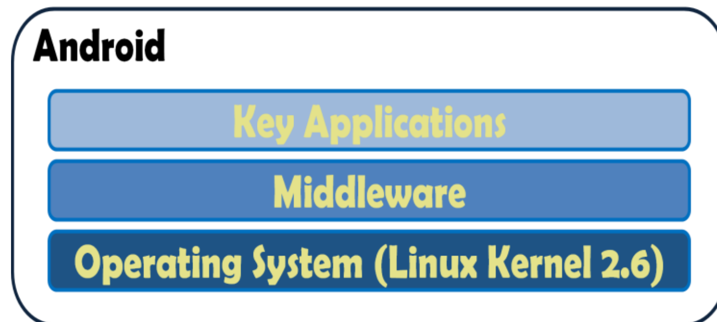
100% COURS
MCours.com
Cours et mémoires gratuits
mcourscom@gmail.com

Introduction

Android est une plateforme pour appareil mobile (téléphone, PDA, netbook, tablettes, etc). Elle est composée d'un système d'exploitation, de librairies "middleware", et d'un ensemble d'applications : un client mail, un navigateur, un calendrier, etc.

Android est basé sur un kernel linux. Les librairies "middleware" qui le compose sont écrite en C/C++. Le Framework est quant à lui écrit en java.

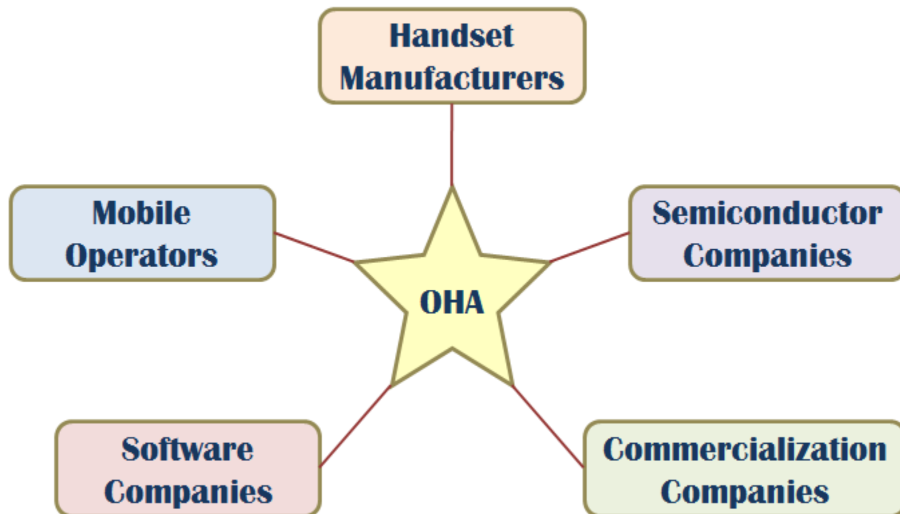


OHA

Android est développé par l'OHA (Open Hanset Alliance), une alliance internationale d'entreprises. Cette alliance se compose de compagnie ne faisant pas partie du même secteur.

Ainsi elle se compose :

- d'opérateur mobile (Vodafone, Telefonica, Telecom Italia, China Mobile, etc.)
- de fabricants de téléphone mobiles (Asus, HTC, LG, Motorola, etc.)
- de fabricants de semi conducteur (Intel, Nvidia, ARM, etc.)
- d'éditeurs logiciels (Ebay, Google, PacketVideo, etc.)
- de distributeurs (Aplix corporation, Borqs, TAT)



Aujourd'hui il y a 2 milliards de télévisions dans le monde. 1,5 milliard de personnes ont accès à internet. Mais près de 4 milliards de personnes ont un téléphone portable, ce qui fait que le téléphone portable est le produit connaissant le plus grand succès dans le monde. C'est pour cela que l'OHA s'est lancée sur le secteur du mobile. Ils espèrent fournir une plateforme mobile innovante et performante fournissant aux utilisateurs une nouvelle expérience d'utilisation de leur mobile.

Historique

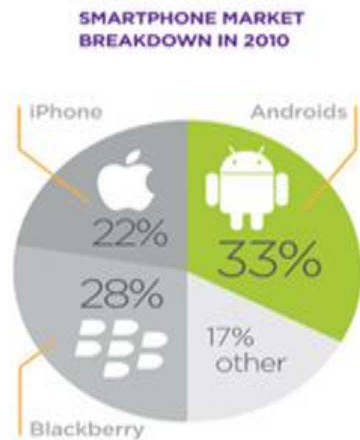
En juillet 2005, Google a acquit Android, Inc., une petite startup qui développait des applications pour téléphones mobiles .C'est à ce moment là que des rumeurs sur l'entrée de Google dans le secteur du mobile ont commencé. Mais personne n'était sur, dans quels marchés ils allaient se positionner.

Après ce rachat, à Google, une équipe dirigée par Andy Rubin, un ancien d'Android Inc, a commencé à travailler sur un système d'exploitation pour appareil mobile basé sur linux. Durant 2 ans, avant que l'OHA soit crée officiellement, un certain nombre de rumeurs ont circulé au sujet de Google. Il a été dit que Google développait des applications mobiles de son moteur de recherche, qu'ils développaient un nouveau téléphone mobile, etc.

En 2007, le 5 novembre, l'OHA a été officiellement annoncée, ainsi que son but. Développer des standards open source pour appareil mobile.

Le premier standard annoncé a été Android, une plateforme pour appareils mobiles basée sur un kernel linux 2.6.

En septembre 2008, la première version stable du SDK est sortie, à ce jour la dernière version est la 1.2.



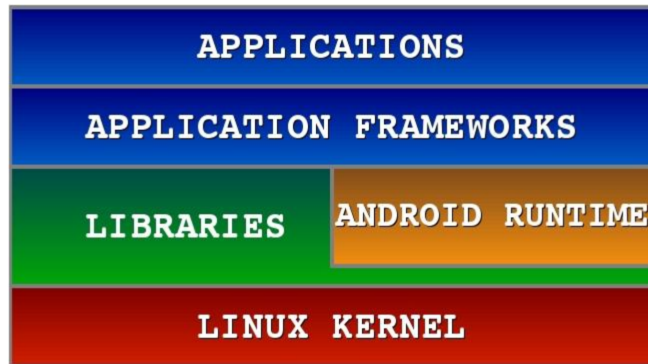
Caractéristiques :

- **Framework :** Framework Java pour le développement d'application pour la plateforme Android
- **Machine virtuelle Dalvik :** Machine virtuelle spécialement développée pour Android. Cette machine virtuelle permet d'exécuter les applications java développées avec le Framework.
- **Navigateur Web :** Navigateur web basé sur le moteur de rendu Webkit. WebKit est une bibliothèque logicielle permettant aux développeurs d'intégrer facilement un moteur de rendu de pages Web dans leurs logiciels.
- **Graphique :** Librairie graphique 2D, librairie graphique 3D basé sur OpenGL ES 1.0. Accélération matériel possible. qui définit une API multiplate-forme pour la conception d'applications générant des images 3D dérivée de la spécification OpenGL, sous une forme adaptée aux plateformes mobiles.
- **Stockage :** Base de données SQL : SQLite est utilisé pour le stockage des données.
- **Media :** Android supporte les formats audio/video/image suivants : MPEG4, H.264, MP3, AAC, AMR, JPG, PNG, GIF
- **Connectivité** : gsm, edge, 3G, bluetooth, wifi
- **Support Matériel :** Android est capable d'utiliser Camera, GPS, accéléromètre….
- **Environnement de développement :** Android possède un environnement de développement complet contenant : un

émulateur, un débuggeur, un analyseur de mémoires et de performances et un plugin eclipse.

Architecture du système Android

1- Vue générale :

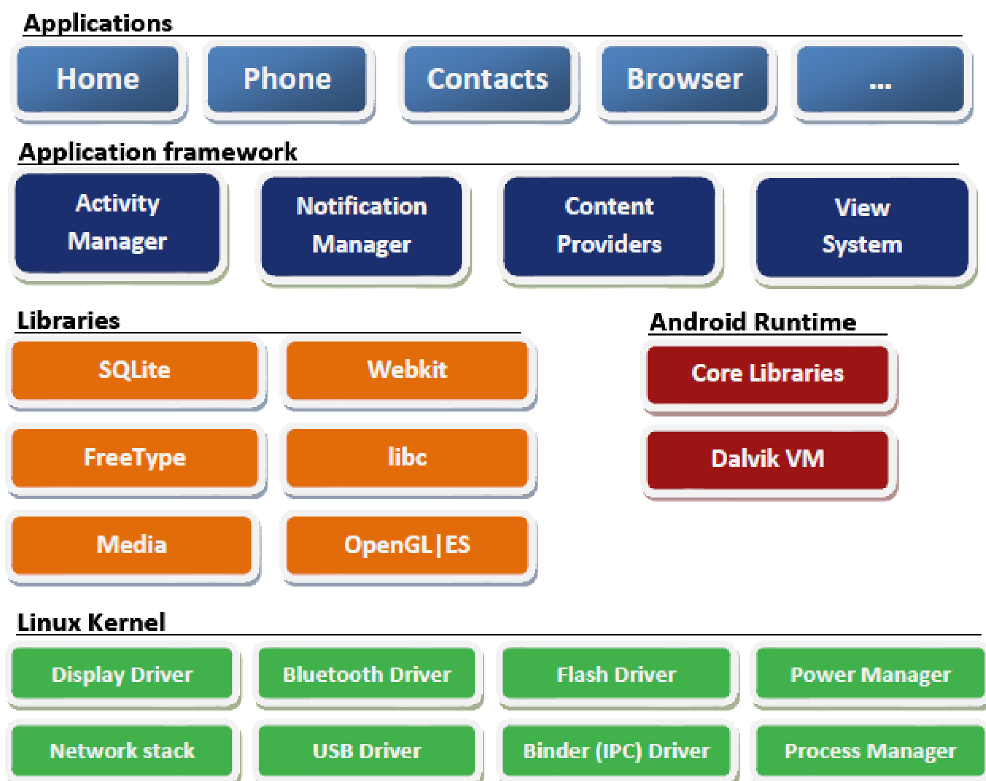


Android est basé sur un kernel linux 2.6.xx, au dessus du kernel il y a "le hardware abstraction layer" qui permet de séparer la plateforme logique du matériel.

