A Gentle But Effective Introduction

Android

Lars Celander

About this book

© 2013 Lars Celander

First published 2013-12-04.

Table of Contents

1 Introduction....5
2 Components....6
Application ("App")....6
App Manifest....6
Activity....6
Fragment....6
Service....6
Bound Service....7
BroadcastReceiver....7
ContentProvider....7
Intent....8
IntentFilter....8
Widget....8
3 App Architecture....9
App File Structure....9
Model-View-Controller (MVC)....9
Concurrency....9
Cloud Services....9
Google Cloud Messaging....9
4 Storage....11
Shared Preferences....11
Tables....11
Local Files....11
External Storage....11
Exposing Stored Data to Other Applications....12
Cloud....12
5 UI Architecture....13
Activities And The Back Stack....13
Tasks....13
Multitasking UI....13
Widgets....13
External Screen As Repeat Display....13
External Screen As Secondary display....14
External Screen Using Chromecast....14
6 Scalable UI....15
Screen Resolution....15
Screen Orientation....15
Screen Size....15
Screen Density....15
Mouse Cursor Bitmap....16
Status and Title Bar....16
Screen Coordinates....16
7 USB Peripherals....18
USB Device....18
USB Host....18
USB Device....18
Open Accessory....19
Open Accessory - Reference Implementation ('ADK')....19

MTP and PTP....19
Mobile High-Definition Link (MHL)....20
UART Peripherals....20
3.5mm Peripherals....20
8 Bluetooth Peripherals....21
Bluetooth Stack....21
Bluetooth Basics....21
Profiles Supported....21
Audio Streams....21
Reversing Roles....22
Bluetooth LE (Low Energy)....22
9 WiFi Peripherals....23
WiFi vs. Bluetooth....23
WiFi Direct....23
Miracast....23
Apple AirPlay....24
Google Chromecast....24
10 NFC Peripherals....25
11 Behind The Kimono....26
Processes and threads....26
Memory Management....26
Crashes....26
Rooting....26
Boot Loader....27
Custom ROM....27
12 Going Native....28
What It Is....28
Usage....28
Libraries Supported (as of Android 4.3)....28
Access to Underlying Hardware....28
ARM and Intel CPU....29
13 Who Owns What....30
Players and Drivers....30
Open Headset Alliance (OHA)....30
Trademark....30
Android Open Source Project (AOSP)....31
Compatibility....31
Google Services....31
Google Apps....32
Google Play Services....32
How Open Is Android?....33
CyanogenMod....34
14 Future of Android....35
Mobiles....35
Desktop....35
Embedded....36
Modular Mobiles....36
Android OS vs Chrome OS....36

1 Introduction

This is *not* your standard text on how to build Android applications. The intended audience is *not* the programming student.

The intended audience of this book is the 'solution designer' needing to know what Android is all about, what can be done and what can't be done, and how about building a well-architected Android-based solution.

There is a lot of standard material that is *not* is this book. Conversely, there are some fairly detailed discussions in this book on the edges of Android and of what Android can do.

2 Components

Application ("App")

An application is something that has at least one activity and is bundled as an .apk.

App Manifest

An AndroidManifest.xml file is part of the .apk file bundle. It lists all the components (activities, services, receivers, providers etc.) and various properties these have such as intent filters, affinities, launch mode and stack clearing attributes.

The manifest file is also used to indicate libraries needed to be linked to and to identify any permissions the application expects to be granted.

Activity

Activities contain the UI parts.

Activities are started, paused, resumed and stopped by the user and sometimes by the system. From the viewpoint of the application, lifetime is uncertain.

Fragment

Fragments are incomplete but re-usable activities that can be embedded in true activities. They can be seen as sub-activities. They are used for scalable UI construction where parts of the UI is added and removed as necessary while the app is running.

Service

Services implement background processes. They are intended to run for extended periods.

Has no UI and lifecycle is independent of UI. Started with an intent message sent to it. May send and receive intents while running. A service is stopped either by own command or by call from another component. If not stopped, it will keep running even if the component that started it is long gone.

A service can be started by any component. As long as it is not declared private in the manifest, it can also be started from within another application.

Services run in the main thread, just like activities. If you have a 'long running task' in the sense of something that is CPU intensive or otherwise blocking, like a network access, and you don't want to run the risk of stalling the UI, then what you need is a new thread, not a service.

Bound Service

This is a form of a service with which you can interact with in more powerful ways, bypassing the limitations involved with using intents.

Only within an application, not between applications (if you need that use Content Provider).

This interface is described using Android Interface Definition Language (AIDL). A Java stubs file is created automatically from an AIDL file.

When a bound service is no longer bound to anything, it is killed.

BroadcastReceiver

Broadcast receivers are objects that implement the BroadcastReceivers interface, i.e. just the onReceive() method. They are created by the system when a suitable intent has been recognized. They are garbage collected by the system soon as possible after it has finished receiving.

Intent broadcasts can come from the system, from other applications or from own applications. Broadcast receivers have no UI but can start activities or use a Notification Manager.

Broadcast receivers can also be implemented inside a longer lived component, e.g. an Activity. In this case the receiver needs to be registered (in the onResume() method) and unregistered (in the onPause() method).

ContentProvider

Content providers serve as a standard interface to various data sources. Provides data as tables, BLOB:s or files. Tables are not necessarily provided by SQLite, any RDB might be used. Content format published in a Contract.

Intended for sharing data with other applications, overkill for use within an application. Solves interprocess communications and security issues.

It is possible to hook monitoring mechanisms into content providers so that applications can be notified when data changes.

Intent

Intent objects are used to send messages to/from activities, services and receivers (but not content providers). Intents are used both within an application and between applications.

