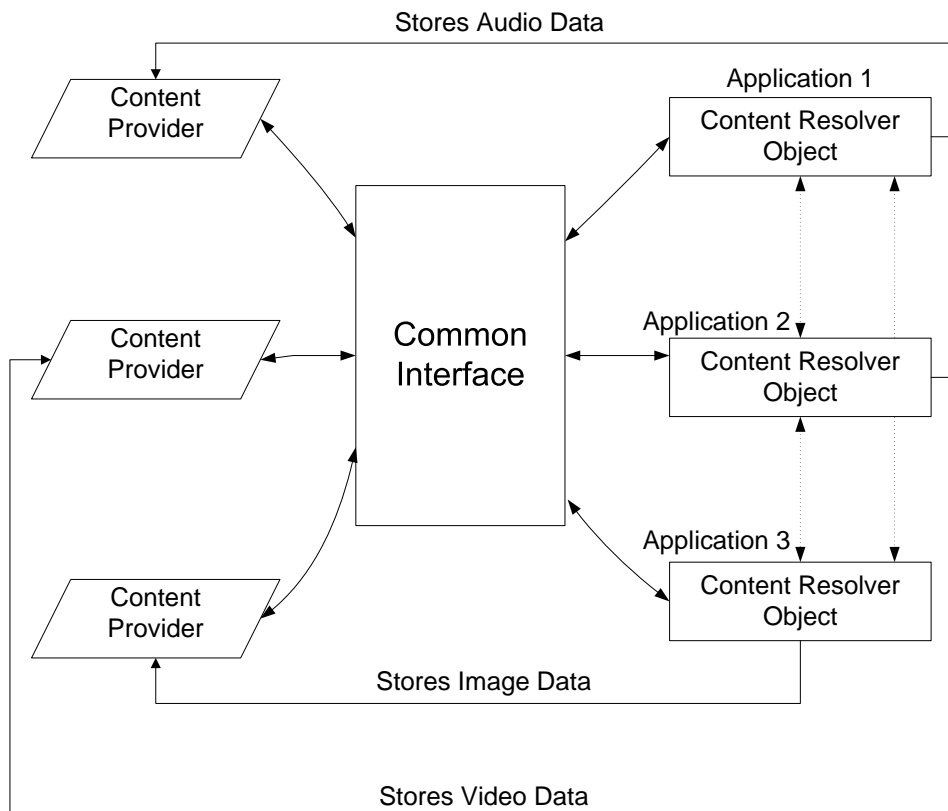


Figure 2.1: Shows the inter application communication

The above figure describes that there are three different applications; use to store different types of data (audio, images, and video respectively) contained in separate content providers. Then the sharing and retrieving of the data among applications are shown by dotted lines. These dotted lines use to describe the indirect communication process among applications (that is the communication process which takes place through content resolver object contained by each application). The

communication shown here is among different applications, which indeed communication among application's components (Therefore, it also known as inter-component communication).

2.2 Activating Components

Content Resolver object is use to activate the content provider, While Intents are use to activate other three components of Android application (Activities, Services, and Broadcast Receivers). Intents are asynchronous messages; it holds the name of action being requested by an application and specifies the URI of the data to act upon, including some other information for Android system (such as component's category that is going to handle the intent and directions on how to launch a target component).

Suppose, it might ask for an activity to allow the user to edit some text, or present an image to the user. Similarly in case of broadcast receiver, it might prompt broadcast receiver to alert the user that selected download has been completed. Different methods are use to activate each type of components.

2.2.1 Activating Activity

An intent object is passed to either `Context.startActivity ()` or `Activity.startActivityForResult ()` in order to launch an activity. Sometimes, when an activity launches another activity, it expects some result in return. So when result is expected by launching new activity, then the intent object is passed to `Activity.startActivityForResult ()`, otherwise the later object is passed to `Context.startActivity ()`. For example, Text messaging activity, launch another activity Phone book and which let the user to select the contact from, then initial activity (Text Messaging) will expects that launched activity (Phone Book) should return the selected contacts. In such a scenario the intent object will be passed to `Activity.startActivityForResult ()` instead `Context.startActivity ()`. The called activity looks at the initial intent that caused it to be launched by calling its `getIntent ()` method and in response the result is returned contained by intent object is passed to `onActivityResult ()` method of calling activity.

2.2.2 Activating Service

A new service is started (or further instructions to an ongoing service is given) by passing an intent object to `Context.startService ()`. Whenever the user explicitly calls the service, Android system calls the service's `onStart ()` method for backward compatibility and passes it the intent object. The `Context.bindService ()` is used to establish a connection between calling component and with an ongoing target service (`bindService ()` method is also capable of starting service, if it's not already running). For example the playback music service is already activated by user and then he switches to new application, now in order to re establish the connection with that ongoing playback music (so that the interface with controls over that service could be accessed), the `bindService ()` method is called. The called service looks at the initial intent that caused it to be launched by calling its `onBind ()` method.

2.2.3 Activating Broadcast Receiver

There are three different methods (`sendBroadcast ()`, `sendOrderedBroadcast ()`, and `sendStickyBroadcast ()`) to activate broadcast receiver. It depends on the type of broadcast that which method out of these three will be called. The intent object is passed to `sendBroadcast ()` method, when you want to restrict the receiver's response and does not allow him to propagate any result and abort the broadcast. Otherwise, if you wish propagation of any result and allow the receiver to abort broadcast, the intent object is passed to `sendOrderedBroadcast ()`. Android delivers the intent to all interested broadcast receivers by calling their `onReceive ()` methods.

2.3 Shutting Down Components

The content provider and broadcast receivers are those components, which do not need to be shut down explicitly. Content Resolver object of an application activates the content provider and after responding that request the components shut down implicitly. Similarly, broadcast receiver is active for a period, in which it responds to broadcast message and then implicitly shutdown. On the other hand both Activities and Services need to be shutdown explicitly. Activities use to provide an interface, which normally interacts with the user for prolonged

time period and remain active, until and unless the Android system orderly shut down it. The `finish ()` method is use to shutdown any activity (that was started through `Context.startActivity ()` method). The activity which is initiated by method `startActivityForResult ()`, needs to be close through `finishActivity ()` method. Similarly, the services are shut down through methods `stopSelf ()` or `Context.stopService ()`, depends on which method use to initiate the service (`Context.startService ()` or `Context.bindService ()` method).

2.4 The Manifest File

Every application of Android has its own manifest file, which is structured in XML format. It is always named as `AndroidManifest.xml` for all applications. The manifest file declares all corresponding components of an application. The existence of any component of an application must be known to Android system, so that the system could be able to initiate the application component, whenever it is required to do so. The manifest file of each application is bundled into Android Package (i.e .apk file), which also contain resources, application's code, and some other files. We know that the main task of the manifest file is to present application's components to the Android system. However, In addition to declaring application components, manifest also contain some other information, which is following [1].

- It names the Java package for the application. The package name serves as a unique identifier for the application.
- It names the classes that implement each of the components and publishes their capabilities (for example, which Intent messages they can handle). These declarations let the Android system know what the components are and under what conditions they can be launched.
- It determines which processes will host application components.
- It declares which permissions the application must have in order to access protected parts of the API and interact with other applications.
- It also declares the permissions that others are required to have in order to interact with the application's components.
- It lists the Instrumentation classes that provide profiling and other information as the application is running. These declarations are present in the manifest only while the application is being developed and tested; they're removed before the application is published.

- It declares the minimum level of the Android API that the application requires.
- It lists the libraries that the application must be linked against.

To give an overview that how that `AndroidManifest.xml` file will look like, we have created an example of Media Player application¹.

The name attribute of `<activity>` tag shows the name of the Activity subclass, which implements the activity. The icon and label attributes point to resource files containing an icon and label that can be displayed to users to represent the activity. The element `<Intent-filter>`, placed under the element `<Activity>`. It shows the Intent that component activity can handle. The upcoming section describes the intents and intent filters in detail.

Similarly, the broadcast receivers, services, and content providers are defined through `<receiver>`, `<service>` and `<provider>` elements. Each of the Android application's components (Activities, Services, and Content Providers) have to be declared into corresponding manifest file; those components which are not declared would not be visible to Android system and hence would never be able to execute. However, only broadcast receivers can either be defined into `AndroidManifest` file, or they can be formed dynamically in code (as `BroadcastReceiver` objects) and registered with the system by calling `Context.registerReceiver()`.

Note that in the above example of `AndroidManifest.xml` file, the element `<intent-filter>` is just shown as subpart of element `<Activity>`; however the services and broadcast receivers could also have one or more `<intent-filter>` elements.

2.5 Intent and Intent Filters

Intents are basically intents objects that are asynchronous messages; it holds the name of action being requested by an application's component and specifies the URI of the data to act upon, including some other information for Android system (such as component's category that is going to handle the intent and instructions on how to launch a target component). The intent raised by an application's component are of two categories; (1) Intent that explicitly mentions the name of requested component known as explicit Intent, (2) whereas the Intent that does not mention the name of requested component known as implicit intent. An application's component

¹The example shown here is not complete `AndroidManifest.xml` file for Media Player application. It's just included to give a quick overview to the readers.

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3     package="com.media"
4     android:versionCode="1"
5     android:versionName="1.0">
6     <application android:icon="@drawable/icon"android:label="@string/app_name">
7         <activity android:name=".MediaActivity"
8             android:label="@string/app_name"
9             android:icon="@drawable/small_pic.png">
10            <intent-filter>
11                <action android:name="android.intent.action.MAIN" />
12                <category android:name="android.intent.category.LAUNCHER"/>
13            </intent-filter>
14        </activity>
15        <receiver
16            android:enabled="true"
17            android:label="My Media Receiver"
18            android:name=".MediaReceiver">
19        </receiver>
20        <service
21            android:enabled="true"
22            android:name=".TimeService">
23        </service>
24        <provider
25            android:permission="com.media.MY_PERMISSION"
26            android:name=".VideoProvider"
27            android:enabled="true">
28        </provider>
29        <provider
30            android:permission="com.media.MY_PERMISSION"
31            android:name=".AudioProvider"
32            android:enabled="true">
33        </provider>
34    </application>
35    <uses-sdk android:minSdkVersion="3" />
36 </manifest>
```

Figure 2.2: AndroidManifest.xml file of Media Player application

names are normally not known to the developer of other applications (due to security concern). Therefore implicit intents are raised to activate the components of other applications and explicit intents are initiated in order to activate the components within the same application. In case of explicit intents, Android locates the requested component based on its declaration into the manifest file and executes it. In case of implicit intent, Android must locate the component that has best possible match to

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <manifest . . . . >
3   <application . . . . >
4     <activity android:name=".MediaActivity"
5               android:label="@string/app_name"
6               android:icon="@drawable/small_pic.png">
7       <intent-filter>
8         <action android:name="android.intent.action.MAIN" />
9         <category android:name="android.intent.category.LAUNCHER" />
10      </intent-filter>
11      <intent-filter>
12        <action android:name="android.intent.action.READ" />
13        <data android:mimeType="Video / avi" />
14        <category android:name="android.intent.category.DEFAULT" />
15      </intent-filter>
16    </activity>
17    . . . .
18
19  </application>
20  <uses-sdk android:minSdkVersion="3" />
21 </manifest>
```

Figure 2.3: AndroidManifest.xml file of Media Player application with extension of intent filter

the raised intent and execute such component. The best possible match is identified through the comparison between Intent object and Intent filter of the potential target components.

The Intent filters declared into the manifest file notify the system that which implicit intents they can manage. These filters publicize the capability of the application's component and describe the set of implicit intents, they are ready to handle. Besides that they also filter out the unwanted implicit intents.

Application's components activities, services, and broadcast receivers consist of one or more intent filter elements. On the other hand, an explicit intent is always delivered to its target, no matter what it contains; the filter is not consulted. But an implicit intent is delivered to a component only if it can pass through one of the component's filters. The following example is an extension of previous media player AndroidManifest file; here we added a new intent filter to the activity.

Here's an extension of the previous example that adds two intent filters to the activity:

The first filter (the combination of the action "android.intent.action.MAIN" and the category "android.intent.category.LAUNCHER") in the above example is common. This filter is used to spot the activity that should be represented into application launcher. The activity presented in the launcher, is the one that is the entry point for the application. The second filter presents an action that the activity can perform on a particular type of data. In this case, it demonstrates that activity can read video files, those who have avi extension.

A component can have any number of intent filters, everyone one describing a different set of capabilities. If a component doesn't have any filters, it can only be activated through explicit intents. For a broadcast receiver that's created and registered in code, the intent filter is instantiated directly as an IntentFilter object. All other filters are set up in the manifest.

Chapter 3

Processes and Threads

When ever an application component want to get executed a Linux process is started for it containing only one thread. All the components of that application run in the same process and thread by default, how ever we can make components run in different processes and threads.

3.1 Processes

The manifest file controls the processes where the components of an application will be run. The elements of a component are `<activity>`, `<service>`, `<receivers>` and `<providers>`. All of them contains an attribute called `process` which determines where a component will be run. These attributes can be configured in such a way that every component can run in its own process or some of the components can share the same process while some will not. These attributes can also be configured in such a way that components from different application can be hosted by the same process keeping in mind that those applications will share same Linux user ID and same authorities have signed the applications. The element `<application>` has also got a `process` attribute that is used to set the default value for all the components. The specific process in the main thread instantiates all the components of an application and calls to those components are initiated from that thread. Separate threads are not created for each instance. `.calls` to methods that require some user interactions or contains long operations or complex computational loops should be included in separate threads, this will be discussed in detail in the threads section of this document.

A process can get killed at any time if the system is running low on memory and some other processes is in dire need of it and they are currently in the use of the user. If there is some work to be done by the killed process, the process is started again.