Au dessus de cette couche d'abstraction on retrouve les librairies C/C++ utilisées par un certain nombre de composants du système Android.

Au dessus des librairies on retrouve l'Android Runtime, cette couche contient les librairies cœurs du Framework ainsi que la machine virtuelle exécutant les applications.

Au dessus la couche "Android Runtime" et des librairies cœurs on retrouve le Framework permettant au développeur de créer des applications. Enfin au dessus du Framework il y a les applications.



100% COURS
MCours.com
Cours et mémoires gratuits
mcourscom@gmail.com

2- Linux Kernel :

Android est basé sur un kernel linux 2.6 mais ce n'est pas linux. Il ne possède pas de système de fenêtrage natif (X window system), la glibc n'est pas supporté, Android utilise une libc personnalisée appelé Bionic libc.

Enfin Android utilise un kernel avec différents patches pour la gestion de l'alimentation, le partage mémoire, etc. permettant une meilleurs gestion de ces caractéristiques pour les appareils mobiles.

Android n'est pas linux mais il est basé sur un kernel linux.

Pourquoi sur un kernel linux ?

- Le kernel linux a un système de gestion mémoire et de processus reconnu pour sa stabilité et ses performances.
- Le model de sécurité utilisé par linux, basé sur un système de permission, connu pour être robuste et performant. Il n’a pas changé depuis les années 70
- Le kernel linux fournit un système de driver permettant une abstraction avec le matériel. Il permet également le partage de librairies entre différent processus, le chargement et le déchargement de modules à chaud.
- le kernel linux est entièrement open source et il y a une communauté de développeurs qui l'améliorèrent et rajoute des drivers.

C'est pour les points cités ci-dessus que l'équipe en charge du noyau a décidé d'utiliser un kernel linux.

Modifications apportées au noyaux
Le kernel Android a été patchés avec différents patchs :

Alarme

Ce patch fournit un certain nombre de timeurs permettant par exemple de "réveiller l'appareil quand il est en veille"

Ashmem

Android shared memory

Ce patch permet aux applications de partager la mémoire. Cette gestion est faite au niveau kernel du fait que le partage mémoire est très utilisé dans la plateforme Android car la mémoire dans les appareils mobiles est limitée par rapport à des PC.

Le partage mémoire est essentiellement utilisé par le Binder (lien).

Binder - Android IPC

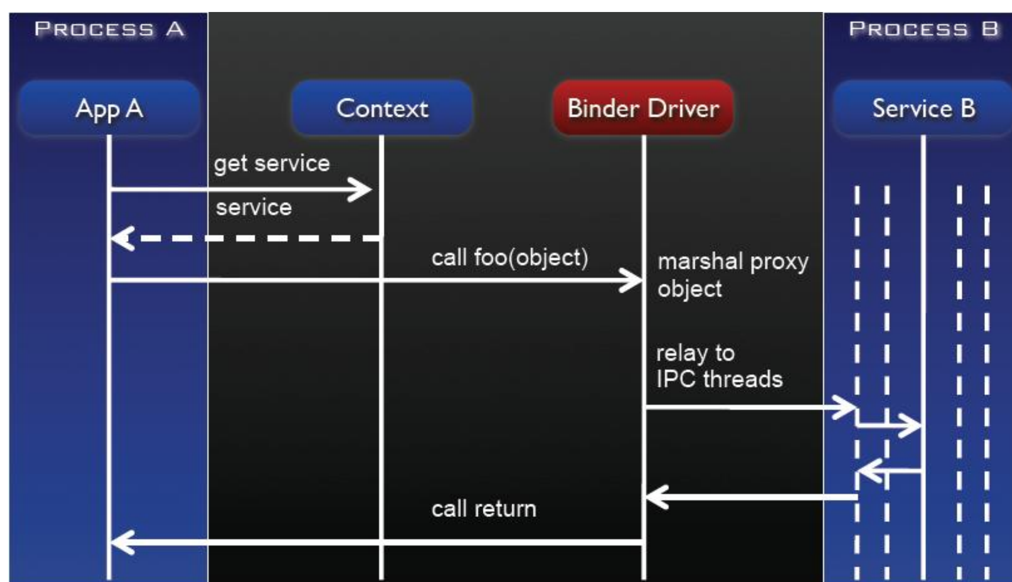
La communication interprocessus (IPC) peut entrainer des trous de sécurité, c'est pour cela qu'Android à son propre IPC, le Binder et que la communication interprocessus n'est pas laissé aux développeurs d'application. De plus, avoir un système IPC centralisé permet une maintenance plus facile et une correction des problèmes de sécurités générales.

Dans Android chaque application est lancée dans un processus différent. Ces différents processus ont besoin de communiquer ensemble, de partager des données. Cet IPC est possible avec le Binder.

Il permet à plusieurs processus de partager des données, de communiquer entre eux en utilisant le partage mémoire (ashmem driver). Cette technique permet des performances accrue par rapports à de la recopie en mémoire des données, ou de la sérialisation.

Les problèmes de concurrence, lorsque plusieurs processus essaye d'accéder en même temps à une "même zone mémoire" (au même objet java) sont gérés par le Binder. Tous les appels sont synchronisés entre les processus.

Fonctionnement



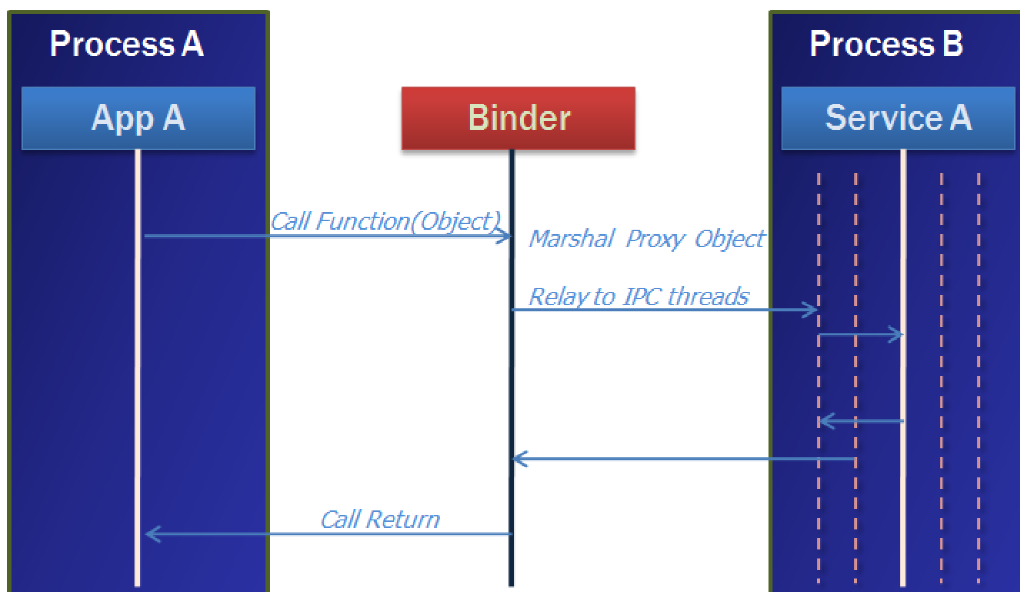
IPC

With its emphasis on isolating apps and services in process boundaries, it is clear that Android requries a lightweight IPC. The IPC mechanism in Android is called Binder and is based on shared memory. Recall that when a process starts it registers itself with the Service Manager. This happens behind the scenes, and on registeration, each process gets what is called a Context object – a reference to the Service manager. Now lets say app A needs to communicate with service B, and the two are running in two separate processes. To do this, app

A asks the Context for the Service B by passing the well known name of the service. The Context returns a reference to the service to A, on which A can call a method – say foo. This method call is intercepted by the Binder driver. The driver marshals the object and passes a reference of it to the receiver – B. Note that this is passing by reference, not passing by value, in which the object is serialized. On the side of the service B, the Binder maintains a thread-pool (transparent to the service). One of the threads in this pool receives the incoming call, locates the actual object in the service B and make the call. The return value is then similarly passed back to the caller. The following diagram (fromhttp://sites.google.com/site/io/anatomy–physiology-of-an-android) illustrates this:

L'application A récupère une référence vers le Service B à l'aide du Context Manager. Le Context Manager peut être comparé à un DNS. Il permet de récupérer à l'aide d'un nom, une référence vers un objet java. Pour ceux qui connaissent RMI (Remote Method Invocation), c'est le registry. Si on veut partager des objets, il faut au préalable les enregistrer dans le Context Manager.