Intents are similar to e-mail. They are asynchronous in that you send them and after having sent one, you can immediately go on with the next task, without waiting for a reply.

Intents contain passive data:

Action	string (may be pre-defined)
Data	URI + MIME type
Category	string (may be pre-defined)
Extras	key-value pairs
Flags	flags (all defined in the Intent class)

Intents can have their targets set directly or indirectly. You set it directly by naming the target class. Class names can vary so this should be limited to within your own application.

Targets can also be set indirectly by specifying action, MIME type and category. Any targets that meet these criteria will receive the intent. Each component have these parameters set in the parameter file in the <intent-filter> element.

An intent may include a reference to a data source. Optionally an intent may also include extras in the form of a Bundle (set of name:value pairs).

IntentFilter

Used for indirect targeting of an Intent. Says which Intents can be received by a component. Owned by an activity, service or broadcast receiver.

Listed in the manifest and may have a label.

Widget

Typically an application that sits on the Home screen and displays periodically updated information. Can be embedded in any application, not just the Home application.

Has its own set of files. Is a BroadcastReceiver and has a Layout, somewhat like a View.

3 App Architecture

App File Structure

Apps are packaged as .apk files, in reality a standardised file structure as a zip file.

Model-View-Controller (MVC)

MVC is about partitioning an application into layers and having done so, being able to change one of the layers with minimal impact others. It is about reducing dependencies and the cost of changes. To an extent it is also about managing complexity by splitting complexity into more manageable parts. In current parlance, MVC is called a 'design pattern'.

Android enables MVC-style partitioning but does not enforce it. The programmer will be able to implement that architecture. There are frameworks and libraries available that facilitates building applications following this and other design patterns.

Concurrency

Standard concepts well supported by Android. Process and threads are there, as are locks and synchronization. Intents, Services and Content Providers provide higher level ways of building concurrent apps.

Then there are frameworks like Akka that makes all objects behave asynchronously.

Don't block UI thread. For example, don't put networking stuff in an Activity, create worker threads to do that (there are then a few ways to get back data into the UI thread). The Volley framework provides east-to-use facilities to do this.

Cloud Services

Latency about 0.6-1.0 seconds for mobile networks.

Google Cloud Messaging

This is a service provided by Google and part of Google Play Services. Used to send messages to an app on Android devices from a 3rd party server (typically provided by the creator of the app). Google handles the queueing and storing of messages until the device

is ready to receive them, for example if the device is not accessible for some reason.

Apps have an ID and the server has an ID and each has a unique ID that is part of the Android platform. These ID:s are used for the messaging and is done without user interaction, it's for example not necessary for the user to deal with an account on a server somewhere. On the other hand, the concept of groups of users is not available and has to be constructed and managed by the app and server.

Implemented using HTTP or XMPP. Message payload is limited to 4 kB. If XMPP is used then messages can also be sent upstream, that is from the app to the server (this feature uses the Cloud Connection Service and uses a *persistent* XMPP connection for speed).

The app does not have to be running, intents can be used to wake it up. Requires Google Play Store to be installed on the device but does *not* require a Google account on the device (for devices running Android OS 4.0.4 or later).

4 Storage

Shared Preferences

Shared Preferences is a lightweight method to store key-value pairs of primitive types.

They can named. If named they are available to other components in the same application, if not named they are local to one component.

The corresponding files used for actual storage are stored in that applications local directory. Removed when the app is uninstalled.

Tables

Tables, in the relational database sense, can be stored using SQLite.

Each table is named and is available to other components within that application.

The corresponding files used for actual storage are stored in that applications local directory. Removed when the app is uninstalled.

Local Files

These are stored in the local directory of that application. Removed when the application is uninstalled.

Static files to be used by an application, for example bitmaps, can be stored in the res/raw/ directory and are then available from within that application.

External Storage

This is an area that any application can access. It is this area that shows up when you connect an Android device to a PC. The Media Scanner looks in certain standard directories in this area:

Music/
Podcasts/
Alarms/
Notifications/
Pictures/

Movies/
Download/

This area used to be the same as the removable SD Card. On modern phones it might or might not be implemented as such. Might be in the form of a reserved area of local flash memory. Might not be present at all, for example the SD Card having been removed.

Exposing Stored Data to Other Applications

Shared Preferences, Tables and local files can only be made available to other applications using Content Providers.

Files on External Storage are available for anyone to access.

Cloud

Finally, if connected to a network, it can be used for storage (remember, this is Google).

Nothing defined within Android OS itself beyond standard Java networking facilities. Implementation is completely up to the application.

Files can be stored using standards based protocols like ftp and WebDAV. Various libraries exists that do that handle the details (for example AndFtp).

Then there are the proprietary file storage alternatives like Dropbox and Google Drive. These define their own APIs, come with an SDK and are typically built in top of http.

Backend as a Service (BaaS) involves having a database in the cloud. These can be with many types of semantics in how they look at data and how it's organized. These services typically also comes with some data management and viewing tools that run in a standard browser. SDKs for not only Android but iOS and JavaScript as well, with a ReST-based API for simpler tasks.

Data is stored as the usual Strings, Dates, Booleans, Numbers, GeoPoints and Binary. Relational data model as rows and columns. Queries are available, for example on radius from a GeoPoint.

Drawbacks include vendor lock-in and data ownership issues. Pricing model is often freemium. Examples of BaaS backends offered are Parse, CloudMine, Stackmob, Kinvey and FatFractal.

More standardized and vendor independent ways of storing objects are XML, JSON and native Java object serialization.