The decision of which process to kill and which to keep alive is done by the system on the relative importance of the process to the user. A process associated with an activity that is not in the foreground and is not visible is more likely to be killed than the process which is associated with an activity that is active, in foreground and the user is currently interacting with it. Therefore the decision of killing a process depends greatly on the state of the process.

3.2 Threads

Normally application runs in the same process but If we have a process that is meant to do some background work, we can not make the user interface un-responsive or hanged etc like doing some heavy download from the internet or listening to live music in the background. If we keep both the tasks in the same thread, the heavy background work and the user interface with which the user is currently interacting with. By doing this it will make the user-interface slow responsive and may be hanged for some time. Therefore a good convention is that the thread the contains the user interface than it should not host time consuming operations. Every thing that takes a long time to complete should be run in a different thread. The java Thread objects are used to create threads. Looper, Handlers and HandlerThread are classess used for running a message in a loop, processing messages and configuring message loop in a thread respectably.

3.3 Remote procedure calls

Android has a lightweight mechanism for remote procedure calls (RPCs), where a method is called locally, but executed remotely (in another process), with any result returned back to the caller. This entails decomposing the method call and all its attendant data to a level the operating system can understand, transmitting it from the local process and address space to the remote process and address space, and reassembling and reenacting the call there. Return values have to be transmitted in the opposite direction. Android provides all the code to do that work, so that you can concentrate on defining and implementing the RPC interface itself.

An RPC interface can include only methods. All methods are executed synchronously (the local method blocks until the remote method finishes), even if there is no return value.

In brief, the mechanism works as follows: You'd begin by declaring the RPC interface you want to implement using a simple IDL (interface definition language). From that declaration, the `aidl` tool generates a Java interface definition that must be made available to both the local and the remote process. It contains two inner class, as shown in the following diagram:

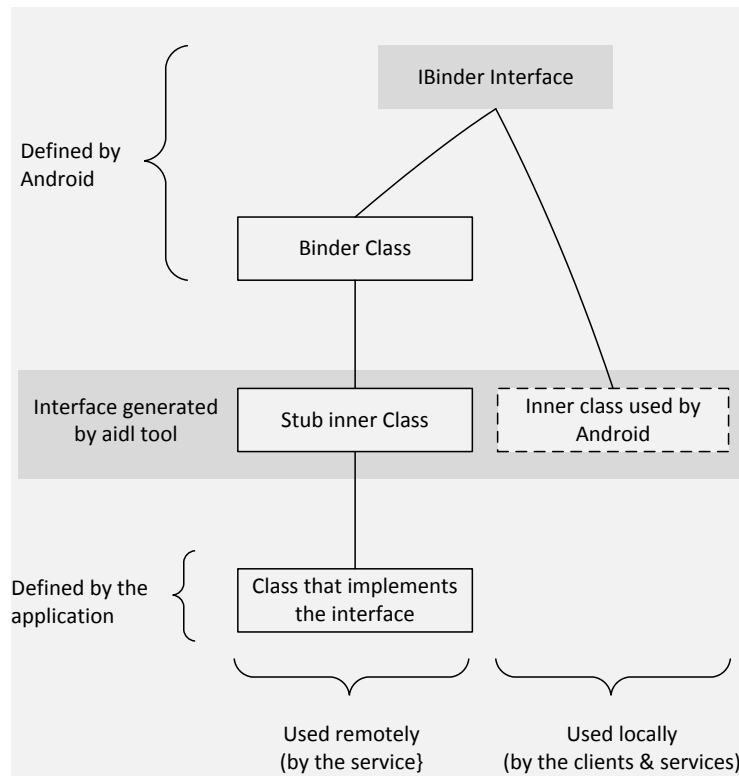


Figure 3.1: Figure showing RPC mechanism

The inner classes have all the code needed to administer remote procedure calls for the interface you declared with the IDL. Both inner classes implement `IBinder` interface. One of them is used locally and internally by the system; the code you write can ignore it. The other, called `Stub`, extends the `Binder` class. In addition to internal code for effectuating the IPC calls, it contains declarations for the methods in the RPC interface you declared. You would subclass `Stub` to implement those methods, as indicated in the diagram.

Typically, the remote process would be managed by a service (because a service

can inform the system about the process and its connections to other processes). It would have both the interface file generated by the `aidl` tool and the `Stub` subclass implementing the RPC methods. Clients of the service would have only the interface file generated by the `aidl` tool.

Here's how a connection between a service and its clients is set up:

Clients of the service (on the local side) would implement `onServiceConnected()` and `onServiceDisconnected()` methods so they can be notified when a successful connection to the remote service is established, and when it goes away. They would then call `bindService()` to set up the connection.

The service's `onBind()` method would be implemented to either accept or reject the connection, depending on the intent it receives (the intent passed to `bindService()`). If the connection is accepted, it returns an instance of the `Stub` subclass.

If the service accepts the connection, Android calls the client's `onServiceConnected()` method and passes it an `IBinder` object, a proxy for the `Stub` subclass managed by the service. Through the proxy, the client can make calls on the remote service.

This brief description omits some details of the RPC mechanism. For more information, see [Designing a Remote Interface Using AIDL](#) and the `IBinder` class description.

3.4 Thread-safe methods

In a few contexts, the methods you implement may be called from more than one thread, and therefore must be written to be thread-safe.

This is primarily true for methods that can be called remotely as in the RPC mechanism discussed in the previous section. When a call on a method implemented in an `IBinder` object originates in the same process as the `IBinder`, the method is executed in the caller's thread. However, when the call originates in another process, the method is executed in a thread chosen from a pool of threads that Android maintains in the same process as the `IBinder`; it's not executed in the main thread of the process. For example, whereas a service's `onBind()` method would be called from the main thread of the service's process, methods implemented in the object that `onBind()` returns (for example, a `Stub` subclass that implements RPC methods) would be called from threads in the pool. Since services can have more than one client, more than one pool thread can engage the same `IBinder` method at the same

time. `IBinder` methods must, therefore, be implemented to be thread-safe.

Similarly, a content provider can receive data requests that originate in other processes. Although the `ContentResolver` and `ContentProvider` classes hide the details of how the interprocess communication is managed, `ContentProvider` methods that respond to those requests—the methods `query()`, `insert()`, `delete()`, `update()` and `getType()`—are called from a pool of threads in the content provider's process, not the main thread of the process. Since these methods may be called from any number of threads at the same time, they too must be implemented to be thread-safe.

Chapter 4

COMPONENT LIFE CYCLE

Life cycle encapsulates the life of a component from the beginning when it is instantiated , to the destruction or end. During the initiation and destruction a component may be active, inactive, visible or invisible. In this chapter we will discuss the life cycle of the following

1.Activities

2.Services

3.Broadcast Receivers

We will also have a look at the different states in which a component can exist and the methods that notify us of the states of transition and the effect of those states on the host processes.

4.1 Activity

Activities has three important and essential states.

4.1.1 Active

When an activity is on top of all other activities (top of the activity stack), it is in active or running state and it is the place where a user can directly interact with.

4.1.2 Paused

A paused activity is when another activity comes on top, the activity at the top is either transparent or doesn't cover the whole screen so the paused activity can be seen through it or from its sides. A paused activity maintains all its states and keeps its member information and stays attached to the window manager. The paused activity can be destroyed if the system is running low on memory.

4.1.3 Stopped

In stopped state the activity is not visible to the users and user can not interact with it directly but still it maintains all its states. A stopped activity can get destroyed if the system needs memory else where. The system can remove a stopped activity by calling the `finish()` method or simply destroying its process.

When an activity transforms from one state to another state, the changes are notified by calls to the following protected methods.

a. `void onCreate(Bundle savedInstanceState)`

b. `void onStart()`

c. `void onRestart()`

d. `void onResume()`

e. `void onPause()`

f. `void onStop()`

g. `void onDestroy()`

The `onCreate()` method must be implemented for the initiation of the objects. The above methods can be override for the actions that we want, when the state changes. To commit the changes in data and to stop user interactions `onPause()` method can also be implemented.

To implement the above methods for an activity first its superclass version is called. e.g,

```
protected void onStart()
```

```
super.onStart();
```

The whole life cycle of an activity is defined by the above mentioned methods. The lifetime of an activity can be differentiated in three nested loops which can be monitored by implementing them.

1. Entire lifetime

Anything that happens to an activity from `onCreate()` method to `onDestroy()` method. During this lifetime the activity performs its initial setup of 'global' state in `onCreate()` method and release all resources by calling `onDestroy()` method e.g, if a thread wants to download data from Internet or any network and the thread is running in the background, in `onCreate()` method the thread will be created and in `onDestroy()` method the thread will be stopped.

2. Visible lifetime

The time of an activity from `onStart()` method to its concerned `onStop()` method. In this period the activity can be seen on the screen although it may not be in the foreground and user won't be able to interact directly. The `onStart()` and `onStop()` methods can be called multiple times depending on the needs and the activity switching between being visible to user and invisible to user.

3. Foreground lifetime

The time of an activity between `onResume()` method and `onPause()` method is the overall foreground lifetime of that activity. In between this time the activity is in front and user can directly interact with it, activities can frequently switch between resume and paused states e.g, if a new activity is started or the current activity goes to sleep, the activity goes to pause state. If a new intent is passed to an activity or result is returned, its `onResume()` method is called, the switching between `onPause()` method and `onResume()` can be of high frequency so the code in these two methods should be light enough to keep the over all application running smooth.

The diagram shows loops and the paths that an activity can take while switching between different states. The major states of an activity are represented by colored ovals and callback methods are represented by rectangles

All the above methods are explained in detail in the following table and shows its location in the life cycle of the activity.

The column Killable in the table shows that whether a process can be killed or

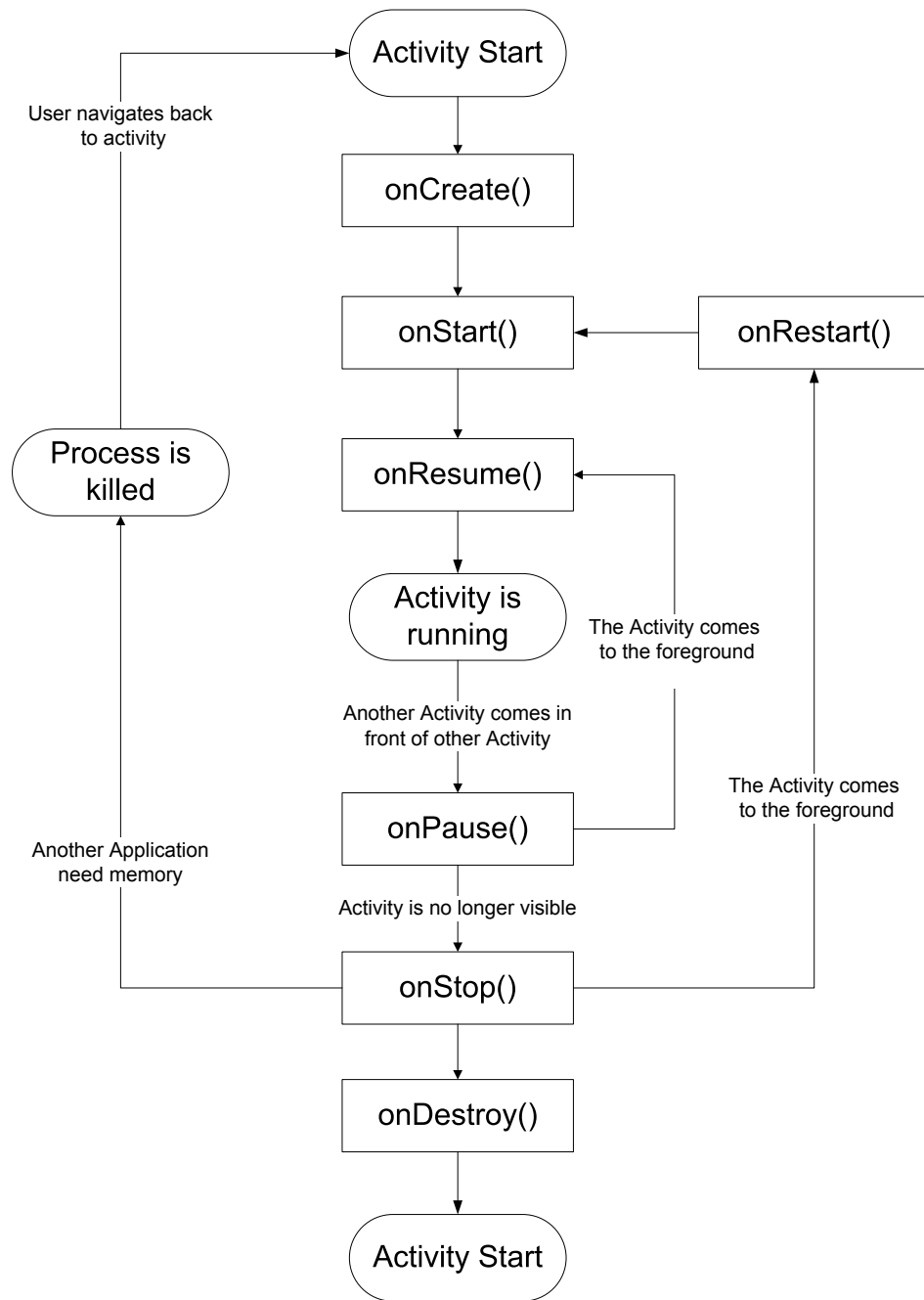


Figure 4.1: Figure showing loops and the paths that an activity can take while switching between different states

not by the system. In the above table three methods have Yes in killable column. `onPause()` is called before a process is killed so if we want to save or write certain things before the process is killed, we should write it in the `onPause()` method. Other two methods that can be killed by the system are `onStop()` and `onDestroy()`.