Une fois la référence vers le service B récupérée, la méthode foo() du service B est appelée par l'application A. Le binder intercepte cet appel, et à l'aide d'un des threads libres présent dans sa thread pool (piscine de thread ou réservoir de threads), il va exécuter la méthode sur le service B.



Gestion de l'alimentation :

- **Problématique :**

Appareils mobiles dépendent de la puissance de la batterie et les batteries ont une capacité limitée.

- **Propriétés de Gestion de l'alimentation**
    - Power Management est construit au-dessus de la Gestion de l'alimentation standard de Linux.
    - Power Management supporte une politique de gestion de l'alimentation plus agressive.
    - Les Composants font des demandes de rester en veille à travers les «Wake Locks ».
    - Power Management supporte différents types de «Wake Locks ».

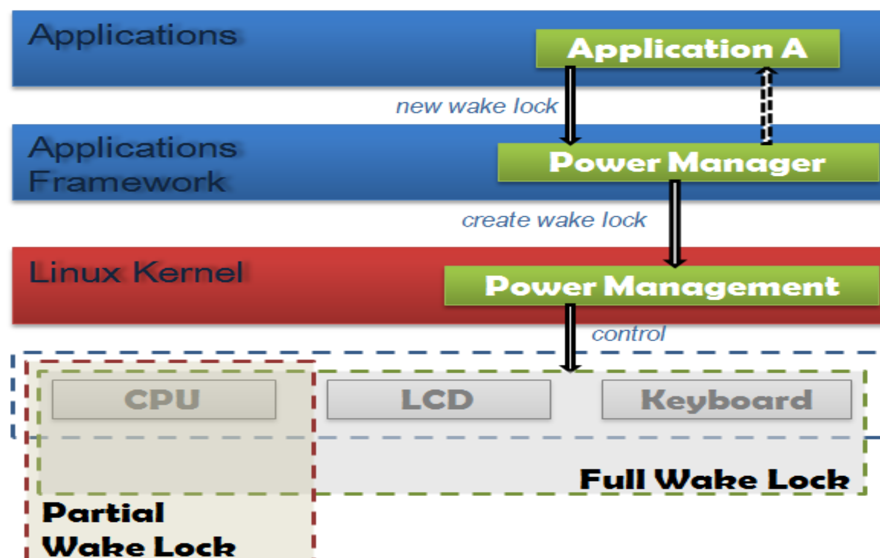
- S'il n'ya pas de «Wake Locks » actifs, le processeur sera désactivée.
- S'il n'ya pas de «Wake Locks» partiel, seul l'écran et le clavier seront éteint.

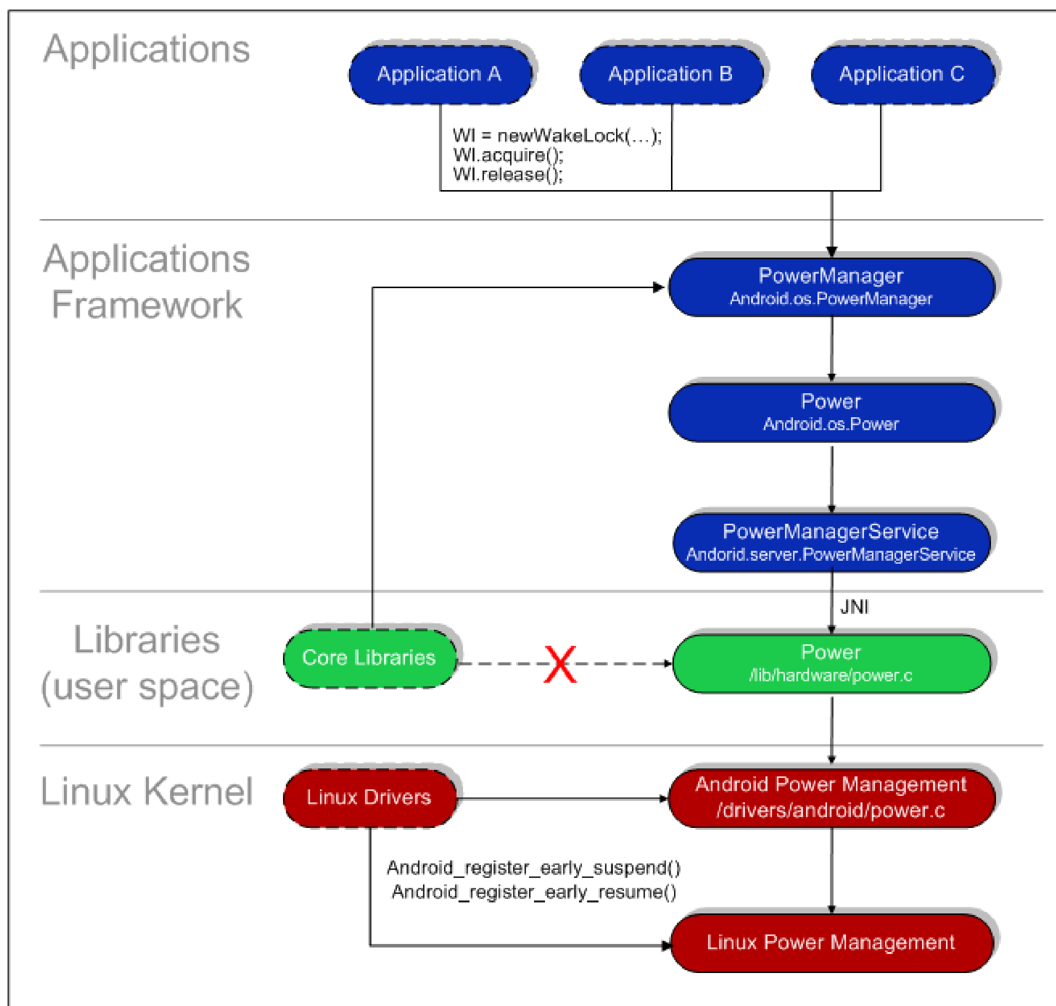
Android supporte ses propres Power Management (basé sur la Gestion d'alimentation standard de Linux), conçu sur le principe que le CPU ne devrait pas consommer de l'énergie si aucune des applications ou services nécessitent une alimentation.

Android exige que les applications et les services demandent des ressources CPU avec «Wake Locks » à travers le « Android Application Framework » et les bibliothèques linux natives.

S'il n'ya pas de «Wake Locks » actifs, Android va arrêter le CPU.

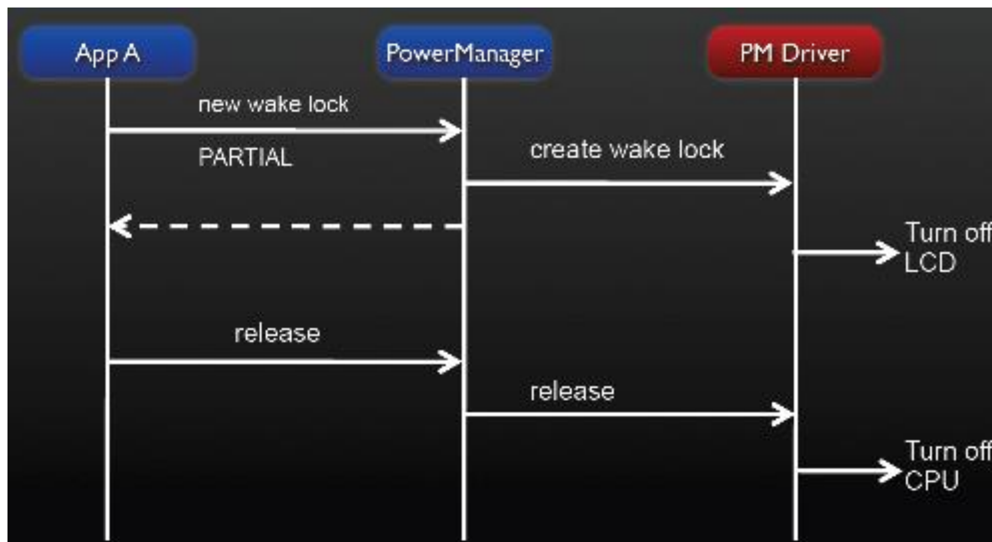
L'image ci-dessous illustre l'architecture Android gestion de l'alimentation.