5 UI Architecture

Activities And The Back Stack

Has it's roots in small screens where the basic thinking that an activity owns the whole screen. Activities are stacked and the recalled.

This set of activities is the called a 'stack' with the originating activity called the root activity.

Tasks

The task concept does not have a strict definition, the term is used to denote a sequence of activities the user goes through, activities which might not necessarily all be part of the same app.

Nevertheless, there is a concept called affinities. The activities within a task move together and can be rearranged by the 'affinities' defined in the root activity.

Multitasking UI

As screens have grown and system matures, we now have a more sophisticated model.

Activity resumed (i.e running), it has focus, it owns the display,. Activity is paused, still visible beneath Activity that has focus. Activity stopped is not visible (disconnected from window manager).

Widgets

Size and functionality limitations?

External Screen As Repeat Display

This typically means repeating what's on the screen of the device onto something like a 24" screen. Yes, everything is going to look a bit coarse as layouts and drawables are intended for the much smaller screen of the device.

In this situation it's nice to have the buttons on screen and not as physical, this way they will show up on the big screen and can be clicked.

External Screen As Secondary display

Hardware permitting, Android allows any number of logical displays and these displays are available concurrently.

You can have separate content on each display and UI events are handled normally, including mouse and keyboard events.

Screen resolution is known but screen size and screen density is not. Layouts and drawables should be created in pixel units, avoiding dp and sp units.

These displays are handled by Display and DisplayManager classes.

The Presentation class, a subclass of Dialog class, is used to display something on a display different from the default display pointed to by default Context.

External Screen Using Chromecast

With this set-up you can send the contents of a Chrome tab to the external screen. That content will have to be constructed as a HTML web page. Typical usage is to display something on a dumb TV so don't expect to have any interactivity. Obviously, none of the usual Android functionality will be available.

Chromecast uses WiFi so the bandwidth is limited to about 20 Mbit/s, which should be compared to 18 Gbit/s for HDMI 2.0, which is why only web content or highly compressed video is sent instead of the raw video signal.

6 Scalable UI

Screen Resolution

Screen resolution is given in pixels. A very popular resolution these days is 1920x1080 pixels but older devices might have as little as 480x320 pixels.

Screen Orientation

Can be either portrait or landscape. Different layouts can be created for these orientations.

Screen Size

Screens come in different sizes. Layouts can be created for certain pre-defined screen sizes, used automatically. The pre-defined screen sizes are as follows:

small	about 2 - 3.5" and at least 426x320 dp
medium	about 3.2 - 4.5" and at least 470x320 dp
large	about 4.2 - 7.1" and at least 640x480 dp
xlarge	about 7 - 12" and at least 960x720 dp

Screen size and orientation is handled by having separate layouts defined for each combination of orientation and size. The system selects which one to use for an application at any one time.

Screen Density

A screen has a certain density expressed in dots per inch, dpi. The dpi value for internal screen hardcoded into properties file and cannot be changed.

Densities are grouped into pre-defined categories:

ldpi	around 120		
mdpi	around 160		default
(tvdpi)	around 216	Nexus 7 is 216	typical for 720p TV
hdpi	around 240		
xhdpi	around 320	Galaxy Nexus is 316	typical for 1080p TV
xxhdpi	around 480		

Screen density does not affect the layout used, it only affects drawables. An app is

expected to provide different drawables for each density.

Screen density indirectly affects layouts as these typically are expressed in density-independent pixel units:

dp density independent pixel, equals pixels at 160 dpi (fewer pixels for lower dpi etc)
sp like dp but scaled by font size user has selected, normally used by text

Icons are obviously drawables and should be provided in different pixel sizes. For comparison, standard icon size in Win XP is 32x32 and in Win 7 it's 48x48. On an mdpi (default) device the icon expected is 48x48, on an xhdpi device it's 96x96.

Mouse Cursor Bitmap

Displaying a small screen on a large screen results in a very large mouse cursor being shown which doesn't look natural. Unfortunately the mouse cursor is hardcoded as a system bitmap image and requires root to change. Some system builders provide a different image and/or allow different images and image sizes. Image can be enlarged as an accessibility issue but we want a smaller cursor. You can also do a hack of having a bitmap track and overlay the regular pointer as it moves around but this typically also means having a larger bitmap. The following Android Debugger commands will load a new cursor image onto a device:

"adb pull /system/framework/framework-res.apk c:\android\framework-res.apk"
open archive
replace "Res/drawable-mdpi/pointer_arrow" (with a .PNG extension)
close archive
"adb remount" makes the system folder writable.
"adb push c:\android\framework-res.apk /system/framework/framework-res.apk"
after the file has finished transferring, you may see a force-close error
hard reboot

Status and Title Bar

Status bar and Title bar both have a height of 25px. This might change. Height and font size etc are not programmable but both bars can be hidden (separately) to increase available screen area.

Screen coordinates and measured screen size are exclusive of the part of the physical screen occupied by Status and Title bars.

Screen Coordinates

The x coordinate is always horizontal and y is always vertical, irrespective of what portrait/landscape mode the phone is in. From a programming perspective, orientation change only means that the available width and height on the screen has changed. Changing screen orientation entails destroying and re-creating all activities, however.

A View knows only about the screen real estate is has been given by the window manager and knows nothing about actual screen size, or thus orientation, except by asking the window manager.

On a platform-wide basis, you can only enable/disable auto-rotation (done under Settings menu or programmatically in android.provider.Settings, might in special cases be forced to enabled/disabled by the underlying implementation). There is no way to lock the whole platform to either landscape or portrait.

The standard Home application is often locked to portrait, depending on how it was designed. There are many “Home” applications available and they can have any behaviour.