The system can not kill a process that has a method that is marked No in the killable column. But in extreme cases and in dire need of resources the system can kill these processes too.

4.1.4 Saving activity state

Some time the system kills an activity to free some memory and the user starts the activity again, the user finds the activity in its previous state. Before an activity gets killed its state can be saved by using `onSaveInstanceState()` method. This method is called before `onPause()` method and a `Bundle` object is passed to it, in which the state of the activity is saved as name-value pairs. When the activity is started again the `Bundle` object is passed to `onCreate()` method and to `onRestoreInstanceState()` method that is called after `onStart()` method. So that any of the two or both of them can restore the saved activity state.

`onSaveInstanceState()` method and `onRestoreInstanceState()` method are not part of the life cycle of an activity because they are not called in every case. It is only called when the system wants to kill an activity but if the user wants to kill the activity through some action like cancel or back or close and the user is not expected to come back to the activity than there is no need of saving the state of that activity.

4.1.5 Coordination activities

Some time one activity can call another activity and both pass through different stages of their life cycle during the transitions. One activity pauses and may stops and the other activity starts. In some cases we may need to do some coordination between these two activities, if the activities are part of the same process than the order of the callback life cycle would be

- 1.`onPause()` method is called for the current activity.
- 2.`onCreate()`,`onStart()` and `onResume` methods are called for the activity that starts.
- 3.`onStop()` method is called if the starting activity is not visible on the screen.

4.2 Services

Services run in the background and normally can be initiated in the following two ways.

Can be initiated and left running until someone stops it or the service stops itself. In this mode a service is initiated through a call to `Context.startService()` method and stopped by calling the `Context.stopServiceI()` method. A service can stop itself through a call to `Service.stopSelf()` or `Service.stopSelfResult()` methods. No matter how many `startService()` were called only one call to `stopService()` is needed to stop the service.

Services defines some interfaces and can export them so services can be programmed to operate through these interfaces. Clients call the service by making a connection with the Service object and than calling the service through it. `Context.bindService()` method is used make a connection and `Context.unbindService()` method is used to close a connection. More than one clients can bind to the same service. `bindService()` method can optionally launch a service if it has not be already launched.

The above mentioned modes are not much separate from each other .

```
Void onCreate()
```

```
void onStart(Intent intent)
```

```
void onDestroy()
```

By implementing these methods, you can monitor two nested loops of the service's lifecycle:

The entire lifetime of a service happens between the time `onCreate()` is called and the time `onDestroy()` returns. Like an activity, a service does its initial setup in `onCreate()`, and releases all remaining resources in `onDestroy()`. For example, a music playback service could create the thread where the music will be played in `onCreate()`, and then stop the thread in `onDestroy()`.

The active lifetime of a service begins with a call to `onStart()`. This method is handed the Intent object that was passed to `StateService()`. The music service would open the Intent to discover which music to play, and begin the playback.

There's no equivalent callback for when the service stops, no `onStop()` method.

The `onCreate()` and `onDestroy()` methods are called for all services, whether they're started by `Context.startService()` or `Context.bindService()`. However, `onStart()` is called only for services started by `startService()`.

If a service permits others to bind to it, there are additional callback methods for it to implement:

```
IBinder onBind(Intent intent)

boolean onUnbind(Intent intent)

void onRebind(Intent intent)
```

The `onBind()` callback is passed the `Intent` object that was passed to `bindService` and `onUnbind()` is handed the `intent` that was passed to `unbindService()`. If the service permits the binding, `onBind()` returns the communications channel that clients use to interact with the service. The `onUnbind()` method can ask for `onRebind()` to be called if a new client connects to the service.

The following diagram illustrates the callback methods for a service. Although, it separates services that are created via `startService` from those created by `bindService()`, keep in mind that any service, no matter how it's started, can potentially allow clients to bind to it, so any service may receive `onBind()` and `onUnbind()` calls.

4.3 Broadcast Receivers

Broadcast receivers are components that are listening for messages and has a one call back method i.e `void onReceive(Context curContext, Intent broadcastMsg)`.

The `onReceive()` method is called when a broadcast message is arrived. When the `onReceive()` method is executing only at that time a broadcast receiver is active. the broadcast receiver becomes inactive when the `onReceive()` method returns.

An active broadcast receivers process is protected from getting killed but inactive broadcast receiver's process can get killed anytime by the system when memory it has reserved is required by other processes. Now this character of broadcast receivers gives birth to problem. E.g. the response to a message is taking a long time due to some reasons, the process become inactive for the time being and becomes in the condition of getting killed. This problem can be solved for the `onReceive()` method to start a service and do the time consuming job so that the system knows that the

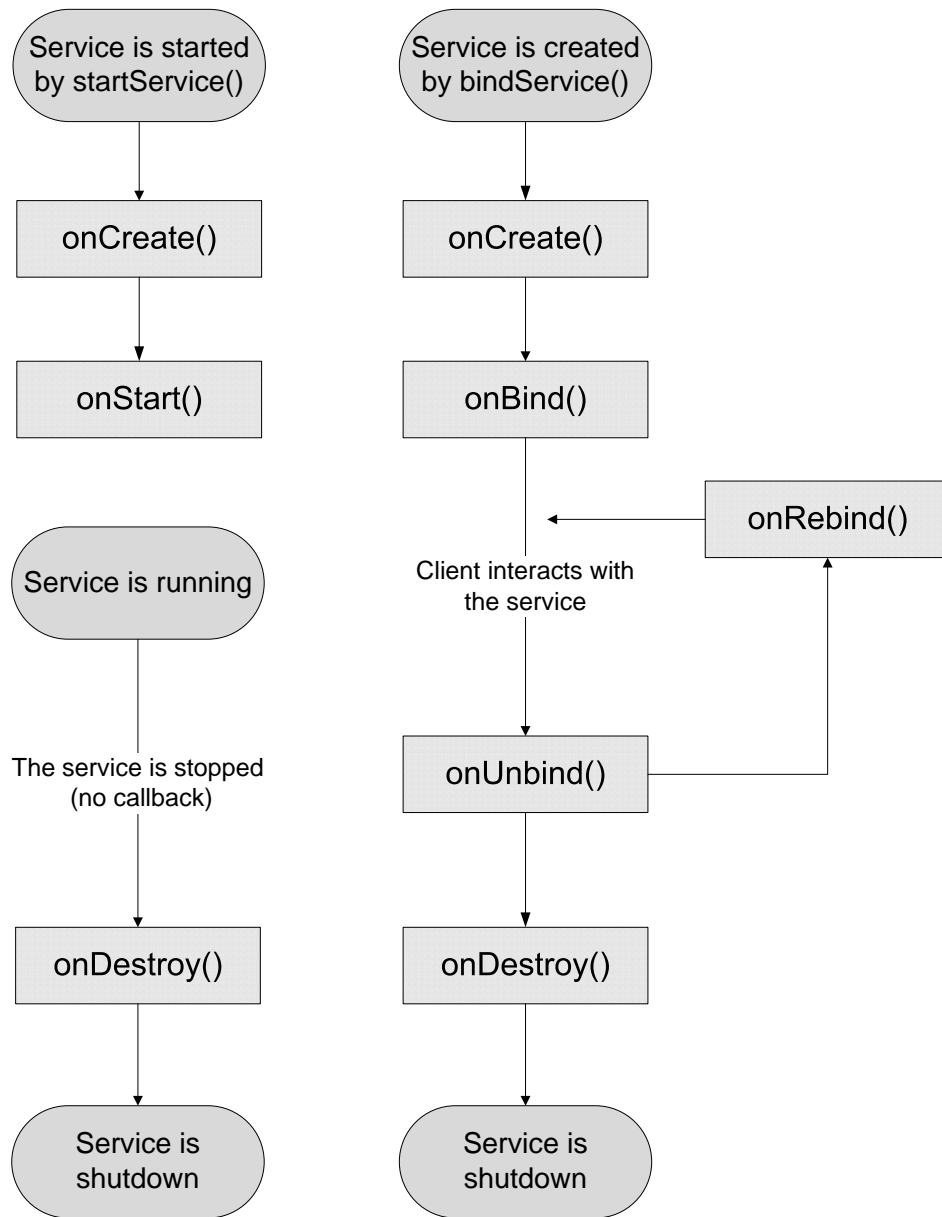


Figure 4.2: Figure illustrating the callback methods for a service

process is not inactive and some work is in progress

4.4 Life cycles and Processes

Processes of an application are tried to kept alive by the system for as long as possible but there is a point when the system has to kill some processes to free memory

specially if the system is running low on memory, and use the freed for some other processes. The system has a mechanism for deciding which process to kill and which to keep alive by placing an 'importance hierarchy'. This 'importance hierarchy' depends on the components that are running and their states. The processes that have the lowest importance are killed first than those that have the next lowest and so on. There are five levels in the 'importance hierarchy'. They are

- 1.Foreground process
- 2.Visible process
- 3.Service process
- 4.Background process
- 5.Empty Process

4.4.1 Foreground Process

Foreground process is that process that is required for what the user is currently doing on the system. A foreground process has the following conditions

The process is running an activity with which the user is currently interacting with. The activity is active activity and its `onResume()` method is called. The process can be hosting a service that is bound to an active activity with which the user is currently interacting. The process has a Service object that is sending or executing callback to `onCreate()`, `onStart()` or `onDestroy()` methods. The process contains an active broadcast receiver object means that the `onReceive()` method of the broadcast receiver is running. Normally a few foreground processes will be in running state at a given instance. These types of processes are killed when there is no other choice and the system is running very low on memory. When the system reaches this stage the device has reached a memory paging state and to keep the user interface active and responsive it becomes necessary to kill some foreground processes.

4.4.2 Visible Process

Visible process is the process that does not have any foreground running components but it can still effect whats going on the screen of the user. A visible process has the following conditions.

The process is running an activity that is not active and is not in the foreground but the user can still see it. Its in the pause state and its `onPause()` method has been initiated. The process can be hosting an activity that is visible. The visible processes are considered very important and will not be killed until and unless memory is required for the foreground processes to keep running.

4.4.3 Service Process

Service process is the process that has an active service running, started with `startService()` method and it can not fit into the above mentioned types. Service processes are not directly bound to what the user is doing or seeing on the screen but they do something that the user is interested in (playing mp3 or downloading some data in the background and typing sms in the foreground). So the system keeps them alive until and unless there is a dire need of memory for the foreground processes.

4.4.4 Background Process

Background process is a process that is holding an activity that is not active or paused or visible. `onStop()` method for that activity has been called. These processes can get killed at anytime because they have no direct affect on the user interactions and activities. Normally there are many background processes in running state so they are kept in LRU(least recently used) list to make sure that the process that the user has used most recently, will be killed in the last.

Empty process is the process that does not hold any active component of the application. These type of processes are used as a cache to improve startup time when the process wants to run the next time. Normally the system kills these processes to keep balance between processes caches and kernel caches.

Process importance and ranking can be increased depending on the other processes that it is serving.

Table 4.1: ENFORCING PERMISSIONS IN AndroidManifest.xml

Method	Description	Killable?	Next
<code>onCreate()</code>	Called when the activity is first created. This is where you should do all of your normal static set up create views, bind data to lists, and so on. This method is passed a <code>Bundle</code> object containing the activity's previous state, if that state was captured (see Saving Activity State, later). Always followed by <code>onStart()</code> .	NO.	<code>onStart()</code>
<code>onRestart()</code>	Called after the activity has been stopped, just prior to it being started again. Always followed by <code>onStart()</code> .	NO.	<code>onStart()</code>
<code>onStart()</code>	Called just before the activity becomes visible to the user. Followed by <code>onResume()</code> if the activity comes to the foreground, or <code>onStop()</code> if it becomes hidden.	NO.	<code>onResume()</code> or <code>onStop()</code>
<code>onResume()</code>	Called just before the activity starts interacting with the user. At this point the activity is at the top of the activity stack, with user input going to it. Always followed by <code>onPause()</code> .	NO.	<code>onPause()</code>
<code>onPause()</code>	Called when the system is about to start resuming another activity. This method is typically used to commit unsaved changes to persistent data, stop animations and other things that may be consuming CPU, and so on. It should do whatever it does very quickly, because the next activity will not be resumed until it returns. Followed either by <code>onResume()</code> if the activity returns back to the front, or by <code>onStop()</code> if it becomes invisible.	YES.	<code>onResume()</code> or <code>onStop()</code>
<code>onStop()</code>	Called when the activity is no longer visible to the user. This may happen because it is being destroyed, or because another activity (either an existing one or a new one) has been resumed and is covering it. Followed either by <code>onRestart()</code> if the activity is coming back to interact with the user, or by <code>onDestroy()</code> if this activity is going away.	YES.	<code>onRestart()</code> or <code>onDestroy()</code>
<code>onDestroy()</code>	Called before the activity is destroyed. This is the final call that the activity will receive. It could be called either because the activity is finishing (someone called <code>finish()</code> on it), or because the system is temporarily destroying this instance of the activity to save space. You can distinguish between these two scenarios with the <code>isFinishing()</code> method.	YES.	nothing

Chapter 5

Case Study

In this case study we have created a very simple application that contains two activities. The first activity is the main activity which displays some text objects and a list view control.

The second activity is a simple form which is used to enter name, address, contact numbers of people.

Now we will discuss this case study in detail with the help of these two figure and some snippets from the actual code of this application.