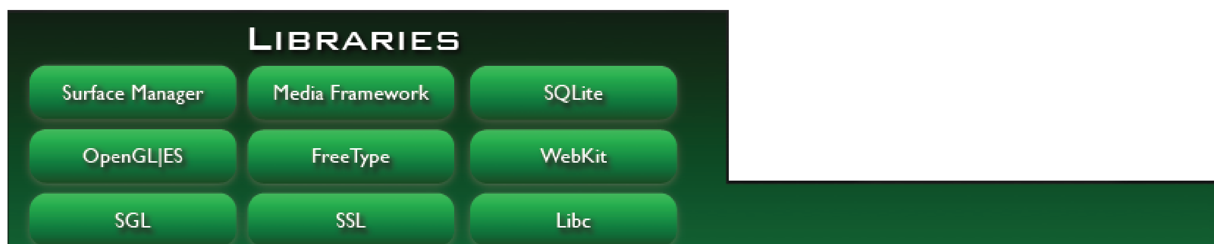
Types de « Wake Locks »

Wake Lock	Description
ACQUIRE_CAUSES_WAKEUP	D'habitude les wake locks ne font pas réveiller le périphérique, ils vont juste l'amener à rester eveillé une fois que c'est déjà fait. Pensez à l'app lecteur vidéo comme un comportement courant. Les notifications qui surgissent et veulent que le dispositif soit éveillé sont les exceptions; utiliser cette option pour être comme eux.
FULL_WAKE_LOCK	Wake lock qui garantit que l'écran et le clavier sont sur à pleine luminosité.
ON_AFTER_RELEASE	Lorsque ce wake lock est libéré , réinitialiser le timer de l'activité de l'utilisateur afin que l'écran reste allumé pendant un peu plus de temps.
PARTIAL_WAKE_LOCK	Wake lock qui assure que le CPU est en marche. L'écran pourrait ne pas être en marche.



3- Les Librairies

Au dessus du kernel, il y a les librairies natives. Ces librairies sont écrites en C/C++. Elles fournissent les fonctionnalités de bas niveau d'Android.



Bionic Libc

Android ne supporte pas la glibc, pour cela on a développé une librairie C (libc) nommé Bionic libc. Elle est optimisée pour les appareils mobiles et a été développé spécialement pour Android.

La plateforme Android à besoin de sa propre libc car il avait besoin d'une libc légère (la libc sera chargé dans chaque processus) et rapide (les appareils mobiles ne disposent de CPU puissant).

La Bionic libc a été écrit pour supporter les CPU ARM, bien que le support x86 est présent. Il n'y pas de support pour les autres architecture CPU tel que PowerPC ou MIPS. Néanmoins, pour le marché des appareils mobiles, seulement l'architecture ARM est importante.

Les **architectures ARM**, développées par *ARM Ltd*, sont des architectures RISC 32 bits.

Dotés d'une architecture relativement plus simple que d'autres familles de processeurs, et bénéficiant d'une faible consommation, les processeurs ARM sont devenus dominants dans le domaine de l'informatique embarquée, en particuliers la téléphonie mobile et les tablettes.

Ces processeurs sont fabriqués sous licence par un grand nombre de constructeurs.

Cette libc est sous licence BSD, elle reprend une grande partie du code des glibc issue d'OpenBSD, FreeBSD et NetBSD.

Caractéristiques :

-Elle à environ une taille de 200Ko soit la moitié de la glibc

-L'implémentation des pthreads (POSIX thread) a été complètement réécrite pour supporter les threads de la machine virtuelle Dalvik. De ce fait la Bionic libc ne supporte les threads POSIX

-Les exceptions C++ et les "wide char" ne sont pas supportés

-Il n'y a pas de "Standard Template Library" (STI)

La *Standard Template Library* (STL) est une bibliothèque C++, normalisée par l'ISO (document ISO/CEI 14882) et mise en oeuvre à l'aide des templates.

Cette bibliothèque fournit :

- un ensemble de classes conteneurs, telles que les vecteurs (vector), les tableaux associatifs (map), les listes chaînées (list), qui peuvent être utilisées pour contenir n'importe quel type de données à condition qu'il supporte certaines opérations comme la copie et l'assignation.
- une abstraction des pointeurs : les itérateurs. Ceux-ci fournissent un moyen simple et élégant de parcourir des séquences d'objets et permettent la description d'algorithmes indépendamment de toute structure de données.
- des algorithmes génériques tels que des algorithmes d'insertion/suppression, recherche et tri.
- une classe *string* permettant de gérer efficacement et de manière sûre les chaînes de caractères.

WebKit :

WebKit est une bibliothèque logicielle permettant aux développeurs d'intégrer facilement un moteur de rendu de pages Web dans leurs logiciels. Elle est disponible sous licence BSD et GNU LGPL. Originellement réservée au système d'exploitation Mac OS X (à partir de la version 10.3 *Panther*), elle a été portée vers Linux et Windows. Ainsi le portage de WebKit pour les environnements GTK+ et Qt se nomment respectivement WebKitGTK+ et QtWebKit.

WebKit est un fork du moteur de rendu KHTML du projet KDE utilisé notamment dans le navigateur Konqueror. Elle intègre deux sous-bibliothèques : WebCoreet JavaScriptCore correspondant respectivement à KHTML et KJS.

WebKit est moteur de rendu, qui fournit une bibliotheque sur lequel on peut développer un navigateur web. Il a été dérivé à l'origine par Apple du moteur de rendu KHTML pour être utilisé par le navigateur web Safari et maintenant il est développé par KDE project, Apple, Nokia, Google et d'autres. WebKit est composé de deux librairies : WebCore et JavascriptCore qui sont disponible sous licence GPL.

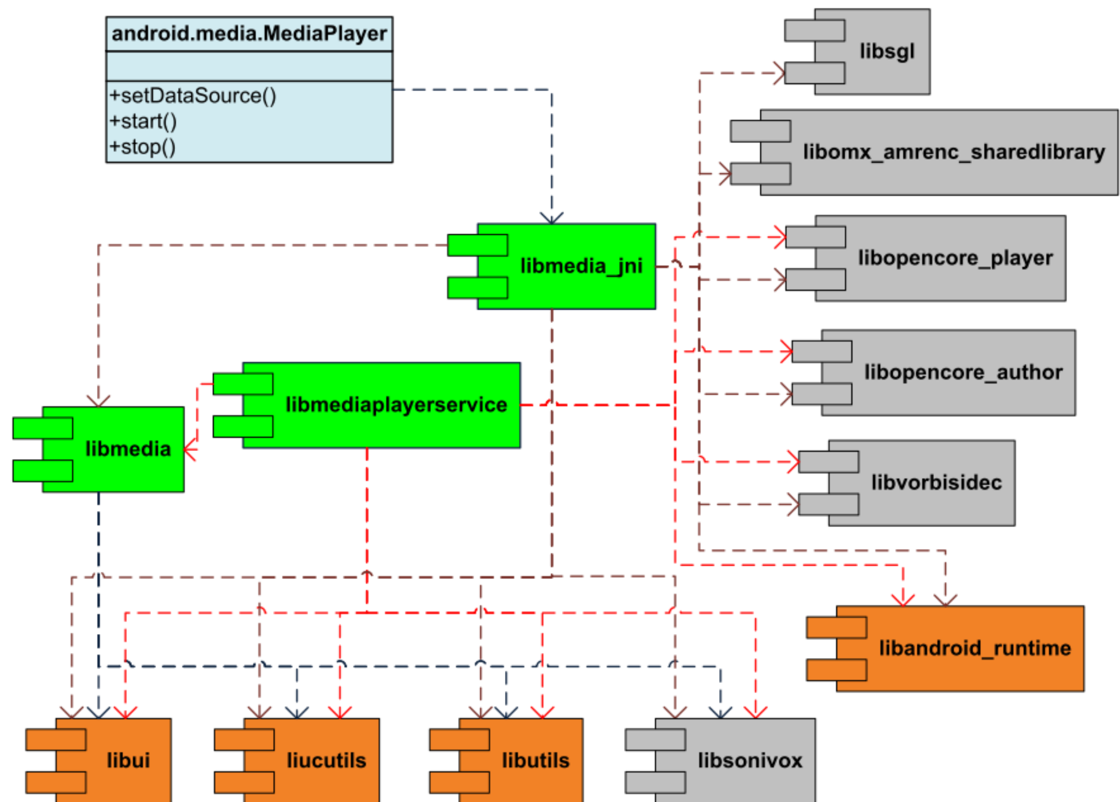
WebKit supporte le CSS, Javascript, AJAX. La version de WebKit présent dans Android à été légèrement modifiée pour s'adapter aux appareils mobiles.

Media framework

La librairie Media Framework est basée sur OpenCore de PacketVideo. Elle permet le support des standards audio/vidéo/images.

Android Media Framework est construit au dessus d'un ensemble de bibliothèques de médias, y compris OpenCore, vorbis et SONiVOX. Donc, un des objectifs de Android Media Framework est de fournir une interface cohérente pour tous les services fournis par les bibliothèques sous-jacentes et de les rendre transparents pour les utilisateurs.

La figure ci-dessous montre les relations de dépendance entre les bibliothèques du média framework.



Dans cette figure, les composants verts sont les médiathèques, les librairies android interne sont des composants jaunes, les composants gris sont les bibliothèques externes, et la classe de couleur bleue clair est le consommateur java du média framework. Sauf pour la classe android.media.MediaPlayer, tous les composants sont mis en œuvre en C ou C + +.