7 USB Peripherals

USB Device

The micro USB port on an Android device is normally used to have the device act as a peripheral to for example a PC.

Many times, this is not we want. We want the Android device to be host to peripherals, for example connecting keyboard and mouse to a device.

USB Host

Android device with 3.1 or later are capable of acting as standard USB host including powering the USB bus and enumerating devices.

The Android host should be able to provide a minimum of 100mA but most peripherals will consume much less.

An OTG Cable is used to tell an Android device that is now a USB host, as opposed to its original role as a USB device. The difference to a normal cable is that pin 5 is now connected to ground and not free-floating (as in a regular non-OTG cable).

For some common devices (mice keyboards, thumb drives etc) support is built-in to the OS and no coding is required. Typically mice and keyboard works fine but thumb drives are read only (without rooting device and installing e.g. StickMount). Both my Galaxy Nexus and Nexus 7 immediately accepted a mouse, neither reacted to a thumb drive (will probably require Nexus Media Importer). Partial functionality with a gamepad and a joystick (no cursor but keys seemed to work, in some way ...).

For other devices, the Android host can interface using custom code using the standard API's provided by Android OS in 3.1 and later.

USB Device

Acts as USB storage device or a communications device, whatever is implemented in the OS. No coding involved to implement these and no API published.

Whatever host the Android device is connected to, provides power and performs enumeration etc. Regular non-OTG cable used.

Open Accessory

An API and a protocol used to interface with add-on hardware. Currently uses USB and Bluetooth as the transport layers.

Many Android devices are only capable of acting as an USB device. Open Accessory allows these to interface with purpose-built Android add-on hardware by adding the accessory mode as an extension to USB device mode.

The accessory handles the USB functions normally associated with being an USB host of powering the USB bus (min 100mA, max 500mA) and enumerating USB devices.

Many accessories don't want to bother with Open Accessory, they just want to be a USB device, able to connect to any type of USB host. As Android devices more and more develops into being a full-fledged USB hosts, Open Accessory will become unnecessary.

Open Accessory - Reference Implementation ('ADK')

Google publishes reference implementations of Open Accessory called an Android Development Kit (ADK).

Exists in different versions with both hardware and software fully provided. Ready-built versions provided to all Google I/O attendees but can't be otherwise bought (except obviously on eBay). Both 2011 and 2012 versions uses Arduino connected over USB with the 2012 adding Bluetooth.

Other manufacturers produce similar boards that also implement the Open Accessory protocol.

MTP and PTP

As of Android 3.0, USM Mass Storage (USM) is no longer supported. This means that when you connect an Android to a PC, it no longer appears as a USB memory stick.

With the PC no longer mounting anything as a drive, the Android device no longer has to unmount anything. The PC and the Android OS can now both access the memory, using a suitable protocol.

The protocols selected are PTP (Picture Transfer Protocol) and MTP (Multimedia Transfer Protocol). PTP is commonly used to download pictures from cameras. MTP is an extension of PTP. CIFS or SMB could have been used but for various reasons are not.

Using MTP, the SD card can now be formatted in any way. The formatting will not be visible to the outside, it will be handled by the MTP driver. Which brings us to the topic of SD card formats. By far the most common is FAT32 on SDHC cards. SDHC works up to 32 GB. For the larger SDXC cards, exFAT is specified to be used which takes us to 2 TB. FAT32 also works up to 2 TB albeit with a max file size of 4GB. The problem with exFAT is that it is proprietary to Microsoft. For licensing reasons, exFAT is not supported by the

Linux kernel but can be supported by a user-space driver. However that is done, all or most modern Android smartphones appear to support exFAT. All this is only really important of you intend to move the SD card to some other machine, if it is going to spend its life inside the Android devoice, it can be formatted with any of the standard file systems in Android.

Mobile High-Definition Link (MHL)

Allows an device to drive an external display. Also allows the display to send commands back to the device. Requires support from both the device and the display.

Intended usage scenarios include a smartphone driving and interacting with a TV, a large desktop monitor or an in-dash display on a car.

MHL-compatible displays can be driven using a simple passive micro-USB to HDMI cable with the display supplying power to the device through that same cable.

Non-compatible displays can be driven through an MHL adapter which has electronics and needs external power, typically using a micro-USB input connector.

UART Peripherals

Android devices don't have a UART the way for example iPhones have. This can make it harder to build an Android peripheral. The go-to solution here is the chips and cables offered by FTDI (http://www.ftdichip.com/Android.htm). Sample source code is available.

3.5mm Peripherals

Yes, we can use the audio in/out connector to attach peripherals. This is a primitive and slow connection but it is doable. What's happening is that audio tones are generated and decoded.

On the peripheral side we can control the hardware and build whatever we want. On the Android side, we have to make do with what is available on the platform.

8 Bluetooth Peripherals

Bluetooth Stack

The Bluetooth software stack in Android used to be based on BlueZ project in Linux. This stack was freely available but evolving only slowly and vendors frequently added significant chunks of functionality. For example, Samsung replaced it with the Broadcom BlueDroid they had licensed.

As of Android 4.2 the Broadcom BlueDroid stack was released under an Apache license is now integrated into AOSP, replacing BlueZ as the standard stack. The old BlueZ stack is still available in the source tree.

Bluetooth Basics

Pairing is a procedure to tell devices that they may connect with each other. May be done in many ways and is only done once. Does necessarily mean that they have actually connected.

An Android device can connect to many Bluetooth devices at the same time, for example both a keyboard and a mouse.

Profiles Supported

HFP, HSP, PBAP, A2DP and AVRCP are supported. Android 2.1 added RFCOMM and SPP as well as OPP. RFCOMM allows generic communication and SPP replaces serial communication in the old RS-232 style.