5.1 The code of activity

```
1 import java.util.ArrayList;
2 import android.app.Activity;
3 import android.content.Intent;
4 import android.os.Bundle;
5 import android.util.Log;
6 import android.view.Menu;
7 import android.view.MenuItem;
8 import android.widget.AdapterView;
9 import android.widget.ListView;
10 import android.widget.TextView;
11 public class MainActivity extends Activity implements Runnable
12     ListView myListView;
13     public static TextView nameView;
14     ArrayList <Person> myArrayList;
15     ArrayAdapter<Person> myAdapter;
16     Person person;
```

```
17         Intent in;
18         private static final int MENU_INSERT = 1;
19         private static final int ADD_ACTIVIY = 1;
20         @Override
21     public void onCreate(Bundle savedInstanceState)
22     {
23         super.onCreate(savedInstanceState);
24         setContentView(R.layout.main);
25         timeView = (TextView) findViewById(R.id.timeText);
26         nameView = (TextView) findViewById(R.id.nameText);
27         myListView = (ListView) findViewById(R.id.myList);
28         myArrayList = new ArrayList<Person>();
29         myAdapter = new Array Adapter<Person>(this,
30         android.R.layout.simple_expandable_list_item_1,
31         myArrayList);
32         myListView.setAdapter(myAdapter);
33         Thread myThread = new Thread(this);
34         myThread.start();
35     }
36     public void run()
37     {
38         int i = 0;
39         while(true)
40         {
41             try
42             {
43                 Thread.sleep(1000);
44                 in = new Intent(this, TimeService.class);
45                 startService(in);
46                 if(i %5 == 0)
47                 {
48                     Intent intent = new Intent("com.and");
49                     intent.putExtra("hello", "Hello");
50                     intent.putExtra("firstName", "Mir");
51                     intent.putExtra("lastName", "Noman!");
52                     sendBroadcast(intent);
53                 }
54                 if(i%2 == 0)
55                 {
56                     Intent intent = new Intent("com.and");
57                     intent.putExtra("hello", "");
58                     intent.putExtra("firstName", "");
59                     intent.putExtra("lastName", "");
60                     sendBroadcast(intent);
61                 }
62                 i++;
63             }
64             catch (Exception e)
```

```
65         {
66             Log.d("Exception", e.getMessage()+"");
67         }
68     }
69 }
70 @Override
71     public boolean onCreateOptionsMenu(Menu menu)
72     {
73         super.onCreateOptionsMenu(menu);
74         menu.add(0, MENU_INSERT, 0, "Insert Record").setIcon
75             (android.R.drawable.ic_input_add);
76         return true;
77     }
78     @Override
79     public boolean onOptionsItemSelected(int featureId, MenuItem item)
80     {
81         super.onOptionsItemSelected(featureId, item);
82         switch(item.getItemId())
83         {
84             case MENU_INSERT:
85                 Intent intent = new Intent(this, AddActivity.class);
86                 startActivityForResult(intent, ADD_ACTIVIY);
87                 default:
88                     break;
89         }
90         return true;
91     }
92     @Override
93     protected void onActivityResult(int requestCode,
94                                     int resultCode, Intent data)
95     {
96         super.onActivityResult(requestCode, resultCode, data);
97         String str;
98         String token[];
99         switch(requestCode)
100        {
101            case ADD_ACTIVIY:
102                if(Activity.RESULT_OK == resultCode)
103                {
104                    Bundle b = data.getExtras();
105                    str = b.getString("personRec");
106                    token = str.split("!");
107                    person = new Person(token[0], token[1], token[2]);
108                    myArrayList.add(person);
109                }
110                break;
111            default:
112                break;
```

```
113         }
114         myAdapter.notifyDataSetChanged();
115     }
```

5.2 Code of TimeService class

At the end of onCreate() method we start a thread by calling the start method of thread class which will automatically call run() method that is provided by the Runnable interface. From here a new independent thread will start. In the run() method we can write any type of code that we want to execute but here in our case study we have created an Intent and started a service. And has passed the Intent to that service. For the time being we can ignore rest of the code of this class and will jump to the service code. We have created a class called TimeService that extends service and contains a very simple code of a timer. This service calculates and shows for how long the application has been running. As long as the application is running this service will be running in the background. This service displays the time elapsed in the MainActivity through a textview control named timeText. Here is the code of the TimeService class.

```
1 import android.app.Service;
2 import android.content.Intent;
3 import android.os.IBinder;
4 import android.util.Log;
5 public class TimeService extends Service
6 {
7     static int hr = 0;
8     static int min = 0;
9     static int sec = 0;
10    public void onStart(Intent intent, int startId)
11    {
12        String hrStr = "";
13        String minStr = "";
14        if(sec<10)
15            secStr = "0" + sec;
16        else
17            secStr = "" + sec;
18        if(min<10)
19            minStr = "0" + min;
20        else
21            minStr = "" + min;
22        if(hr<10)
23            hrStr = "0" + hr;
```

```
24         else
25             hrStr = "" + hr;
26             MainActivity.timeView.setText("Time Elapsed "
27             + hrStr + ":" + minStr + ":" + secStr);
28         sec++;
29         if(sec == 60)
30         {
31             sec = 0;
32             min++;
33         }
34         if(min == 60)
35         {
36             min = 0;
37             hr++;
38         }
39         if(hr == 12)
40             hr = 0;
41         stopSelf();
42     }
43     @Override
44     public void onCreate()
45     {}
46     @Override
47     public IBinder onBind(Intent intent)
48     {
49         return null;
50     }
51 }
```

5.3 Code of AddActivity class

There is a list view control on the main activity screen. To populate this listview control an option menu is created and in `menuItemSelected()` method an Intent is created. After this the `startActivityForResult()` method is called and the above created Intent is passed to this method. By clicking the option menu button the main activity goes to pause state and the second activity is opened, shown in figure 1.2. In this case the MainActivity is in pause state and the second activity named AddActivity becomes the active activity. When the name, address and phone number fields are filled and the AddRecord button is pressed, the values are returned to the MainActivity and the second activity gets killed by calling the method `finish()`.

The code for AddActivity class is given below.

```
1 import android.app.Activity;
2 import android.app.AlertDialog;
3 import android.content.DialogInterface;
4 import android.content.Intent;
5 import android.os.Bundle;
6 import android.view.View;
7 import android.view.View.OnClickListener;
8 import android.widget.Button;
9 import android.widget.EditText;
10 public class AddActivity extends Activity
11 {
12     EditText nameText;
13     EditText addressText;
14     EditText phoneText;
15     Button addBtn;
16     Person person;
17     String nameStr;
18     String addressStr;
19     String phoneStr;
20     @Override
21     protected void onCreate(Bundle savedInstanceState)
22     {
23         super.onCreate(savedInstanceState);
24         setContentView(R.layout.addperson);
25         nameText = (EditText)findViewById(R.id.name);
26         addressText = (EditText) findViewById(R.id.address);
27         phoneText = (EditText) findViewById(R.id.phone);
28         addBtn = (Button) findViewById(R.id.submit);
29         addBtn.setOnClickListener(new OnClickListener()
30         {
31             public void onClick(View v)
32             {
33                 nameStr = (nameText.getText().toString()).trim();
34                 addressStr = (addressText.getText().toString()).trim();
35                 phoneStr = (phoneText.getText().toString()).trim();
36                 if( nameStr.equals("") || addressStr.equals("")
37                 || phoneStr.equals("") )
38                 {
39                     AlertDialog.Builder ad =
40                     new AlertDialog.Builder(AddActivity.this);
41                     ad.setTitle("Warning!");
42                     if(nameStr.equals(""))
43                         ad.setMessage("Enter Person Name");
44                     else if(addressStr.equals(""))
45                         ad.setMessage("Enter Address");
46                     else if(phoneStr.equals(""))
47                         ad.setMessage("Enter Phone Number");
```

```
48         ad.setPositiveButton("Ok",
49             new DialogInterface.OnClickListener()
50         {
51             public void onClick(DialogInterface dialog,int arg1)
52                 {}
53             }
54         ad.show();
55     }
56     else
57     {
58         person = new Person(nameStr, addressStr, phoneStr);
59         Intent resIntent = new Intent();
60         resIntent.putExtra("personRec", person.myString());
61         setResult(RESULT_OK,resIntent);
62         finish();
63     }
64 }
65 }
66 }
67 }
```

Now coming back to the MainActivity , on the main activity there is textview control that flashes after every 5 seconds. This text message flashing is done through broadcast receivers. The process is explained below.

In the MainActivity class there is a run() method, in this method an Intent named intent is created. After that some hard code values are passed to that intent using the outExtra() method. After this the sendBroadcast() method is called and the intent in is passed to it. After this the MyBroadcastReceiver class is initiated. This class extends the BroadcastReceiver. When the onReceive() method of MyBroadcastReceiver class is called it gets the values from the intent through getStringExtra() method. Then the broadcast receiver calls the setText() method of the MainActivity and set the text of the textview control to the values received.

5.4 The code of the MyBroadcastReceiver class

```
1 import android.content.BroadcastReceiver;
2 import android.content.Context;
3 import android.content.Intent;
4 public class MyBroadcastReceiver extends BroadcastReceiver
5 {
6     @Override
7     public void onReceive(Context context, Intent intent)
```

```
8      {
9      String firtnName  = intent.getStringExtra("firstName");
10     String lastName = intent.getStringExtra("lastName");
11         String hello = intent.getStringExtra("hello");
12     MainActivity.nameView.setText(hello +" "+ firtnName + " " + lastName);
13     }
14 }
```

5.5 The code of the manifest file

In addition to the above classes, all the activities, services and broadcast receivers are added or registered to the AndroidManifest.xml file.

5.5.1 Application Component used

```
1 <activity android:name=".MainActivity"
2         android:label="@string/app_name">
3     <intent-filter>
4         <action android:name="android.intent.action.MAIN" />
5         <category android:name="android.intent.category.LAUNCHER" />
6     </intent-filter>
7 </activity>
8 <activity android:name=".AddActivity"
9         android:label="Add Activity">
10 </activity>
11 <service android:enabled="true" android:name=".TimeService">
12 </service>
13 <receiver android:enabled="true"
14         android:label="My Broadcast Receiver"
15         android:name=".MyBroadcastReceiver">
16     <intent-filter>
17         <action android:name="com.and"/>
18     </intent-filter>
19 </receiver>
```

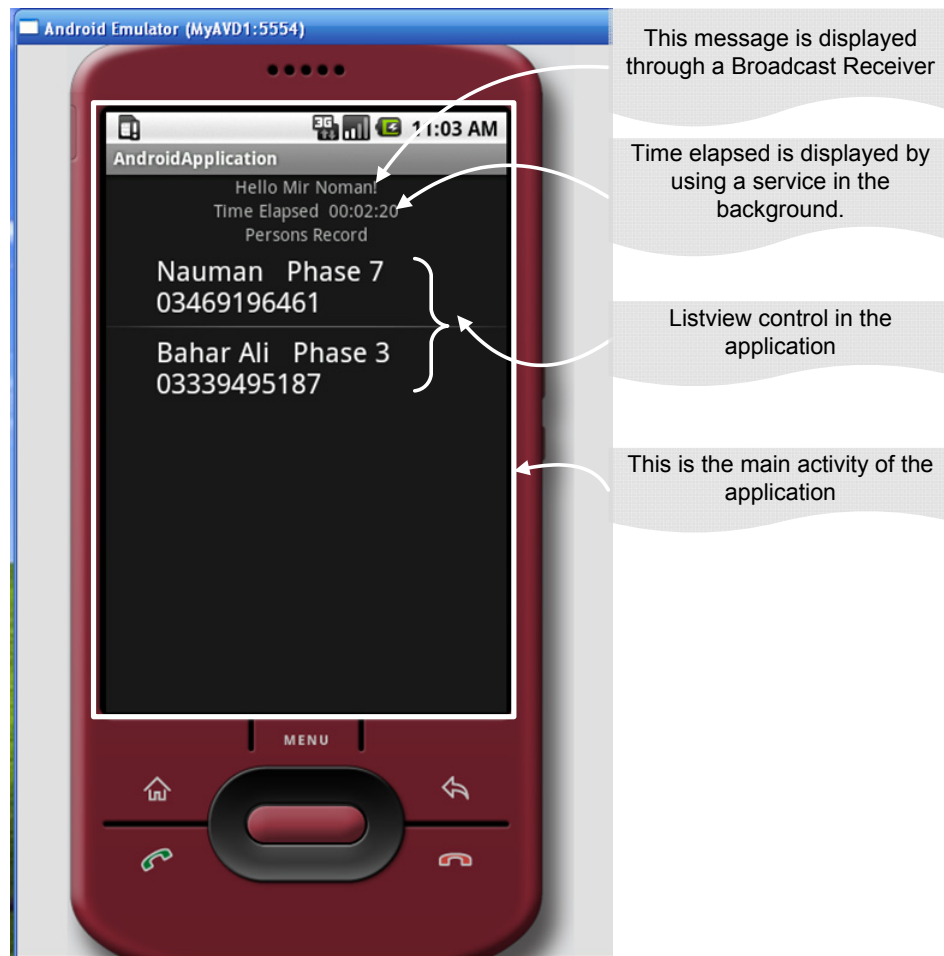


Figure 5.1: The Figure shows the first activity is main activity which displays some text objects and a list view control