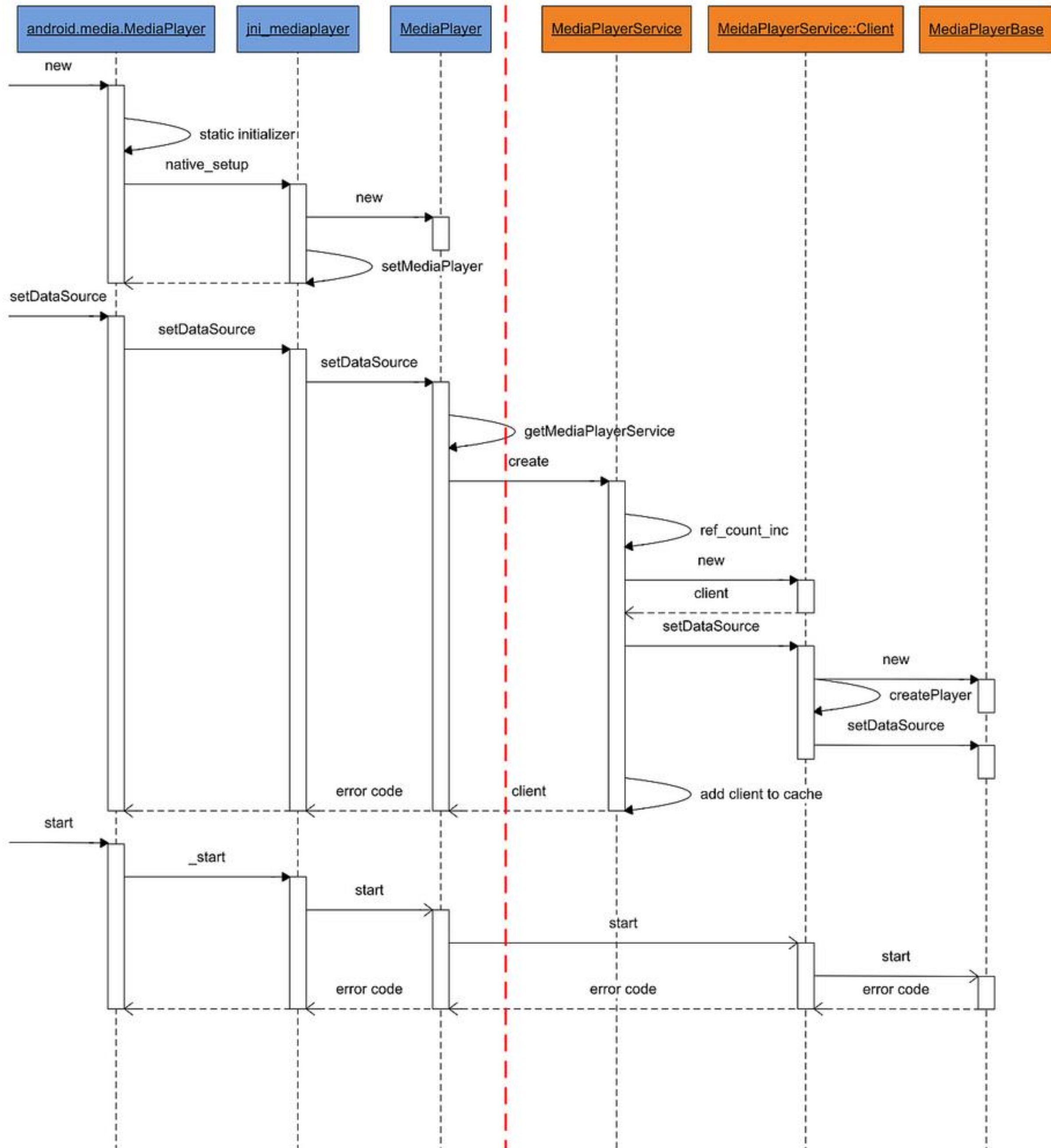
Le noyau du Media Framework est composé de libmedia, libmediaplayerservice et libmedia_jni.

libmedia définit la hiérarchie d'héritage et les interfaces de base. C'est la bibliothèque de base.

libmedia_jni est l'intermédiaire entre l'application Java et la bibliothèque native. D'abord, il implémente la spécification JNI (Le JNI (Java Native Interface) est un framework qui permet au code Java s'exécutant à l'intérieur de la JVM d'appeler et d'être appelé1 par des applications natives (c'est-à-dire des programmes spécifiques aumatériel et au système d'exploitation de la plate-forme concernée), ou avec des bibliothèques logicielles basées sur d'autres langages (C, C++, assembleur, etc.).) afin qu'il puisse être utilisé par une application Java. Deuxièmement, il implémente le modèle façade pour la commodité de l'appelant.

libmediaplayerservice met en œuvre concrètement certains Lecteurs de Medias et les services de médias qui vont gérer les instances de ces lecteurs.

Ce qui est intéressant à noter dans Android, C'est que l'application qui a l'intention d'afficher le contenu des médias et le lecteur qui affiche réellement ce contenu (audio, vidéo…) fonctionnent dans des processus différent. La ligne rouge dans le diagramme de séquence ci-dessous montre la limite de deux processus.



La figure montre les trois opérations les plus courantes, la création d'un nouveau Lecteur, mise en DataSource et la Lecture. Le dernier objet MediaPlayerBase est l'interface que l'objet MediaPlayerService::client utilise pour se référer à l'instance du lecteur concret. Le lecteur

100% COURS
MCours.com
Cours et mémoires gratuits
mcourscom@gmail.com

concret peut être VorbisPlayer, pvPlayer, ou n'importe quel autre Lecteur, en fonction du type du média lu.

Quand une application crée un objet android.media.MediaPlayer, il utilise en fait un objet Proxy pour manipuler le Lecteur concret résidant dans le processus MediaServer. Pendant toute la procédure, deux processus communique avec Binder IPC.

Ayant des connaissances ci-dessus, il n'est pas difficile de comprendre pourquoi MediaPlayer ne fournit pas une API à utiliser comme source de flux de mémoire. Parce que la mémoire manipulée par le flux est en l'espace d'adressage de l'application, et ce n'est pas directement accessible par le processus media server.

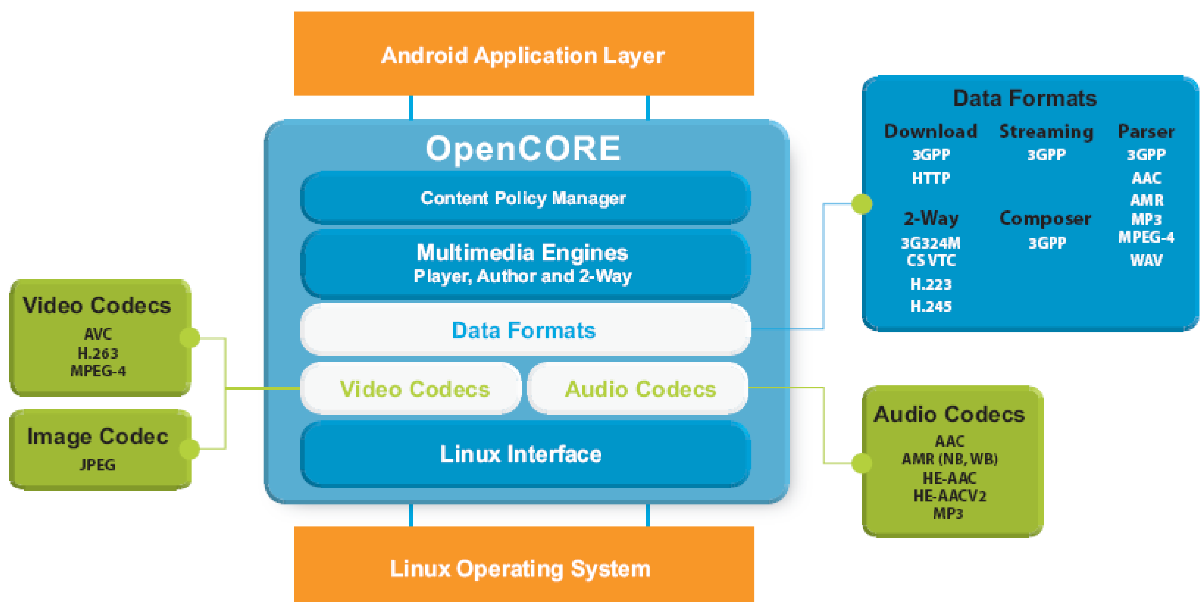
Le schéma ci dessous décrit toutes les fonctionnalités fournit par cette librairie :

Médiathèques – basée sur PacketVideo de OpenCore; les librairies permettant la lecture et

l'enregistrement audio et vidéo, ainsi que la gestion des fichiers image, y compris MPEG4, H.264,

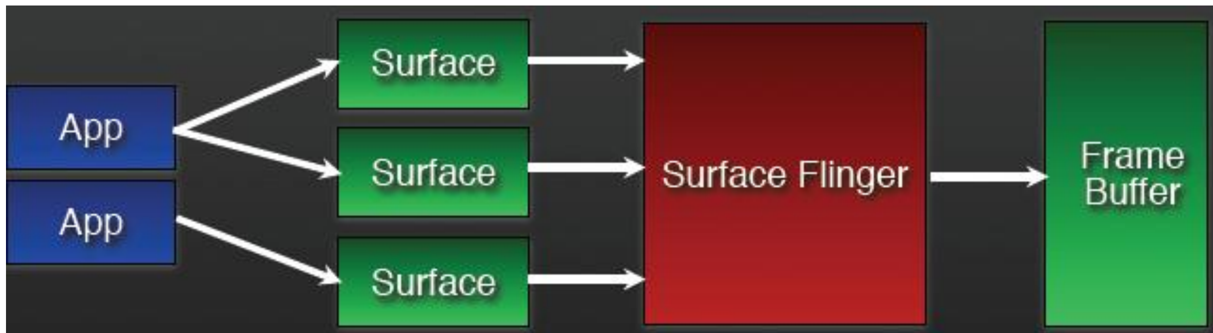
MP3, AAC, AMR, JPG et PNG.