Audio Streams

The phone settings only deal with identities and pairings, i.e. the hardware side. No mention of anything related to routing or routing logic, presumably this is left to the applications.

There are two audio streams mentioned within the Android SDK named A2DP and SCO (with SCO intended for HSP and HFP). A2DP is high quality stereo audio and SCO is 8 bit 8 kHz mono audio. The phone uses SCO and if an HSP or HFP device is paired and connected, will use it automatically. The media player and other applications use A2DP

which is routed to the speaker by default, to a wired headphone if connected or to a Bluetooth-connected headphones if found. Headsets that can switch between being using HSP/HFP and A2DP are supported.

The AudioManager is treated the same as a sensor which means that it applies to the whole phone and is not limited to one application and its VM instance. AudioManager has settings for directing both A2DP and SCO audio streams to Bluetooth. Before Android 2.2 *apps* where unable to send audio to SCO, i.e. to HSP/HFP devices.

Reversing Roles

An Android is built to be a phone and has the Bluetooth personality for that. In embedded applications, the Android device being built might be a phone peripheral and not a phone.

This means that the built-in stack has to be re-configured, modified or replaced.

After this modification to Bluetooth stack functionality, compatibility with standard Android apps might be lost with regards to Bluetooth phone behaviour.

Bluetooth LE (Low Energy)

Uses a different stack, simpler and more energy efficient. Originally not Bluetooth but now included in Bluetooth 4.0. Supported since Android 4.3.

9 WiFi Peripherals

WiFi vs. Bluetooth

WiFi is basically Ethernet over the air. It may be noted that stock Android does not support wired Ethernet. WiFi is strictly a transport mechanism.

Like Ethernet, WiFi is peer-to-peer. There is no master and collisions are handled in a way similar to Ethernet. This is in contrast to Bluetooth that employs a master/slave architectures.

WiFi has much higher speed and somewhat longer range then Bluetooth, up to about 30 meters instead of 10, but also has a higher power consumption.

WiFi Direct

This is an extension of traditional WiFi in that no access point is needed, communication can be directly from one device to another. Pairing, in the Bluetooth sense, is done by NFC or by Bluetooth or by buttons.

Supported by Android 4.0 and later, recent versions of Linux. Windows 8 and Windows Phone 8 has support for peripherals using WiFi Direct but has not made any API available for developers. Not supported by Apple (uses its own Airplay instead).

Can be used for anything, for example to hook up peripherals like printers, mice, keyboards, remote controls, headsets, speakers and/or a display.

No semantics have been defined. There is nothing like the Bluetooth profiles. All we have is basic, if flexible, networking. You can build whatever you want but you have to do it yourself, on both sides of the WiFi link.

Miracast

A specification in how to stream audio and video to a display using WiFi Direct. Created by WiFi Alliance and defines the semantics of how to use WiFi Direct for that task. Roughly equivalent to a Bluetooth profile. As usual for a new standard, it takes some time for the kinks to be worked. At this stage, Miracast has relatively poor reliability.

Apple AirPlay

Another streaming/control protocol, this time developed by Apple. Works very well but is proprietary to Apple and limited to Apple or licensed products. Not compatible with WiFi Direct and competes with Miracast.

Google Chromecast

With Miracast stuck with interoperability issues and Apple steaming full speed ahead with its own protocols, Google decided to go with its own proprietary combination of protocols and devices.

Built on top of WiFi Direct and used to stream from the router to the display with the device only acting as controller (or remote control), alternatively streaming from the device to the display.

Implemented as a small dongle that fits in an HDMI input on a display. Currently no support for MHL so any TV remote cannot be used, the device is used instead to control dongle and streaming. Devices supported are Androids and anything with a Chrome browser. An SDK is available.

The Chromecast dongle is a very strategic product for Google, granting Google access to for example the living room TV. Google now has a complete eco-system with both cloud services and the mobile device connected to any large display with all parts of the system under Google control, allowing Google to build anything they want without having to wait for anybody else. They have good business reasons for pricing it so low.

It can be expected that future version of Android will allow any HDMI-equipped display to act as an external screen for Android apps.

10 NFC Peripherals

NFC is a wireless communications technology is intentionally limited short distances. Thus the name Near Field Communications.

Compared to Bluetooth, speeds are about an order of magnitude to slower. Power consumption is correspondingly much lower. It is even possible to communicate with completely unpowered tags.

A big advantage with NFC is that no pairing is required. The communications link is set up as soon as the two devices are brought on close proximity to each other and will cease as soon as they are separated by more than a few centimetres.

The whole field of NFC is well standardised with mature specifications.

Android supports NFC and most modern phones have the necessary hardware. This whole field will see an explosion of use in the coming years.

Most popular current use is for payments of small amounts. NFC is also used as a means of pairing Bluetooth and WiFi devices.

11 Behind The Kimono

Processes and threads

By default each application lives in its own Linux process with its own copy of the VM and with its own Linux user ID.

However, components may be allocated their own processes if this is required for some reason. This is done in the manifest.

Within the application process, there is typically one thread called the 'main' thread. This thread handles all UI events and is therefore sometimes also called the UI thread. Additional threads may be created as required by the application, for example to handle slow and blocking taska such as networking.

The Android UI toolkit is *not* threadsafe. Care has to be taken when calling UI stuff from a worker thread (for example using AsyncTask).

Memory Management

Each application is given a fixed amount of memory. This amount depends on screen size and screen density, varying between 16 MB and 128 MB.

Crashes

ANR (Application Not Responding), when an app has not calls from the system for a duration of longer than 5 seconds. Programmer error such as endless loop, or some resource is not available.