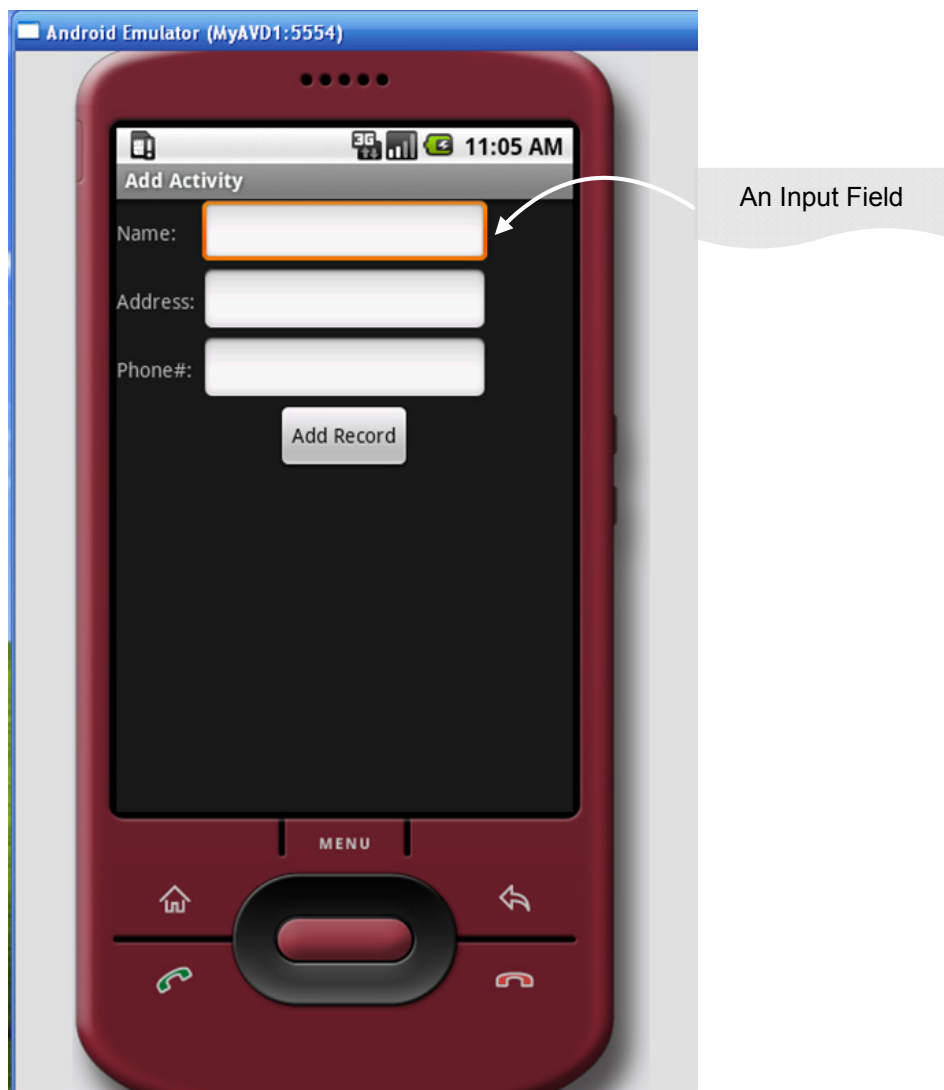


Figure 5.2: The Figure shows the second activity that is a simple form which is used to enter name, address, contact numbers of people

Chapter 6

Intent and Intent Filters

Three of the core components of an application—activities, services, and broadcast receivers—are activated through messages, called intents. Intent messaging is a facility for late run-time binding between components in the same or different applications. The intent itself, an Intent object, is a passive data structure holding an abstract description of an operation to be performed—or, often in the case of broadcasts, a description of something that has happened and is being announced. There are separate mechanisms for delivering intents to each type of component:

An Intent object is passed to `Context.startActivity()` or `Activity.startActivityForResult()` to launch an activity or get an existing activity to do something new. (It can also be passed to `Activity.setResult()` to return information to the activity that called `startActivityForResult()`.)

An Intent object is passed to `Context.startService()` to initiate a service or deliver new instructions to an ongoing service. Similarly, an intent can be passed to `Context.bindService()` to establish a connection between the calling component and a target service. It can optionally initiate the service if it's not already running.

Intent objects passed to any of the broadcast methods (such as `Context.sendBroadcast()`, `Context.sendOrderedBroadcast()`, or `Context.sendStickyBroadcast()`) are delivered to all interested broadcast receivers. Many kinds of broadcasts originate in system code.

In each case, the Android system finds the appropriate activity, service, or set of broadcast receivers to respond to the intent, instantiating them if necessary. There is no overlap within these messaging systems: Broadcast intents are delivered only to broadcast receivers, never to activities or services. An intent passed to

`startActivity()` is delivered only to an activity, never to a service or broadcast receiver, and so on.

This document begins with a description of Intent objects. It then describes the rules Android uses to map intents to components how it resolves which component should receive an intent message. For intents that don't explicitly name a target component, this process involves testing the Intent object against intent filters associated with potential targets.

6.1 Intent objects

An Intent object is a bundle of information. It contains information of interest to the component that receives the intent (such as the action to be taken and the data to act on) plus information of interest to the Android system (such as the category of component that should handle the intent and instructions on how to launch a target activity). Principally, it can contain the following:

6.1.1 Component name

The name of the component that should handle the intent. This field is a `ComponentName` object a combination of the fully qualified class name of the target component (for example `"com.example.project.app.FreneticActivity"`) and the package name set in the manifest file of the application where the component resides (for example, `"com.example.project"`). The package part of the component name and the package name set in the manifest do not necessarily have to match.

The component name is optional. If it is set, the Intent object is delivered to an instance of the designated class. If it is not set, Android uses other information in the Intent object to locate a suitable target see Intent Resolution, later in this document.

The component name is set by `setComponent()`, `setClass()`, or `setClassName()` and read by `getComponent()`.

6.1.2 Action

A string naming the action to be performed or, in the case of broadcast intents, the action that took place and is being reported. The Intent class defines a number of action constants, including these:

Table 6.1: String Table

CONSTANT	TARGET COMPONENT	ACTION
ACTION_CALL	activity	Initiate a phone call
ACTION_EDIT	activity	Display data for the user to edit
ACTION_MAIN	activity	Start up as the initial activity of a task, with no data input and no returned output
ACTION_SYNC	activity	Synchronize data on a server with data on the mobile device
ACTION_BATTERY_LOW	broadcast receiver	A warning that the battery is low
ACTION_HANDSET_PLUG	broadcast receiver	A handset has been plugged into the device, or unplugged from it.
ACTION_SCREEN_ON	broadcast receiver	The screen has been turned on
ACTION_TIMEZONE_CHANGED	broadcast receiver	The setting for the time zone has changed

See the Intent class description for a list of pre-defined constants for generic actions. Other actions are defined elsewhere in the Android API. You can also define your own action strings for activating the components in your application. Those you invent should include the application package as a prefix for example: "com.example.project.SHOW_COLOR".

The action largely determines how the rest of the intent is structured particularly the data and extras fields - much as a method name determines a set of arguments and a return value. For this reason, it's a good idea to use action names that are as specific as possible, and to couple them tightly to the other fields of the intent. In other words, instead of defining an action in isolation, define an entire protocol for the Intent objects your components can handle.

The action in an Intent object is set by the `setAction()` method and read by `getAction()`.

6.1.3 Data

The URI of the data to be acted on and the MIME type of that data. Different actions are paired with different kinds of data specifications. For example, if the action field is `ACTION_EDIT`, the data field would contain the URI of the document to be displayed for editing. If the action is `ACTION_CALL`, the data field would be a tel: URI with the number to call. Similarly, if the action is `ACTION_VIEW` and the data field is an http: URI, the receiving activity would be called upon to download and display whatever data the URI refers to.

When matching an intent to a component that is capable of handling the data, it's often important to know the type of data (its MIME type) in addition to its URI. For example, a component able to display image data should not be called upon to play an audio file.

In many cases, the data type can be inferred from the URI particularly content: URIs, which indicate that the data is located on the device and controlled by a content provider (see theseparate discussion on content providers). But the type can also be explicitly set in the Intent object. The `setData()` method specifies data only as a URI, `setType()` specifies it only as a MIME type, and `setDataAndType()` specifies it as both a URI and a MIME type. The URI is read by `getData()` and the type by `getType()`.

6.1.4 Category

A string containing additional information about the kind of component that should handle the intent. Any number of category descriptions can be placed in an Intent object. As it does for actions, the Intent class defines several category constants, including these:

See the Intent class description for the full list of categories.

The `addCategory()` method places a category in an Intent object, `removeCategory()` deletes a category previously added, and `getCategories()` gets the set of all categories currently in the object.

Table 6.2: String Table

CONSTANT	MEANING
CATEGORY_BROWSABLE	The target activity can be safely invoked by the browser to display data referenced by a link. For example, an image or an e-mail message.
CATEGORY_GADGET	The activity can be embedded inside of another activity that hosts gadgets.
CATEGORY_HOME	The activity displays the home screen, the first screen the user sees when the device is turned on or when the HOME key is pressed.
CATEGORY_LAUNCHER	The activity can be the initial activity of a task and is listed in the top-level application launcher.
CATEGORY_PREFERENCE	The target activity is a preference panel.

6.1.5 Extras

Key-value pairs for additional information that should be delivered to the component handling the intent. Just as some actions are paired with particular kinds of data URIs, some are paired with particular extras. For example, an `ACTION_TIMEZONE_CHANGED` intent has a "time-zone" extra that identifies the new time zone, and `ACTION_HEADSET_PLUG` has a "state" extra indicating whether the headset is now plugged in or unplugged, as well as a "name" extra for the type of headset. If you were to invent a `SHOW_COLOR` action, the color value would be set in an extra key-value pair.

The Intent object has a series of `put...()` methods for inserting various types of extra data and a similar set of `get...()` methods for reading the data. These methods parallel those for Bundle objects. In fact, the extras can be installed and read as a Bundle using the `putExtras()` and `getExtras()` methods.

6.1.6 Flags

Flags of various sorts. Many instruct the Android system how to launch an activity (for example, which task the activity should belong to) and how to treat it after it's

launched (for example, whether it belongs in the list of recent activities). All these flags are defined in the Intent class.

The Android system and the applications that come with the platform employ Intent objects both to send out system-originated broadcasts and to activate system-defined components. To see how to structure an intent to activate a system component, consult the list of intents in the reference.

6.2 Intent Resolution

Intents can be divided into two groups:

Explicit intents designate the target component by its name (the component name field, mentioned earlier, has a value set). Since component names would generally not be known to developers of other applications, explicit intents are typically used for application-internal messages such as an activity starting a subordinate service or launching a sister activity.

Implicit intents do not name a target (the field for the component name is blank). Implicit intents are often used to activate components in other applications.

Android delivers an explicit intent to an instance of the designated target class. Nothing in the Intent object other than the component name matters for determining which component should get the intent.

A different strategy is needed for implicit intents. In the absence of a designated target, the Android system must find the best component (or components) to handle the intent—a single activity or service to perform the requested action or the set of broadcast receivers to respond to the broadcast announcement. It does so by comparing the contents of the Intent object to intent filters, structures associated with components that can potentially receive intents. Filters advertise the capabilities of a component and delimit the intents it can handle. They open the component to the possibility of receiving implicit intents of the advertised type. If a component does not have any intent filters, it can receive only explicit intents. A component with filters can receive both explicit and implicit intents.

Only three aspects of an Intent object are consulted when the object is tested against an intent filter:

action

data (both URI and data type)

category

The extras and flags play no part in resolving which component receives an intent.

6.2.1 Intent filters

To inform the system which implicit intents they can handle, activities, services, and broadcast receivers can have one or more intent filters. Each filter describes a capability of the component, a set of intents that the component is willing to receive. It, in effect, filters in intents of a desired type, while filtering out unwanted intents but only unwanted implicit intents (those that don't name a target class). An explicit intent is always delivered to its target, no matter what it contains; the filter is not consulted. But an implicit intent is delivered to a component only if it can pass through one of the component's filters.

A component has separate filters for each job it can do, each face it can present to the user. For example, the NoteEditor activity of the sample Note Pad application has two filters—one for starting up with a specific note that the user can view or edit, and another for starting with a new, blank note that the user can fill in and save. (All of Note Pad's filters are described in the Note Pad Example section, later.)

6.2.2 Filters and security

An intent filter cannot be relied on for security. While it opens a component to receiving only certain kinds of implicit intents, it does nothing to prevent explicit intents from targeting the component. Even though a filter restricts the intents a component will be asked to handle to certain actions and data sources, someone could always put together an explicit intent with a different action and data source, and name the component as the target.

An intent filter is an instance of the `IntentFilter` class. However, since the Android system must know about the capabilities of a component before it can launch that component, intent filters are generally not set up in Java code, but in the application's manifest file (`AndroidManifest.xml`) as `<intent-filter>` elements. (The one exception would be filters for broadcast receivers that are registered dynamically by calling `Context.registerReceiver()`; they are directly created as `IntentFilter` objects.)

A filter has fields that parallel the action, data, and category fields of an Intent object. An implicit intent is tested against the filter in all three areas. To be delivered to the component that owns the filter, it must pass all three tests. If it fails even one of them, the Android system won't deliver it to the component at least not on the basis of that filter. However, since a component can have multiple intent filters, an intent that does not pass through one of a component's filters might make it through on another.

Each of the three tests is described in detail below:

Action test

An `<intent-filter>` element in the manifest file lists actions as `<action>` subelements. For example:

```
<intent-filter . . . >

<action android:name="com.example.project.SHOW_CURRENT" />

<action android:name="com.example.project.SHOW_RECENT" />

<action android:name="com.example.project.SHOW_PENDING" />

. . .

</intent-filter>
```

As the example shows, while an Intent object names just a single action, a filter may list more than one. The list cannot be empty; a filter must contain at least one `<action>` element, or it will block all intents.

To pass this test, the action specified in the Intent object must match one of the actions listed in the filter. If the object or the filter does not specify an action, the results are as follows:

If the filter fails to list any actions, there is nothing for an intent to match, so all intents fail the test. No intents can get through the filter.