Le schéma ci-dessous décrit tous les éléments de l'architecture de ces médiathèques:



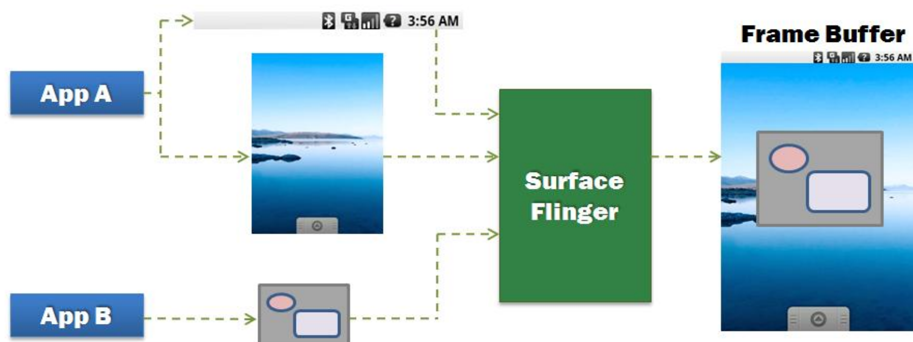
SURFACE FLINGER :

Le Surface Flinger permet de construire le rendu graphique, il manipule toutes les surfaces à afficher provenant du frame buffer. Il peut combiner de la 2D et de la 3D provenant de différentes applications. Les surfaces à afficher sont passées par buffers via le Binder.



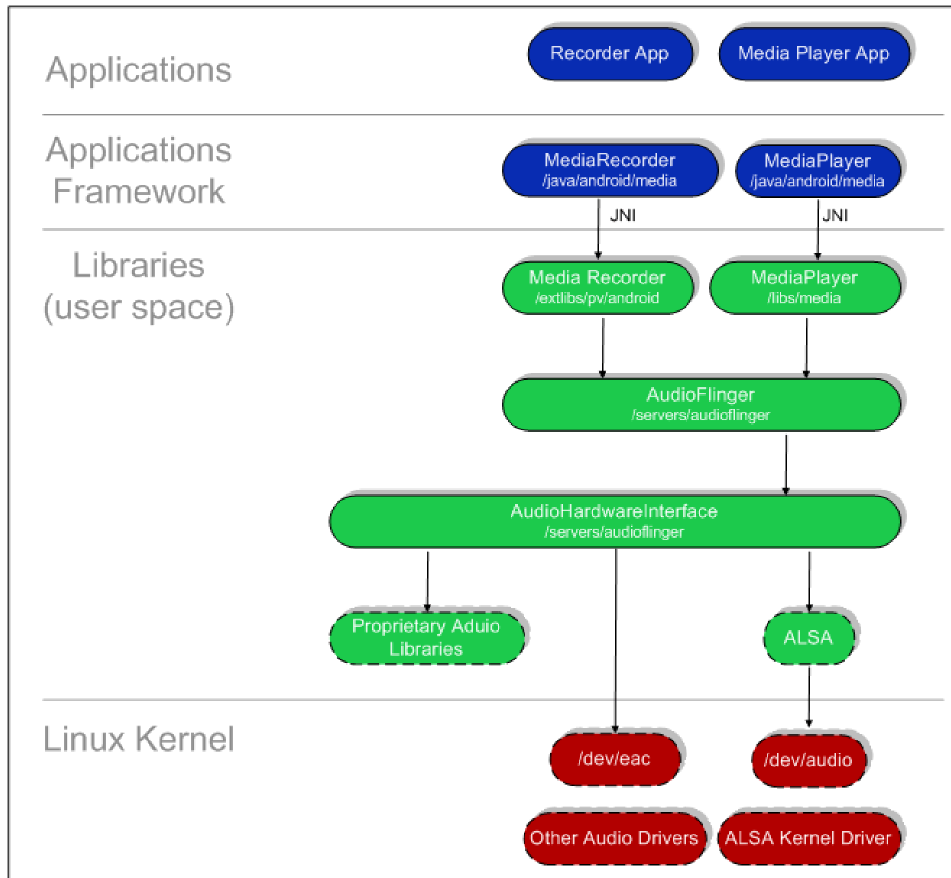
Le surface flinger utilise un double buffer permettant de basculer d'une surfaces à une autre rapidement. Ce double buffer permet également de ne jamais afficher des surfaces incomplètes, car le deuxième buffer n'est affiché que lorsque celui est complet. Les deux buffers sont utilisés tour à tour.

Il peut utiliser OpenGL ES et l'accélération matérielle pour le rendu 2D en utilisant l'API khronos.

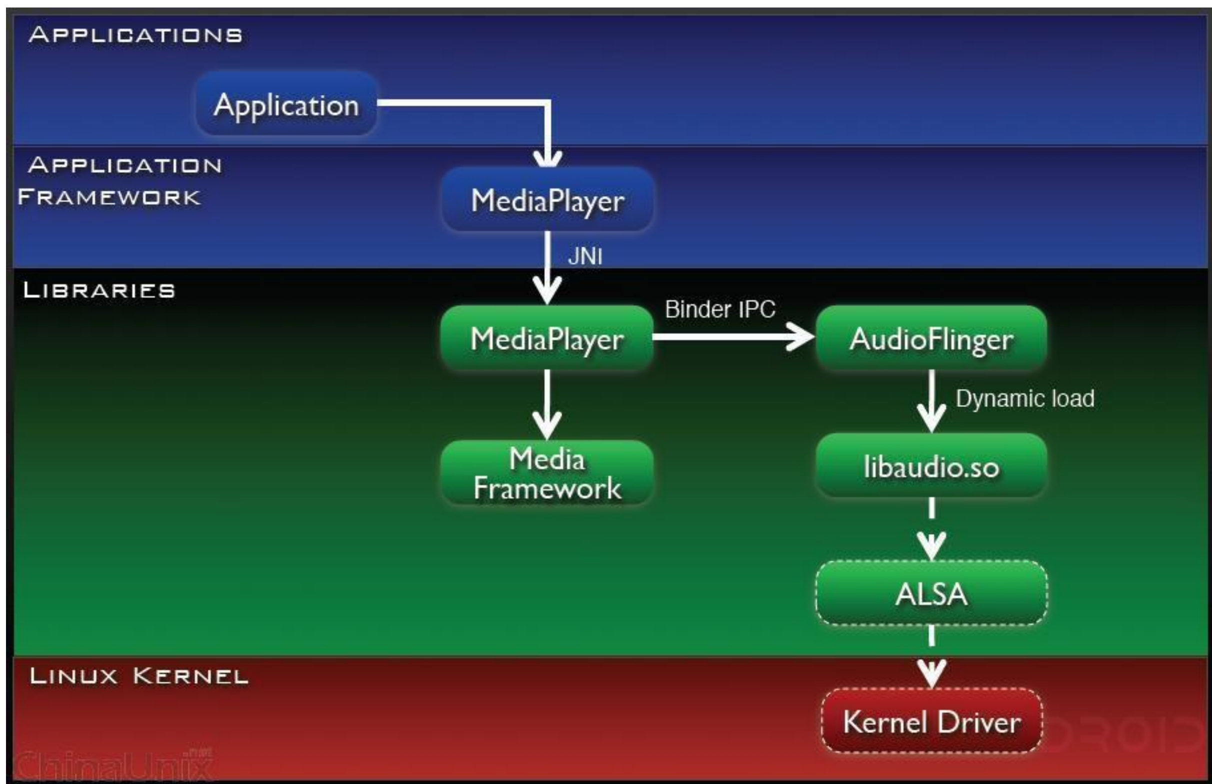


Audio Flinger

L'audio flinger gère tous périphériques audio. Il traite les flux audio et les route vers les périphériques de sortie (haut parleur, Bluetooth, casque).