We also have true crashes, usually some unforeseen condition that has caused an error that the programmer has failed to tell the software how to handle.

Rooting

Rooting means getting administrator privileges to the user, allowing the user full access to everything on the device.

Devices are not meant to be rooted. Rooting usually means exploiting some security weakness in the device.

If the user knows what he/she is doing, it will allow the user to for example install a custom version of Android. Any mistake and your device can become corrupted and unusable.

Boot Loader

The boot loader is a piece of software that handles the boot procedure, i.e. loading the Android installation after the device has been turned off.

Recovery Mode is a special troubleshooting mode that the device can boot into, typically when the regular Android installation for some reason has become unusable. The main function of the recovery mode is to re-install the regular Android installation.

Pretty much all devices come with a recovery mode. You boot into it by some key combination, specific for that device. Different devices have different functionality and usefulness in their recover mode software so you might want to install a fancier recovery mode, particularly if you are going to replace the Android installation that came with the device with a custom installation.

Many boot loaders only allow booting into what the device manufacturers intended, the recovery mode and the regular mode. This known as a locked boot loader. Other devices have a boot loader that allows booting into whatever the user has installed. This is known as an unlocked boot loader.

Rooting a device may make it possible for the user to unlock a boot loader (or replace with an unlocked version).

Whether or not a phone has a locked boot loader is determined by the manufacturer, it's not a feature of Android.

Custom ROM

The acronym "ROM" stands for "Read Only Memory" and is the memory used to store the Android installation. It's called the ROM because it's not intended to be written to by the user. But it can be. This is known as installing a "custom ROM" and is equivalent to installing a version of Android built by someone else that the original device manufacturer.

On a standard smartphone, a custom ROM is installed by performing a "recovery" into a file that the user has downloaded and transferred to the SD Card. This usually requires having an unlocked boot loader which in turn might require having rooted the device.

12 Going Native

What It Is

A set of standard C libraries are supported and guaranteed to be stable in future versions of the Android platform.

Typically used to enable the reuse of existing libraries of C/C++ code, for example for computationally intensive tasks like physics in gaming or image processing.

Produces library which can be packaged with standard apk files, can thus be packaged and sold as normal applications (no need for a developer or hacked version phone).

Usage

All accesses to C/C++ code are done using JNI from inside the Android application. JNI headers are provided.

Libraries Supported (as of Android 4.3)

libc (C library)
libm (math library)
Minimal support for C++
JNI interface
liblog (Android logging)
libz (Zlib compression)
libdl (dynamic linker)
libGLESv1_CM.so (OpenGL ES 1.0)
libGLESv2.so (OpenGL ES 2.0)
libGLESv3.so (OpenGL ES 3.0)
libEGL.so (native OpenGL surface management)
libjnigraphics.so
libOpenSLES.so (OpenSL ES 1.0.1 audio support)
libOpenMAXAL.so (OpenMAX AL 1.0.1 support)
libandroid.so (native Android activity support)

Access to Underlying Hardware

Possible but is not guaranteed to be stable and involves testing on a phone by phone and on a platform version basis.

ARM and Intel CPU

Using bytecode and a VM, it doesn't matter whether the hardware is ARM-based or Intel-based. This breaks down when using native code. When compiling you must state which architecture you are using. In practice this is not a big problem since ARM dominates but this might change in the future.

13 Who Owns What

Players and Drivers

Google sells services. To enable more services being sold, Google wants clients and networking to be cheap. It is therefore in Googles interest to have a free operating system and oodles of low cost smartphones. This is why Android exists.

Phone manufacturers don't have a problem with this. They typically don't sell services. For them it's a good partnership. They get a good operating system for free and are allowed to tinker with it and add stuff. The ecosystem is in place and has long since reached critical mass. They can all compete on what they do best, manufacture phones.

Having reached dominance in the mobile phone space, Android is now set to capture the embedded and desktop markets.

Open Headset Alliance (OHA)

A consortium of 84 companies to handle open standards for phone standards, notably Android OS. Set up 2007 by Google.

Doesn't seem to do much really. No press release has been issued since 2011 and those were only about new members. Apparently member companies do provide useful contributions to the Linux kernel and does so in the open, just like any other contributor.

Principal function seems to be of a business nature, members are not allowed to produce non-compatible versions of Android. This is clearly anti-competitive. Since most players in the business are members, for example OEM:s, it can be difficult to find any to work with if you want to be independent of Google.

It's not clear why anybody needs to be a member of OHA. Most of these agreements and contracts are secret and might only become public in the context of court cases.

Trademark

Android is a trademark owned by Google (and thus not by OHA). Google uses this ownership to enforce compatibility (see below).

Android Open Source Project (AOSP)

This is the core of Android as an open source project. Most of it is licensed under Apache License 2.0.

The Linux kernel is licensed separately under its own license (GPL v2). Android used to built on a forked version of Linux kernel but those changes have now been worked back into mainline Linux.

Much of the low-level firmware is proprietary software by chip set manufacturers, particularly radios that need to be licensed by government authorities for compliance. There is also considerable IP (Intellectual Property) involved in some of the firmware, for example the GPS receiver software. These are typically delivered as binaries and available for download free of charge within the AOSP. Google and OHA has little interest or involvement here.

AOSP is run by Google as a private project. Contributions are welcomed but everything is owned and controlled by Google. Google releases new versions of AOSP at times of Googles choosing.

Compatibility

Google publishes the Android Compatibility Definition Document (CDD) which lists software and hardware requirements for what it means to be a compatible Android device.