On the other hand, an Intent object that doesn't specify an action automatically passes the test as long as the filter contains at least one action.

Category test

An `<intent-filter>` element also lists categories as subelements. For example:

```
<intent-filter . . . >  
  
<category android:name="android.intent.category.DEFAULT" />  
  
<category android:name="android.intent.category.BROWSABLE" />  
  
. . .  
  
</intent-filter>
```

Note that the constants described earlier for actions and categories are not used in the manifest file. The full string values are used instead. For instance, the "android.intent.category.BROWSABLE" string in the example above corresponds to the `CATEGORY_BROWSABLE` constant mentioned earlier in this document. Similarly, the string "android.intent.action.EDIT" corresponds to the `ACTION_EDIT` constant.

For an intent to pass the category test, every category in the Intent object must match a category in the filter. The filter can list additional categories, but it cannot omit any that are in the intent.

In principle, therefore, an Intent object with no categories should always pass this test, regardless of what's in the filter. That's mostly true. However, with one exception, Android treats all implicit intents passed to `startActivity()` as if they contained at least one category: "android.intent.category.DEFAULT" (the `CATEGORY_DEFAULT` constant). Therefore, activities that are willing to receive implicit intents must include "android.intent.category.DEFAULT" in their intent filters. (Filters with "android.intent.action.MAIN" and "android.intent.category.LAUNCHER" settings are the exception. They mark activities that begin new tasks and that are represented on the launcher screen. They can include "android.intent.category.DEFAULT" in the list of categories, but don't need to.) See Using intent matching, later, for more on these filters.)

Data test

Like the action and categories, the data specification for an intent filter is contained in a subelement. And, as in those cases, the subelement can appear multiple times, or not at all. For example:

```
<intent-filter . . . >  
  
<data android:mimeType="video/mpeg" android:scheme="http" . . . />
```

```
<data android:mimeType="audio/mpeg" android:scheme="http" . . . />
. . .
</intent-filter>
```

Each `<data>` element can specify a URI and a data type (MIME media type). There are separate attributes `scheme`, `host`, `port`, and `path` for each part of the URI:

```
scheme://host:port/path
```

For example, in the following URI,

```
content://com.example.project:200/folder/subfolder/etc
```

the scheme is "content", the host is "com.example.project", the port is "200", and the path is "folder/subfolder/etc". The host and port together constitute the URI authority; if a host is not specified, the port is ignored.

Each of these attributes is optional, but they are not independent of each other: For an authority to be meaningful, a scheme must also be specified. For a path to be meaningful, both a scheme and an authority must be specified.

When the URI in an Intent object is compared to a URI specification in a filter, it's compared only to the parts of the URI actually mentioned in the filter. For example, if a filter specifies only a scheme, all URIs with that scheme match the filter. If a filter specifies a scheme and an authority but no path, all URIs with the same scheme and authority match, regardless of their paths. If a filter specifies a scheme, an authority, and a path, only URIs with the same scheme, authority, and path match. However, a path specification in the filter can contain wildcards to require only a partial match of the path.

The `type` attribute of a `<data>` element specifies the MIME type of the data. It's more common in filters than a URI. Both the Intent object and the filter can use a "*" wildcard for the subtype field for example, "text/*" or "audio/*" indicating any subtype matches.

The `dataTest` compares both the URI and the data type in the Intent object to a URI and data type specified in the filter. The rules are as follows:

An Intent object that contains neither a URI nor a data type passes the test only if the filter likewise does not specify any URIs or data types.

An Intent object that contains a URI but no data type (and a type cannot be inferred from the URI) passes the test only if its URI matches a URI in the filter and the filter likewise does not specify a type. This will be the case only for URIs like `mailto:` and `tel:` that do not refer to actual data.

An Intent object that contains a data type but not a URI passes the test only if the filter lists the same data type and similarly does not specify a URI.

An Intent object that contains both a URI and a data type (or a data type can be inferred from the URI) passes the data type part of the test only if its type matches a type listed in the filter. It passes the URI part of the test either if its URI matches a URI in the filter or if it has a `content:` or `file:` URI and the filter does not specify a URI. In other words, a component is presumed to support `content:` and `file:` data if its filter lists only a data type.

If an intent can pass through the filters of more than one activity or service, the user may be asked which component to activate. An exception is raised if no target can be found.

Common cases

The last rule shown above for the data test, rule (d), reflects the expectation that components are able to get local data from a file or content provider. Therefore, their filters can list just a data type and do not need to explicitly name the `content:` and `file:` schemes. This is a typical case. A `<data>` element like the following, for example, tells Android that the component can get image data from a content provider and display it:

```
<data android:mimeType="image/*" />
```

Since most available data is dispensed by content providers, filters that specify a data type but not a URI are perhaps the most common.

Another common configuration is filters with a scheme and a data type. For example, a `<data>` element like the following tells Android that the component can get video data from the network and display it:

```
<data android:scheme="http" android:type="video/*" />
```

Consider, for example, what the browser application does when the user follows a link on a web page. It first tries to display the data (as it could if the link was to an HTML page). If it can't display the data, it puts together an implicit intent

with the scheme and data type and tries to start an activity that can do the job. If there are no takers, it asks the download manager to download the data. That puts it under the control of a content provider, so a potentially larger pool of activities (those with filters that just name a data type) can respond.

Most applications also have a way to start fresh, without a reference to any particular data. Activities that can initiate applications have filters with "android.intent.action.MAIN" specified as the action. If they are to be represented in the application launcher, they also specify the "android.intent.category.LAUNCHER" category:

```
<intent-filter . . . >

<action android:name="code android.intent.action.MAIN" />

<category android:name="code android.intent.category.LAUNCHER" />

<intent-filter>
```

6.2.3 Using intent matching

Intents are matched against intent filters not only to discover a target component to activate, but also to discover something about the set of components on the device. For example, the Android system populates the application launcher, the top-level screen that shows the applications that are available for the user to launch, by finding all the activities with intent filters that specify the "android.intent.action.MAIN" action and "android.intent.category.LAUNCHER" category (as illustrated in the previous section). It then displays the icons and labels of those activities in the launcher. Similarly, it discovers the home screen by looking for the activity with "android.intent.category.HOME" in its filter.

Your application can use intent matching in a similar way. The PackageManager has a set of query...() methods that return all components that can accept a particular intent, and a similar series of resolve...() methods that determine the best component to respond to an intent. For example, queryIntentActivities() returns a list of all activities that can perform the intent passed as an argument, and queryIntentServices() returns a similar list of services. Neither method activates the components; they just list the ones that can respond. There's a similar method, queryBroadcastReceivers(), for broadcast receivers.

Chapter 7

Android Security Architecture

7.1 Introduction

Android is build on Linux kernel 2.6. The security architecture and permission model are therefore same on Android as that on linux OS. This makes Android to be a multi processing system, and running each application in its own separate process. Security is implemented at process level in Android. It implements security procedures through different mechanisms at different levels. This includes implementing security at application level through user and group ID's. At component level it makes use of permission mechanism to restrict access to specific component while at data level it implements security through per URI basis permissions.

Android architecture is defined such that no application can perform an operation on any other application, its components or its data, such as, Reading and/or writing. This restricts access to Private data (such as contacts or e-mails), another application's files, performing network access, keeping the device awake etc [10].

The only way to get access to any component or data is to explicitly declare the permissions it needs for that specific additional capabilities.

Application level security is discussed in this chapter, while components and data level security is discussed in followed chapter.

7.2 Application Level Security

7.2.1 Security implemented by Application Signing using Certificates mechanisms

The author of application is identified by a certificate. This certificate is signed by the developer of that particular application, whose private key is held by the same person. The certificate impacts the security in the following way:

To establish a trust full relation between different applications.

Who can access the signature based permissions.

Who can share the user id.

7.2.2 Security enforcement using USER ID

As Android is based on Linux kernel, so the security policy enforced is same as that implemented for Linux based system. At the time of installation of an application on Android, its assigned with a unique user ID. This user ID will be remain same throughout the application life time. Assigning User ID prevents applications to access other applications or their data, as the data associated with a particular application is assigned the ID of that application.

7.2.3 Security enforced on FILE

The file stored with an application can be accessed that is read and write by other applications by explicitly setting flags `MODE_WORLD_READABLE` and or `MODE_WORLD_WRITEABLE` with that file. These flags set the mode of file access to readable and writable.

The flags can be set and retrieved by using following fuctions:

`getSharedPreferences (String,int)` //This function Retrieves and modify the contents of Preferences, that are stored with that particular file.

`openFileOutput(String,int)` //This function opens a file that is associated with the Context's application package for writing. The file will be created if it doesn't exist.

`openOrCreateDatabase(String,int, SQLiteDatabase.CursorFactory)` //This function opens a new SQLiteDatabase that is associated with the Context's application package. The database file will be created if it doesn't exist.

7.3 Related Work

We include here a brief description of researches carried out in android security and its permission model. A few models have been proposed to make Android framework more secure against security threats.

7.3.1 Semantically Rich Application-Centric Security in Android

The author have discussed existing Android framework and then presented a modified infrastructure known as SAINT (Secure Application Interaction), that governs install time permission assignment and their run-time use as dictated by application provider policy. They also presented in depth description of the semantics of application policy. According to the author Saint provides necessary utility for applications to assert and control the security decisions on the platform. [13]

7.3.2 Towards Formal Analysis of the Permission-based Security Model for Android

The author have specified the permission mechanism for the system. They have presented the system in terms of a state machine, elucidate the security needs, and showed that the specified system is secure over the specified states and transitions. They have claimed that their work will provide the basis for assuring the security of the Android system. The specification and verification were carried out using the Coq proof assistant [17].

7.3.3 Developing Secure Mobile Applications for Android. An introduction to making secure Android applications

It discusses the security model of Android, including many of the key security mechanisms and how one can use them safely. While it is targeted towards applications developers, it provides a useful background for those intending to change

or extend the platform. The author concludes by a note on 'root' access code with root access, that is useful but dangerous. Obviously some system processes like the Zygote need root to do their jobs. On the emulator it is easy to become root (su, adb shell, etc), but Android distributions trying slow recovery of data from lost phones or meet government regulations might not allow this. One may find needing root access, for example to create some super encrypted and compressed virtual memory feature. This sort of thing is easy for custom builds of Android, running on emulators. Turning on security is actually a feature of an Android distribution. If the SystemProperty ro.secure is not set to 1, then the platform makes little effort at security [4]

7.4 Security Flaws in Android

Til now, less or more , security holes have been found in Android operating system. They have been reported to Google's team, so that they fix these issues and deliver solution by distributing the patches or by providing updates to the system software. We have included few of these flaws. The reasons to include them here is that the reader understand importance of security in cell phones technology. Moreover it provides research vision to those who are working on security of mobile phones (particularly Android). The security threats mentioned are classified on basis of types of attacks and google's response to these threats are also discussed.

7.4.1 Vulnerability to image processing libraries

Vulnerability reported on: March 06, 2008

Researchers have found some holes in Google's Android SDK that could make the software vulnerable to hack attacks.

Core Security published an alert stating that it had found eight vulnerabilities related to some open-source image processing libraries in Google's Android SDK, which the group claims are outdated. Attackers exploiting these vulnerabilities could take complete control of Android handsets. [14]

7.4.2 Security hole in web browser

Scurity hole reported on: Sat Oct 25, 2008

A security flaw discovered in the Android phones. A security hole in the browser, allowing Trojans to install themselves in the same security partition as the browser. The risk in the Google design, security flaw in Android Web Browser posted lies in the danger from within the Web browser partition in the phone. It would be possible, for example, for an intruder to install software that would capture keystrokes entered by the user when surfing to other Web sites. That would make it possible to steal identity information or passwords. [7]

7.4.3 Bugs that leads to Denial of Service attack

There have been two bugs reported to Ocert .org related to Denial of Service attacks in Android operating system. We discuss here both of them briefly

a. DoS vulnerability :

Bug reported on: 24 April, 2009

Leads to restarting the system process.

This Issue was discovered in the Android's Dalvik API. A specific malicious application can be crafted so that if it is downloaded and executed by the user, it would trigger the vulnerable API function and restart the system process. The same condition could occur if a developer unintentionally places the vulnerable function in a place where the execution path leads to that function call [2].

b. Loss of connectivity :

Bug reported on Date :19 June,2009

This bug concerns Android handling of SMS messages in which a specific malformed SMS message can be crafted to trigger a condition that disconnects the mobile phone from the cellular network. The malformed SMS message consists of a badly formatted WAP Push message which causes an `Java ArrayIndexOutOfBoundsException` in the phone application (`android.com.phone`).

The phone application silently restarts without user awareness, this leads to a temporary loss of connectivity (as well as dropping of current calls, if any) [2].

7.5 GOOGLE'S response to reported Security Threats

Google responded to most of flaws that were reported by releasing the patches for it or by providing updates to the system software.

7.5.1 Patch For Android Security Flaw Released By Google And T-Mobile

System update released on :November 4th, 2008

Google has started issuing a patch to tighten up a well-publicized security hole in its Google Android mobile operating system. The patch is being pushed out to users in the form of a system update and users are advised to update their sytem software. The flaw and the PoC were communicated to Google on October 20th, with the vulnerability itself that made it possible Android's use of outdated third-party software packages

Users of the G1 Android phone on Friday have begun receiving a software update that fixes a flaw that security researchers found. The update included the fix to the browser vulnerability and a couple of other minor changes as well [5].

7.5.2 Patch for Android Malformed SMS DoS

Patch released on: 30 July,2009.

Android Security team publicly release patch to open source Android platform providing solution to Malformed SMS DoS attacks [2].