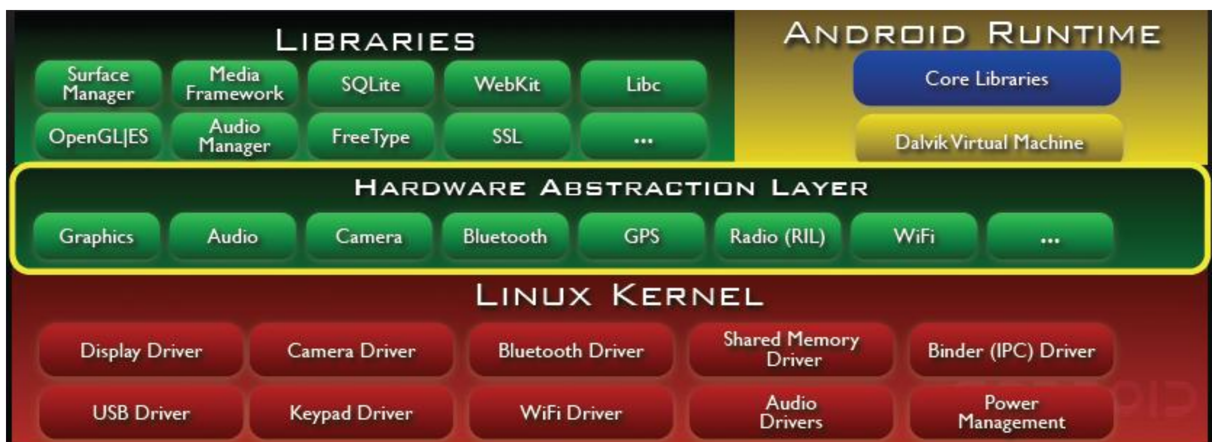

Applications
Recorder App
Media Player App
Applications Framework
MediaRecorder
/java/android/media
MediaPlayer
/java/android/media
JNI
JNI
Libraries (user space)
Media Recorder
/extlibs/pv/android
MediaPlayer
/libs/media
AudioFlinger
/servers/audioflinger
AudioHardwareInterface
/servers/audioflinger
Proprietary Aduio Libraries
ALSA
Linux Kernel
/dev/eac
/dev/audio
Other Audio Drivers
ALSA Kernel Driver

App A
App B
Tone Audio
Media Player
Game Audio
Audio Flinger
Earpiece
Speaker
Bluetooth



Le Hardware Abstraction Layer

Cette couche se situe entre les librairies et le kernel linux, elle fournit les interfaces que doivent implémenter les drivers kernel. Cette couche sépare la plateforme logique des interfaces matérielles. Le but de cette couche est de faciliter le portage des librairies sur différents matériels.



Les concepteurs d'Android ont décidé de faire cette couche car :

- Les drivers kernel ne possèdent pas tous des interfaces standardisées.
- les drivers kernel sont sous licence GPL ce qui exposerait les interfaces propriétaires des fabricants. Les fabricants veulent pouvoir garder ces interfaces en "closed source"
- Android a des besoins spécifiques pour les drivers kernel.

4- Android Runtime

Cette couche se situe au dessus des libraires C/C++, elle se compose du "cœur" du Framework et de la machine virtuel dalvik.

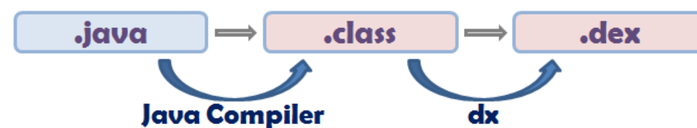
Dalvik

La machine virtuelle Dalvik est basée sur une architecture de registre à l'instar de beaucoup de machine virtuel et de la machine virtuelle Java qui ont une architecture de pile.

Utiliser une architecture de pile ou de registre dépends des stratégies de compilation et d'interprétation choisit. Généralement, les machines basées sur une architecture de pile, doivent utiliser des instructions pour charger les données sur la pile et manipuler ces données. Ce qui rajoute des instructions dans le code machine, et donc il y a plus de code que pour une machine basé sur une architecture de registre.

Cependant, les instructions pour une machine basée sur une architecture de registre doivent être encodées pour les registres sources et destinations, ce qui prend également de la place dans le code machine résultant. La différence est essentiellement importante suivant l'interpréteur de code machine présent dans la VM.

Les applications Java développées pour Android doivent être compilées au format dalvik exécutable (.dex) avec l'outil dx. Cet outil compile les .java en .class et ensuite il convertit ces .class en .dex. Un .dex peut contenir plusieurs classes. Les strings dupliqués et autre constantes utilisées dans de multiples classes sont regroupés dans un .dex. Le bytecode utilisé dans les .dex est le Dalvik bytecode et non le java Bytecode.



Pour comparaison un .dex décompressé est un peu plus petit en taille qu'un .jar compressé dérivé des mêmes fichiers .class.

Etant optimisé pour utiliser une quantité de mémoire minimale, la VM Dalvik a quelques caractéristiques spécifiques par rapport aux autres VM:

- la VM a été "dégraissée" pour utiliser moins d'espace mémoire
- pas de compilation à la volé (JIT)
- Elle utilise sont propre bytecode et pas le Java bytecode
- La table des constantes a été modifié pour n'utiliser que des indexes de 32 bit afin de simplifier l'interpréteur.

Core Libraries

Les libraries Core fournissent le langage Java disponible pour les applications. Le langage Java fournit avec Android reprend en grande partie l'API JSE 1.5. Il y a des choses qui ont été mis de coté car cela n'avait pas de sens pour Android (comme les imprimantes, swing, etc.) et d'autres par ce que des APIs spécifiques sont requises pour Android.

Packages JSE 1.5 supportés par Android	Packages JSE 1.5 **non** supportés par Android
• java.io • java.lang (sauf java.lang.management)	• java.applet • java.awt

• support • java.math • java.net • java.nio • java.security • java.sql • java.text • java.util • javax.crypto • javax.net • javax.security (sauf javax.security.auth.kerberos, javax.security.auth.spi, and javax.security.sasl) • javax.sound • javax.sql (sauf javax.sql.rowset) • javax.xml.parsers • org.w3c.dom • org.xml.sax	• java.beans • java.lang.management • java.rmi • javax.accessibility • javax.activity • javax.imageio • javax.management • javax.naming • javax.print • javax.rmi • javax.security.auth.kerberos • javax.security.auth.spi • javax.security.sasl • javax.swing • javax.transaction • javax.xml (sauf javax.xml.parsers) • org.ietf.* • org.omg.* • org.w3c.dom.*

Librairies spécifiques ajoutées dans les Core Libraries d'Android :

- org.apache.commons.codec
- org.apache.commons.httpclient
- org.bluez
- org.json

Le Framework Application :

Le framework est situé au dessus de l'Android Runtime et des librairies. Il fournit des API permettant aux développeurs de créer des applications riches.

Core Plateform Services

Android introduit la notion de services. Un service est une application qui n'a aucune interaction avec l'utilisateur et qui tourne en arrière plan pendant un temps indéfini.

Les services cœurs de la plateforme (Core Plateform Services) fournissent des services essentiels au fonctionnement de la plateforme :

- **Activity Manager** : gère le cycle de vie des applications et maintient une "pile de navigation" (navigation backstack) permettant d'aller d'une application à une autre et de revenir à la précédente quand la dernière application ouverte est fermée.
- **Package Manager** : utilisé par l'Activity Manager pour charger les informations provenant des fichiers .apk (android packaga file)
- **Window Manager :** juste au dessus du Surface Flinger (lien), il gère les fenêtres des applications : quelle fenêtre doit être affichée devant une autre à l'écran ?
- **Resouce Manage :** Gère tous ce qui n'est pas du code, toutes les ressources --> images, fichier audio, etc.
- **Content Provider :** gère le partage de données entre applications, comme par exemple la base de données de contact, qui peut être consultée par d'autres applications que l'application Contact. Les Données peuvent partager à travers une base de données (SQLite), des fichiers, le réseau, etc.
- **View System** : fournit tous les composants graphiques : listes, grille, textbox, buttons et même un navigateur web embarqué.

Hardware Services

Les services matériels (Hardware Services) fournissent un accès vers les API matérielles de bas niveau :

- **Telephony Service** : permet d'accéder aux interfaces "téléphonique" (gsm, 3G, etc.)
- **Location Service** : permet d'accéder au GPS.
- **Bluetooth Service** : permet d'accéder à l'interface bluetooth.
- **WiFi Service** : permet d'accéder à l'interface Wifi.
- **USB Service** : permet d'accéder aux interfaces USB.
- **Sensor Service** : permet d'accéder aux détecteurs (détecteurs de luminosité, etc.)

Le Processus de Démarrage :

A- Comme tout système Linux, au démarrage le bootLoader charge le kernel et lance le processus init. Le père de tous les processus.

B- Ensuite un certain nombre de daemons sont lancés par le processus init :

  - USB daemon (usbd)
  - Android Debug Bridge (adbd)
  - Debugger Daemon (debuggerd); Il permet de gérer les requêtes de debug de processus (dump memory, etc.)
  - Radio Interface Layer Daemon (rild)

Par la suite le processus init lance le processus Zygote. Au démarrage, une instance spéciale de la machine virtuelle Java est lancé, appelé zygote. Ce processus charge un tas de classes Java de base et effectue leur traitement initial, ce qui rend possible d'éviter cette étape pour chaque lancement d'application. Une fois le travail initial est fait, le processus attend les requêtes en écoutant sur un socket.