CDD is a highly detailed description of how an Android device must behave. Originally meant for phones but has been relaxed somewhat to allow non-phones to be compatible. For example, many sensors are listed as "SHOULD" which means that they must not necessarily be present.

There is very little of non-technical nature. The thrust of the document is to ensure that the device behaves as expected and that apps are run without problems. Nothing in the CDD is of a business nature, for example there is nothing that ties anything to any Google services.

The content of the CDD is owned and decided by Google. It is Google policy that non-compatible devices are not allowed to use the Android trademark and may not license Google apps.

In theory you can be compatible with the CDD without being a member of OHA. In practice it might not be that simple.

Google Services

Google markets several very popular services like YouTube, Google Maps and Google Search. These are the core of Googles business model.

There are 'Terms and Conditions' that apply to anyone wishing to access these services. These conditions are set by Google and can be whatever Google likes. These services are not open to everyone, they are not necessarily free and can be subject to arbitrary

limitations. Googles own apps can do whatever Google wants and allows in relation to services available from Google and these may or may not be used by anyone else.

One such app is the Google Play Store app. Only the Google Play Store app is allowed to access Google Play Store.

Beyond apps, most of these services are also accessible through browsers (to the extent that Google chooses to make them available through their web sites).

Selling an app on Google Play Store requires you to abide by a set of rules and recommendations for how apps are supposed to behave. These are all set by Google. Google does not have any problem with apps that compete with Googles own apps but competing apps can only access Googles own services to the degree that Google allows.

Google Apps

Most phones come with a set of apps by Google. These are not open source, they are proprietary to Google and are normally licensed from Google by device manufacturers. They are not mandatory, they are not part of the Android platform per se and can be removed.

The source code of AOSP can be downloaded, perhaps tweaked, and then built into a functioning device. This known as a 'custom ROM'. If you install a custom ROM on a device, the various Google apps are normally not included, not having been licensed by whoever created the custom ROM. This includes the Play Store app which means that you now have to get your apps from somewhere else. As always, there are various ways around this but this is the general principle of how things are intended to work.

Google Apps are licensed as a bundle. Google does not allow anyone to only license what it wants. This is particularly important in relation to Play Services.

Some manufacturers, for example Samsung, delivers their phones with a set of apps that do the same work as the Google apps. These tend to be seen by users as bloatware but are important for Samsung to reduce their dependence on Google. If Samsung wants to fork Android it will have to make do without Google apps and with all Samsung devices already having the necessary apps installed, Samsung can do just that.

Google Play Services

This is one of the Google apps. It is unique in that it's not really an app in the usual sense, it's an app that provide background services to other apps. For example, most Google apps require having Google Play Services installed. Other third party apps sometimes also elects to use Play Services, for example to access some Google service.

Play Services is updated by Google. It is not part of Android platform. Google keeps it automatically updated so the latest version is always installed. This effectively means that Google has created a its own platform for its own apps. Since it is automatically kept updated, there is essentially no fragmentation.

This is in marked difference to the underlying Android OS platform itself on a device which tends to stay on the Android version the device had when sold. That underlying platform is quite complex and time-consuming to update, for example involving regulatory compliance testing and certification. For the manufacturers, this is only cost-effective for recent devices which have sold in large volume. With a significant chunk of all devices in use remaining on older Android versions, we then have a real problem of platform fragmentation.

This is why recent Android OS updates have been fairly boring, it's mostly low-level stuff that is being updated. All the Google-specific goodies are now updated separately, decoupled from the Android OS updates.

Anyone can make the same. If you have a bunch of services you would like to make available in some way, you are free to do the same. Most phone manufacturers don't and are happy to have their customers use Google services or anyone else's.

The main problem with Play Services is that third party apps also use it. These apps are then tied to the Google version of Android, acting as a defence for Google against forks. The way that apps are more and more based on Play Services instead of only on standard Android gives Google more and more power over Android in general.

How Open Is Android?

As an operating system, Android is completely open. You can build whatever you want using the Android software stack. And people do so. It is rapidly becoming quite popular to build devices using this stack, available chips and available tools. The major exception to this openness is the closed firmware that ties into the hardware.

The key decision is whether or not to be compatible. If you are, you get access to Google Services including Play Store. If not, you don't have this access. For good reasons, as you presumably cannot run Android apps reliably. You also don't get to call your device an Android device. You might be able to that it is 'built on Android' but you can't say that it *is* an Android device.

With compatibility you can access Google services but you don't have to. You are still free to access whatever service you want and install whatever app you want from wherever you want.

If you are ready to build your own apps and services, for example an app store, then you can do exactly as you please.

Android is open but Google is not open. The openness of Android is real but should not necessarily be thought of as a business model or philosophy in itself, at the end of the day it's just a tool for Google. Which is fine. Google has not been very good at explaining this and you often criticism against Google for not being as open as someone would want but more often than not, this criticism is based on a poor understanding of how the Android eco-system is constructed.

Google can attempt close off Android by releasing new versions of Android only as closed source. Existing versions of Android will still be open source and that is what all the apps are using. What very likely will happen then is the creation of a new consortium of some

type with the intent of updating the platform retaining its open source nature, bypassing Google. All existing app will run on both platforms, as long as they don't use Play Services. What app developers will likely prefer, depends on the situation. It's basically a choice between the open version and the one that has access to Google services.

Yes, Google has considerable influence over the platform but only as long as everybody else allows them to. Google has essentially two trump cards. The first is the Android trademark but, at the end of the day, it's only a trademark. The second is the set of apps that depend on Play Services and by extension on Google services but that, at the end of the day, is what Android is all about, a conduit for Google services. It's not Android that matters, it's the services.