7.5.3 Patch for Dalvik API DoS

Patch released on: 21 July,2009,

Android Security team agrees that one issue has a security impact, does not oppose to advisory release. Android security team committed patch to open source Android repository [6].

Chapter 8

Android Permissions

8.1 Components Level Security

8.1.1 Security through Permission Mechanism

As discussed before Android application framework is designed such that it harms no other application or users data. In other words an application can't access any component of another application until explicit permissions are declared for using that particular components, service or accessing application data or user data. These permissions are declared in manifest.xml file, using `<uses-permission>` tag.

In the flow diagram we present permission enforcement at install time of an application asking user to grant or deny permissions requested.

8.1.2 Permission enforcement during a Program operation

Permissions can be enforced in the following manner during the executing and operation of an installed application.

8.1.3 Declaration of Permissions in Android Manifest.xml file

Each permission is to be declared in the android manifest.xml file before they can be enforced. We include a sample code snippet of the android manifest.xml file , enlightening the permissions declaration and their use.

Install time permissions

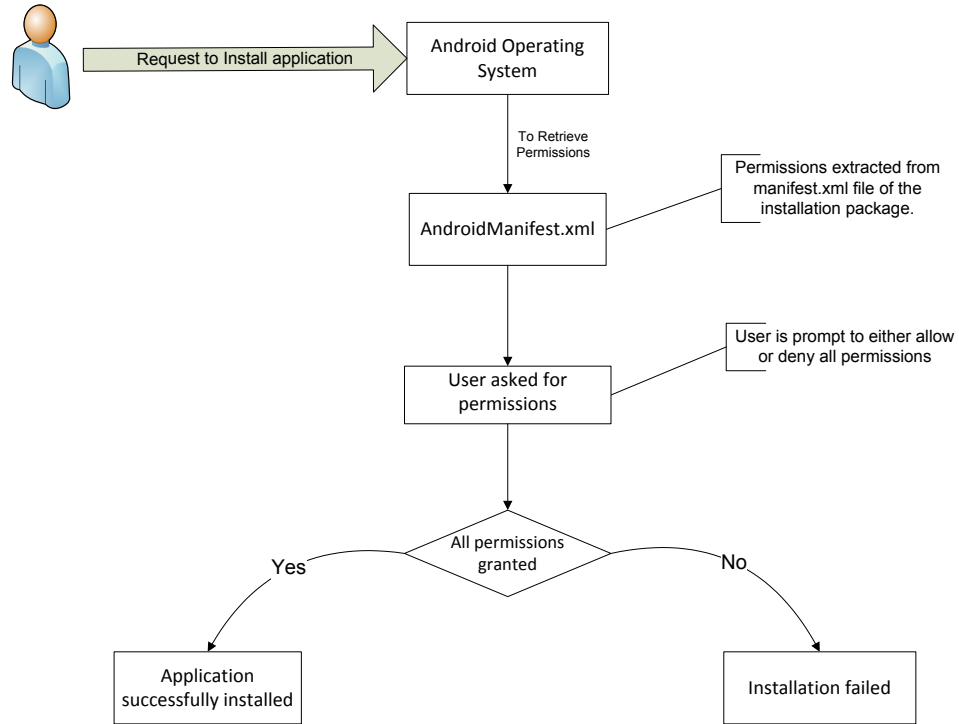


Figure 8.1: Install time permissions

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <manifest xmlns:android="http://schemas.android.com/apk
3 /res/android" package="com.serg.simpleintent"
4 android:versionCode="1"
5 android:versionName="1.0">
6 <permission
7 //all the permission are declared under this tag
8 android:name="perm"
9 //name of the permission,which other components
10 //and packages can use when referring to this permission.
11 android:label="@string/perm"
12 //label for permission shown when the user view a permission,
13 android:permissionGroup="@string/permission
14 //optional, Specifies the name of a group this permission
15 //is associated with
16 android:description="@string/Launches_activity">
17 //descriptive text about permission that can be few lines,
18 //telling user the details about the permission
19 android:protectionLevel="dangerous"
  
```

```

20 //indicates procedure the system should follow to determine
21 //how to grant the permission to an application requesting it.
22 </permission>
23 </manifest>

```

There are four protection levels defined.

Normal: System always grants this type of permission at installation, without asking for the user’s explicit approval. Its a low risk permission.

Dangerous: A higher-risk permission that may give access to private user data or control over the device. Such permissions are granted by the user, and the system can not grant them.

Signature: System grant it if the requesting application is signed with the same certificate as the application that declared the permission

Signature or System: System grant it only to packages in the Android system image or that are signed with the same certificates.

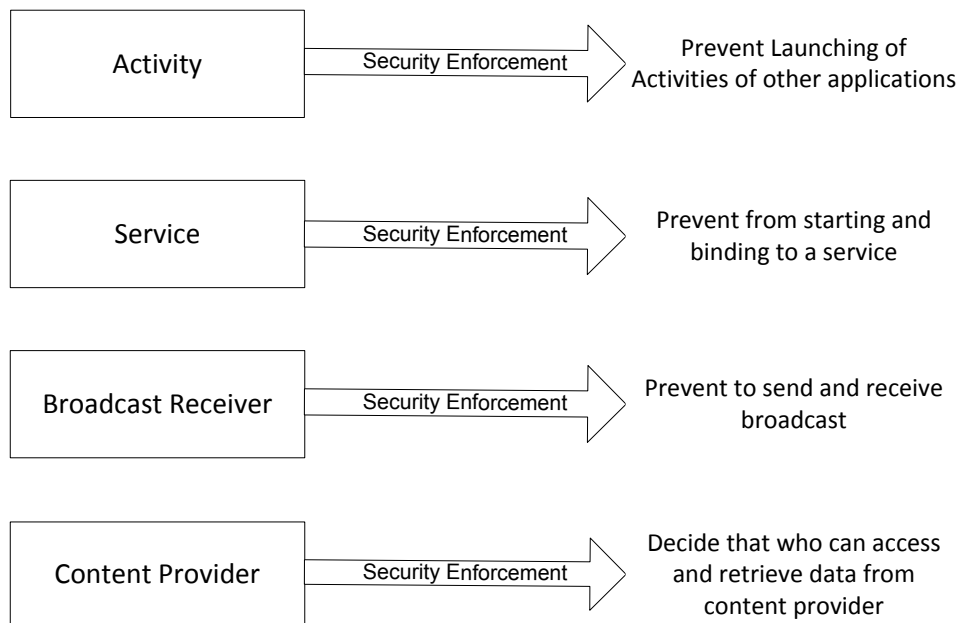


Figure 8.2: Permission enforcement during a program operation

8.1.4 Enforcing Permissions in Android manifest.xml

Android enforces permissions to each component of the system or application as a whole, declared in manifest.xml file. The table describes enforcement at each components and that how these permission will restrict that component. It also shows how the permissions are checked and the action taken if the components fails to acquire that particular permission.

Table 8.1: ENFORCING PERMISSIONS IN AndroidManifest.xml

ANDROID COMPONENTS	PERMISSIONS RESTRICTS	PERMISSION CHECKED DURING	ACTION ON FAILING TO ACQUIRE PERMISSION
Activity	Starting of an associated activity	<code>Context.startActivity()</code> , <code>Activity.start</code> <code>ActivityForResult()</code> .	Security Exception is thrown from the call
Service	Starting or binding to associated service	<code>Context.startService()</code> , <code>Context.stopService()</code> , <code>Context.bindService()</code> .	Security Exception is thrown from the call
Broad Cast Receiver	Sending broadcast to associated receiver	<code>Context.sendBroadcast()</code> .	Not results in an exception being thrown back to the caller, it will just not deliver the intent
Content Provider	Access to data in content provider	<code>ContentResolver.query()</code> , <code>ContentResolver.insert()</code> , <code>ContentResolvr.updat()</code> , <code>ContentResolver.delete()</code>	SecurityException being thrown from the call

8.1.5 Checking Permissions against a Process

Permission can be invoked by a number of ways.

If a call is coming from another process then we can use `Context.checkCallingPermission()` method.

If you have the pid of another process, you can use the Context method `Context.checkPermission(String, int, int)`.

If you have the package name of another application, you can use the direct `PackageManager` method `PackageManager.checkPermission(String, String)`

8.2 Data Level Security

8.2.1 Security through PER-URI permissions

Data needs to be protected with read / write permissions that are provided by per URI basis. Such permissions are applied to the content providers that give access to the data. The `Intent.FLAG_GRANT_READ_URI_PERMISSION` and `Intent.FLAG_GRANT_WRITE_URI_PERMISSION` sets the receiving activity access to specific data URI in the Intent.

These are adhoc based fine- grained permissions. Further operations on URI permissions can be done with `Context.grantUriPermission()`, `Context.revokeUriPermission()`, and `Context.checkUriPermission()` methods.

These are explained here briefly.

Context.grantUriPermission() :

To access a specific Uri to another package, no matter whether that package has general permission to access the Uri's content provider. It can be useful in case where response is expected after users interaction, for example, allowing another application to open an attachment for that user.

Context.revokeUriPermission() :

Remove all permissions to access a particular content provider Uri that were previously added with `grantUriPermission(String, Uri, int)`.

Context.checkUriPermission() :

Determine whether a particular process and user ID has been granted permission to access a specific URI.

8.3 Android Security Exceptions

Google is among one of the more secure mobile operating systems. There are certain security features that makes android to be more secure. Some of these features are multiprocessing that allows each application to run in its own process without having access to any component of another application. Implemented on Linux platform enables security based on User and Group ID's. Access restriction and signature mechanism are two other features that make hard for attackers to run harmful scripts. Open Source nature and enlisting user's help let a combine effort in making android more secure. Similarly asking permissions on access to sensitive data or services, enforcing audio and video to run on outside media servers, will block the access to cookies and user credentials and are key security features that are implemented in Andoidr OS.

However, there are a number of exceptions.

- Public Vs Private Components:

- > Exception: Components can be public or private.

- By making a component private, the developer doesn't need to worry which permission label to assign it or how another application might acquire that label.

- Default is dependent on 'intent-filter' rules

- The manifest schema defines an 'exported' attribute

- Why: Protect internal components

- Especially useful if a 'sub-Activity' returns a result

- > Implication: Components may unknowingly be (or become) accessible to other applications.

- > Best Practice: Always set the 'exported' attribute.

- Implicitly Open Components :

- > Exception: If the manifest file does not specify an access permission on a public component, any component in any application can access it.

- e.g., the main Activity for an Application

- Permissions are assigned at install-time

- Why: Some components should provide 'global' access

- > Implication: Unprivileged applications have access

-> Best Practice: Components without access permissions should be exceptional cases, and inputs must be scrutinized (consider splitting components).

- Broadcast Intent Permissions :

-> Exception: The code broadcasting an Intent can set an access permission restricting which Broadcast Receivers can access the Intent.

Why: Define what applications can read broadcasts

-> Implication: If no permission label is set on a broadcast, any unprivileged application can read it.

-> Best Practice: Always specify an access permission on Intent broadcasts (unless explicit destination).

- Content Provider Permissions :

-> Exception: Content Providers have two additional security features

Separate 'read' and 'write' access permission labels

URI permissions allow record level delegation (added Sep 2008)

Why: Provide control over application data

-> Implication: Content sharing need not be all or nothing

URI permissions allow delegation (must be allowed by Provider)

-> Best Practice: Always define separate read and write permissions.

Allow URI permissions when necessary

- Service Hooks:

-> Exception: A component (e.g., Service) may arbitrarily invoke the `checkPermission()` method to enforce ICC.

Why: Allows Services to differentiate access to specific methods.

-> Implication: The application developer can add reference monitor hooks

-> Best Practice: Use `checkPermission()` to mediate 'administrative' operations.

Alternatively, create separate Services

- Protected APIs:

-> Exception: The system uses permission labels to mediate access to certain resource APIs resources

Why: The system needs to protect network and hardware

e.g., Applications request the `android.permission.INTERNET` label to make network connections.

-> Implication: Allows the system or a user to assess how 'dangerous' an application may be.

-> Best Practices: Judiciously request permissions for protected APIs

- Permission Protection Levels

-> Exception: Permission requests are not always granted

Permissions can be:

normal - always granted

dangerous - requires user approval

signature - matching signature key

signature or system - same as signature, but also system apps

Why: Malicious applications may request harmful permissions

-> Implication: Users may not understand implications when explicitly granting permissions.

-> Best Practice: Use signature permissions for application 'suites' and dangerous permissions otherwise include informative descriptions

- Pending Intents:

-> Exception: `PendingIntent` objects allow another application to 'finish' an operation for you via RPC.

Introduced in the v0.9 SDK release (August 2008)

Execution occurs in the originating application's 'process' space

Why: Allows external applications to send to private components

Used in a number of system APIs (Alarm, Location, Notification)

-> Implication: The remote application can "All in unspecified values.

May influence the destination and/or data integrity

Allows a form of delegation

-> Best Practice: Only use Pending Intents as 'delayed callbacks' to private Broadcast Receivers/Activities and always fully specify the Intent destination.

Chapter 9

The Android Application Life Cycle

The lifecycle of an application starts when it is installed on android, for our example we will use either the Android Emulator or the G1 phone by HTC. The primary and most used way to install applications on your Android based phone is by using Android Market. The Android market is a centralized repository of all the applications developed for Android. It has been modeled after the very popular Apple iPhone AppStore. Third party developers can upload their applications on the Android market and then anyone can download them. Most of the applications are free but some are also paid.

To install applications using Android Market perform the following steps:

- Open the Android Market application in the Applications menu.
- User can select any option - Applications, Games, Search or My Downloads based on what User want to do. User will then get a list of applications or games to choose from.
- User can also search for any particular application. User will then get a list of apps with their description and rating.
- Click on the application you want and you will get more details of that application - the number of downloads and user comments.
- To install it, just click on the Install button on the bottom of the screen.

In the next screen, it will give more details about the application including the

different functionalities it will need to access. To be more precise the application will show a list of resources and permissions that the application needs to work on android. Here one thing should be kept in mind that the list of permissions that is displayed is not editable and does not have any choices. So if the user wants application to run on android, he must allow access to all the resources for which the application is asking permission. If the user denies to give the permissions, the application will simply not install. Its an 'All or None' policy, which means that the user has to grant all permission to make the application successfully install or the user won't be able to even install the application.

Every application must have an `AndroidManifest.xml` file (with precisely that name) in its root directory. The manifest presents essential information about the application to the Android system, information the system must have before it can run any of the application's code. Among other things, the manifest does the following:

It names the Java package for the application. The package name serves as a unique identifier for the application. It describes the components of the application i.e. the activities, services, broadcast receivers, and content providers that the application is composed of. It names the classes that implement each of the components and publishes their capabilities (for example, which Intent messages they can handle). These declarations let the Android system know what the components are and under what conditions they can be launched. It determines which processes will host application components. It declares which permissions the application must have in order to access protected parts of the API and interact with other applications. It also declares the permissions that others are required to have in order to interact with the application's components. It lists Instrumentation classes that provide profiling and other information as the application is running. These declarations are present in the manifest only while the application is being developed and tested, they're removed before the application is published. It declares the minimum level of the Android API that the application requires.

It lists the libraries that the application must be linked against.

The permissions required by an application to successfully install it and to successfully run it are defined in the `AndroidManifest.xml` file. The android permissions and how the android permission model works are explained in detail in Chapter 8, Android Permissions.

Another way of installing application would be, installing the non-market applications.

- Download and install Google Android SDK. (<http://developer.android.com/sdk/index.html>)
- Now type adb in a command shell will display all the options,adb.exe is SDK tool used to install applications and interface with the device.
- Now Connect Your G1 Phone to your computer using USB cable. You need to install Drivers for this (<http://developer.android.com/sdk/win-usb.html>) .This driver is required for adb to interface with an android G1 phone using USB cable.
- Go to Android Settings SD card and phone storage and disable Use for USB storage. You can enable it again later after you installed your third-party application
- Go to Settings/Application settings and enable Unknown sources.
- Download the Application installer .apk file to a local folder on your computer, Now type below command at command shell to install app.
- adb install <Full Path to apk file><apk file name>

To uninstall applications that user installed using the Android Market:

- Open the Google Android Menu.
- Go to the Settings icon and select Applications.
- Next, click on Manage.
- You will be presented with a list of applications you have installed.
- Select the application you want to uninstall, and click the Uninstall button.

Unlike most traditional environments, Android applications have no control over their own life cycles. Instead, application components must listen for changes in the application state and react accordingly, taking particular care to be prepared for untimely termination.

By default, each Android application is run in its own process that's running a separate instance of Dalvik. Memory and process management of each application is handled exclusively by the run time.

While uncommon, it's possible to force application components within the same application to run in different processes or to have multiple applications share the

same process using the `android:process` attribute on the affected component nodes within the manifest.

Android aggressively manages its resources, doing whatever it takes to ensure that the device remains responsive. This means that processes (and their hosted applications) will be killed, without warning if necessary, to free resources for higher-priority applications - generally those that are interacting directly with the user at the time. The prioritization process is discussed as follows.

The order in which processes are killed to reclaim resources is determined by the priority of the hosted applications. An application's priority is equal to its highest-priority component. Where two applications have the same priority, the process that has been at a lower priority longest will be killed first. Process priority is also affected by interprocess dependencies; if an application has a dependency on a Service or Content Provider supplied by a second application, the secondary application will have at least as high a priority as the application it supports.

All Android applications will remain running and in memory until the system needs its resources for other applications.

The figure shows the priority tree used to determine the order of application termination. It's important to structure your application correctly to ensure that its priority is appropriate for the work it's doing. If you don't, your application could be killed while it's in the middle of something important. The following list details each of the application states shown in Figure, explaining how the state is determined by the application components comprising it:

9.1 Active Processes

Active (foreground) processes are those hosting applications with components currently interacting with the user. These are the processes Android is trying to keep responsive by reclaiming resources. There are generally very few of these processes, and they will be killed only as a last resort.

Active processes include:

- Activities in an 'active' state; that is, they are in the foreground and responding to user events. You will explore Activity states in greater detail later in this chapter.

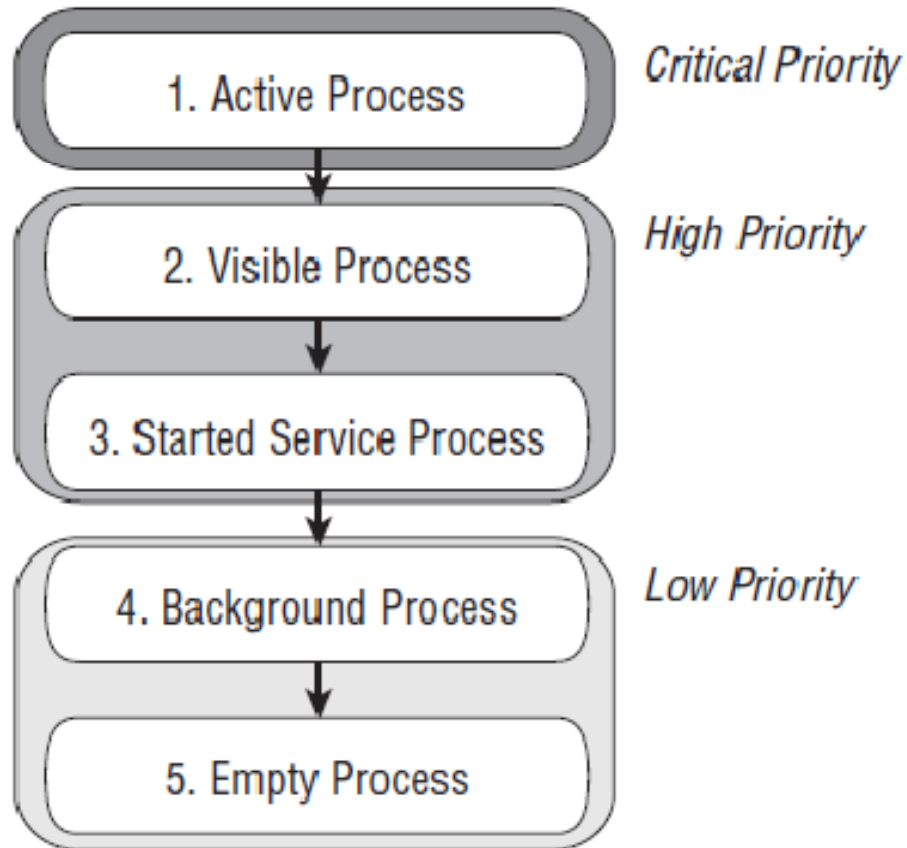


Figure 9.1: Priority tree used to determine the order of application termination

- Activities, Services, or Broadcast Receivers that are currently executing an `onReceive` event handler.
- Services that are executing an `onStart`, `onCreate`, or `onDestroy` event handler.

9.2 Visible Processes

Visible, but inactive processes are those hosting 'visible' Activities. As the name suggests, visible Activities are visible, but they aren't in the foreground or responding to user events. This happens when an Activity is only partially obscured (by a non-full-screen or transparent Activity). There are generally very few visible processes, and they'll only be killed in extreme circumstances to allow active processes to continue.

9.2.1 Started Service Processes

Processes hosting Services that have been started. Services support ongoing processing that should continue without a visible interface. Because Services don't interact directly with the user, they receive a slightly lower priority than visible Activities. They are still considered to be foreground processes and won't be killed unless resources are needed for active or visible processes.

9.2.2 Background Processes

Processes hosting Activities that aren't visible and that don't have any Services that have been started are considered background processes. There will generally be a large number of background processes that Android will kill using a last-seen-first-killed pattern to obtain resources for foreground processes.

9.2.3 Empty Processes

To improve overall system performance, Android often retains applications in memory after they have reached the end of their lifetimes. Android maintains this cache to improve the start-up time of applications when they're re-launched. These processes are routinely killed as required.

Chapter 10

Conclusion

Android is the first comprehensive open source mobile software stack introduced in the market by Open Handset Alliance (OHA - A global alliance of leading technology and mobile industries). It consists of complete mobile operating system supported by Linux kernel, a newly built Dalvik virtual machine, and some modern day mobile applications. Android system architecture is composed of four layers. Mobile applications are on upper most layer, underneath that the second layer contains the application frame work. The application framework is a built-in toolkit, which provides set of services to the Android developers in order to build innovative and efficient Android applications. The application architecture is flexible enough to provide reusability depending upon security policies constrain imposed by underlying application framework. The third layer provides the *c/c++* native libraries and Android Runtime (which is further consist of two modules, Dalvik virtual machine and Android core libraries which reflects the functionality of core libraries written in java). The last layer is Linux kernel that manages low level resources; such as memory management, power management, hardware drivers, process management, etc. The Android application is categorized into four components. These are Activities, services, broadcast receivers, and content providers. Android's application may comprise of one or more such components, depending on their architecture and design. An Activity is a visual screen through which user interact with mobile device and its applications. Service runs in background and cannot be seen by the mobile user (For example, playback music). Broadcast receiver reacts to broadcast announcements initiated by application (Every application has its own broadcast receiver, if needed so). For example, browser generate broadcast announcement that the requested download has been completed and it's now available for other applications to use. Finally, the fourth possible component of an application is

content provider. In Android there is not any sort of centralized database storage that is available for all Android Packages. Content providers are the only way to store, retrieve, and share information among applications. There are separate content providers for different data types like images, audio, videos, etc. The data stored by content provider is only accessible through specified application's object, i.e. content resolver.

We have incorporated a case study, which describes the application components life cycle. It includes two activities, a broadcast receiver, and a service. The service executes to prompt the main activity in order to display the timer to user on mobile screen. There are three intents in our case study example; (1) - describes the action of switching between two activities, (2) - Request the action to start timer service, (3) - it asks the broadcast receiver to receive and announce the broadcast initiated by the application.

Android system implements security at process level. Variables, such as user IDs and Group IDs are use to identify the applications, which in turn use to control access of that application. The components of one application could access the services provided by other application's components, this inter component communication is controlled through permissions assigned in AndroidManifest.xml file. The URI based security permissions further refines the control access to any application's component. The Android's policy enforcement is mandatory; all permission labels are set at install time and can't change until the application is reinstalled. Android's permission label model only restricts access to components and doesn't currently provide information flow guarantee. Android is newly born operating system, which is yet to get mature. Google will have to do a lot in order to make Android the most secure mobile operating system. Google's Android Security Team is working on Android to remove reported security holes in order to make Android more powerful and secure platform.

References

- [1] . Android manifest file. Available at: <http://developer.android.com/guide/topics/manifest/manifest-intro.html/>.
- [2] . Bugs that leads to denial of service attack. Available at: <http://www.ocert.org/advisories/ocert-2009-014.html>.
- [3] Homepage: Dalvik Virtual Machine. Available at: <http://www.dalvikvm.com/>.
- [4] . Patch for android security flaw released by google and t-mobile. Available at: https://www.isecpartners.com/files/iSEC_Securing_Android_Apps.pdf.
- [5] . Patch for android security flaw released by google and t-mobile. Available at: <http://cyberinsecure.com/patch-for-android-security-flaw-released-bygoogle-and-t-mobile/>.
- [6] . Patch for dalvik api dos. Available at: <http://www.ocert.org/advisories/ocert-2009-014.html>.
- [7] . Security hole in web browser. Available at: <http://mobile.slashdot.org/article>.
- [8] . what is android. Available at: <http://developer.android.com/guide/basics/what-is-android.html>.
- [9] Amit Kumar Saha. Introandroiddevnetbeans. Available at: <http://wiki.netbeans.org/IntroAndroidDevNetBeans>.
- [10] Android Developer. Android security and permissions. Available at: <http://developer.android.com/guide/topics/security/security.html>.
- [11] Hatem Ben Yacoub. Qualcomm demonstrates android on snapdragon platform. Available at: <http://openhandsetmagazine.com/tag/qualcomm/>.

- [12] Lior Baruch. Enhanced data gsm environment. Available at: http://searchmobilecomputing.techtarget.com/sDefinition/0,,sid40_gci213691,00.html.
- [13] Machigar Ongtang, Stephen McLaughlin, William Enck and Patrick McDaniel. Semantically rich application-centric security in android. Available at: www.patrickmcdaniel.org/pubs/acsac09a.pdf.
- [14] Marguerite Reardon. Vulnerability to image processing libraries. Available at: <http://www.builder.au.com.au/news/soa/Security-flaws-uneearthed-in-Google-s-Android/0,339028227,339286533,00.htm>.
- [15] Y. Shi, K. Casey, M.A. Ertl, and D. Gregg. Virtual machine showdown: stack versus registers. *ACM Transactions on Architecture and Code Optimization (TACO)*, 4(4):2, 2008.
- [16] Sohail Khan, Shehryar Khan, Syed Hammad Khalid Banuri. Technical report on analysis of dalvik virtual machine class path library. Available at: <http://imsciences.edu.pk/serg/projects/easip/resources/>.
- [17] Wook Shin, Shinsaku Kiyomoto, Kazuhide Fukushima, and Toshiaki Tanaka. Towards formal analysis of the permission-based security model for android. Available at: www.portal.acm.org/citation.cfm?id=1636735.