The old Windows monopoly will not be replaced by a new monopoly, that problem has been solved with AOSP. Unfortunately, we haven't really solved the problem as much as moved it. We now have a new de-facto monopoly in the area of services. Which is fine with the device manufacturers. Microsoft realizes this and had a choice between copying the business model of Google, giving the OS away for free and making money on services, or copying the business model of Apple or Amazon, having devices and services tightly integrated and either making money on the hardware and using services to drive hardware sales (Apple) or using cheap devices to sell content services (Amazon), both of which requires building both the devices and the services yourself. Device manufacturers are taking note and are being driven into the arms of Google, seen as the lesser of two evils.

CyanogenMod

Not everyone is happy with the Android version that came with the phone. Phone manufacturers need to differentiate themselves and are less keen on supporting older devices, preferring to focus on current and future models. So they modify it and then don't support it. Some phone owners prefer the latest version without the bloat.

This is where CyanogenMod ("CM") comes in. It's an aftermarket Android version replacing the version that came with the phone.

CM builds on stock Android as available from AOSP. The work they do is to essentially to integrate AOSP with phone hardware, the work normally done by the phone manufacturer. CM also provides some installation mechanism to replace the old Android version with their newer and better.

Nothing stops a manufacturer from going straight to CM, effectively having CM doing the integration of AOSP with it's own hardware. Cyanogen Inc. is a company that has been started to cater to various such business opportunities around CM, as well as to provide better support to CM users.

CM typically conforms to the Android compatibility definition although it might not have been officially been certified as such. CM might or might not include any apps from Google. CM is not a fork of Android, it's much more of a phone manufacturer in the role it plays with actual hardware built by someone else.

14 Future of Android

Mobiles

With Android enjoying a market share in excess of 80%, in a market where critical mass is everything, that battle is over and has been so for quite a while.

Apple, as a high end brand with focus on profit margin and will then strive for the largest possible market share without sacrificing profits. Their optimum market share is probably 10-20%. Anything less and volume becomes a problem, anything more can only be reached by slashing profits. The existence of Apple is actually useful for Android, it doesn't threaten Android and it relegates Windows, the real threat, to third place.

A resurgent Windows on phones is the only possible threat. This is not likely, much for the same reasons that Android thrashed Windows in the first place. Microsoft has to fundamentally change it's business model, which will take a lot of time and effort and money and courage and after having successfully executed that change, it's still going to be an uphill struggle.

Desktop

Very much work in progress against well-entrenched opposition in the form of Microsoft Windows.

The dominance of Windows in terms of market share is not going to change very rapidly. Windows is going to be with us for a long time. What might change rapidly, in case it hasn't happened already, is the perception of where the market is going. This can rapidly erode the market share for Windows for consumers and many business needs, effectively reducing Windows to the status of a platform needed for legacy applications.

Microsoft is going all-in on touch, aiming for one platform to rule everything, leveraging it's desktop monopoly to embrace and extend into the mobile space. This is a risky strategy. What might well happen is that major developers see that everything is going towards touch UI and having accepted that, then decides to develop their next apps for the OS that dominates the touch UI world which, to Microsofts horror, could well be seen to be Android and Microsoft would end up having handed over it's desktop monopoly to Android.

A less risky strategy would be to consciously have two platforms, one is traditional non-touch UI running on Intel, the other touch-based and running on ARM. This is what Apple is doing. Cross-functionality between the two eco-system can the be controlled using bundling or other licensing tricks on the app store (buy Office for Intel, get a free Office

for ARM etc.).

An Android desktop would currently be a phone connected to a standard 24” screen plus Bluetooth keyboard and mouse. This works but is poor use of the desktop monitor as apps are optimized for small screens (icons are now gigantic etc.). There are also limitations in the Windowing architecture optimized for small screens (only full-screen apps etc). An app can be built having different modes adapted for different screen sizes and densities but few apps do that and there are some remaining issues (mouse cursor size etc.). It would be fairly easy to remove these obstacles but Google seems to take it's time here. Android 5 is going to be interesting.

Mobiles currently lag behind desktop hardware in performance by a factor of about 3-5x. Desktop hardware is essentially stuck at 3-4 GHz and mobiles are able to gradually close this gap. Nothing also stops Android from running on Intel processors providing the same performance as todays desktop PC's but before that makes sense, Android needs to migrate to 64 bits architecture, which is in full swing.

Embedded

Again, work in progress. The market here is very fragmented and the underlying reasons for that are not going away.

The sheer size of the mobiles ecosystem means that are lots of powerful hardware available at low cost that supports Android. Being open source also helps. Expect this market to grow rapidly but in relative obscurity.

Modular Mobiles

This is new and radical idea sold on the advantage reducing waste. With something like that as the main selling point, it's not going to go anywhere.

Might get some traction as a platform for Androids featuring custom hardware capabilities but that market might simply not be there given that there are many other ways to connect add-ons to a mobile (for example USB, Bluetooth and WiFi).

Android OS vs Chrome OS

So what is the long term relationship between Android OS and Chrome OS, surely you are not going to have two overlapping operating systems? Good question. Google says nothing and actually makes quite a point of saying nothing.

One obvious way would be to allow Android applications to run inside a browser window much like the old applet concept. The big point here is that now an Android app could run on any platform to which Chrome browser has been ported. So far it hasn't happened. One obvious reason might be the somewhat silly situation where you run Chrome on Android running on Chrome etc, ad infinitum. Not something that looks sensible.

The direction Google seems to be going is enable Chrome apps to run inside the Chrome

browser on Android (and later the Chrome browser on iOS), i.e. Chrome runs on top of Android and not the other way around.