Le processus Zygote :

- initialise une instance de la Dalvik VM
- Pré charge les classes de base et écoute sur un socket pour créer des Dalvik VM
- Fork sur demande pour créer des instances de Dalvik VM pour chaque application. fork () crée un nouveau processus en dupliquant le processus appelant. Le nouveau processus, appelé l'enfant, est une copie exacte du processus appelant, appelé en tant que parent
- Les VM créées partagent des zones mémoire communes ce qui permet de minimiser la mémoire utilisées
- Chaque VM créer par zygote est un fork d'une VM "mère", ce qui permet d'accélérer le démarrage d'une application.

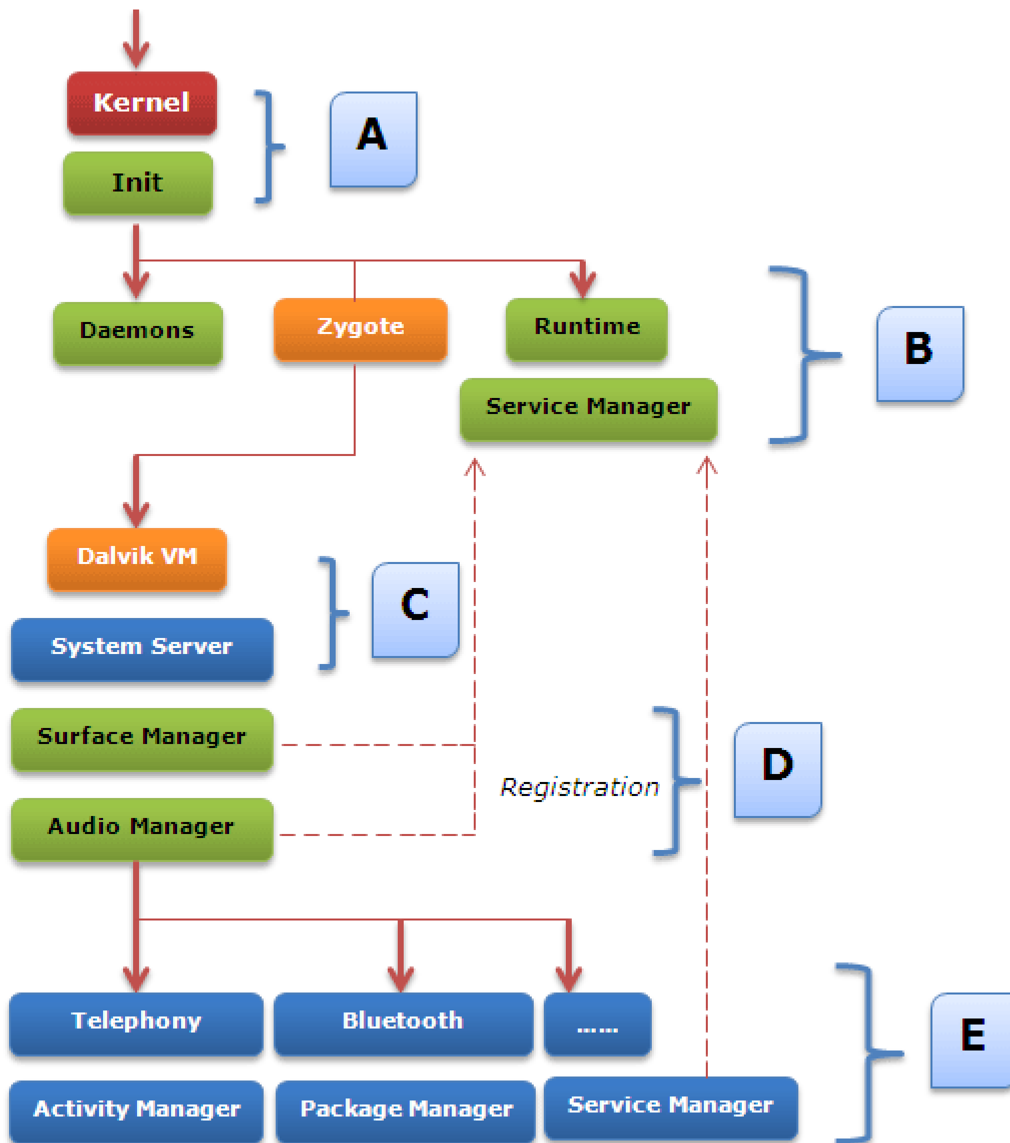
Ensuite le processus **init** lance le processus **Runtime** qui va lancer à son tour le **Service Manager** ("DNS" fournissant un moyen de communiquer avec un service donné par son nom, responsable de la gestion de l'IPC permettant d'enregistrer et de récupérer des références vers des services andoid) et enregistre ce Service Manager comme le Context Manager par défaut.

C- Une fois tout cela est fait, le processus **Runtime**, envoie une requête au processus zygote lui demandant de lancer le **System Server**. Zygote va forker une nouvelle instance de Dalvik VM pour le processus System Server et démarrer le service. C'est le premier processus (après Zygote) qui execute une instance de Dalvik VM.

D- Le System Server va lancer à son tour l'Audio Flinger et le surface Flinger qui vont ensuite s'enregistrer au près du Service Manager.

E- Le System Server lance ensuite les services d'Android (gestionnaire de fenêtres,gestionnaire de téléphonie, Power Manager, gestionnaire de l'activité, etc…). Ces services une fois lancés vont s'enregistrer au près du Service Manager.

Une fois tous les services chargés, le système est prêt. Des applications utilisateur peuvent être lancées.



A ce stade, nous avons les processus suivants en place:

1. **Init** - le processus d'initialisation d'origine a commencé par le programme d'amorçage

2. **Démons** - démarrés par init

3. **Runtime** - aussi commencé par init

4. **Zygote** - le zygote original qui va sans cesse faire des forks pour que de nouvelles applications soient lancées

5. **System Server** - le premier processus géré qui contient tous les services et composants de la plateforme de base.

Après cela, l'écran d'accueil ou l'écran de veille est lancé - essentiellement le gestionnaire d'activité (qui est à l'intérieur du systèm server) envoie un message à Zygote pour lancer l'activité «Home», ce qui conduit Zygote à faire un fork dans un nouveau processus d'une Dalvik VM et de l'activité «Home». Maintenant, en fonction de l'action menée par l'utilisateur, l'application appropriée serait

lancé avec le fork Zygote à chaque fois pour créer une nouvelle instance VM dans un nouveau processus. Ainsi, par exemple, si l'utilisateur lance l'application Contacts, le processus apt serait installé:

Chaque nouvelle application reçoit un ID utilisateur unique qui est inconnu par l'application, et le système définit les autorisations pour tous les fichiers dans l'application de sorte que seul l'ID utilisateur attribué à l'application peut accéder à ces fichiers. ceci rend le système sécurisé puisque c'est la seule application a accès à des composants dont il a besoin pour faire son travail et rien d'autre. Toute fois, les applications peuvent avoir besoin de communiquer entre eux et avec les services du système, qui sont tous exécutés dans des processus séparés. Ceci est accompli grâce au Binder IPC

Applications

Une application Android est une collection de composants. Il ya quatre types de composants:

- Activité: Une activité est un composant d'interface utilisateur correspondant à un écran avec lequel un utilisateur interagit de manière à faire quelque chose.
- Service: un service est un composant de l'application sans interface utilisateur utilisée pour effectuer de longues opérations en tâche de fond.
- Broadcast Receiver: est un élément qui répond à l'échelle du système à des émissions (par exemple pour l'écran éteint, batterie faible, etc)

- Fournisseur de contenu: Un fournisseur de contenu est un élément qui stocke et récupère les données et les rend disponibles à toutes les applications. Il existe différents types de fournisseurs de contenu: audio, vidéo, contacts, etc et vous pouvez créer votre propre fournisseur de contenu aussi.

100% COURS
MCours.com
Cours et mémoires gratuits
mcourscom@gmail.com

As mentioned earlier, each application runs in its own process. By default, all components used by that app also run in the same process, and on the main thread. However, it is possible to make a component of your app run in a separate process thru the manifest file. Thus, an application in Android may span multiple processes.

Comme mentionné précédemment, chaque application s'exécute dans son propre processus. Par défaut, tous les composants utilisés par cette application sont également exécutés dans le même processus, et sur le thread principal.

Cependant, il est possible de faire un composant de votre application s'exécutant dans un processus séparé à travers le fichier « manifest » de l'application. Ainsi, une application sous Android peut s'étendre sur plusieurs processus.

Démarrage d'une Application

Android suit un modèle assez unique en ce qu'il n'ya pas de point d'entrée unique pour une application - il n'ya pas de fonction main ().

Par contre, un composant dans une application Android peut démarrer un autre dans une autre application. Cette communication entre les applications qui se passe à travers le mécanisme IPC décrit précédemment. Ainsi, alors que l'activité est détenue par une application, il est possible pour une autre application de la lancer (si l'application propriétaire le permet).

Un exemple est de cliquer sur un lien hypertexte dans une application qui ouvre le navigateur. Ceci s'applique non seulement pour les activités, mais d'autres types de composants aussi. Pour ce faire, il ya deux étapes nécessaires:

1. Dans le cas où l'application n'est pas déjà lancé, le système Android apporterait l'application à la vie dans un nouveau processus par un fork du zygote.

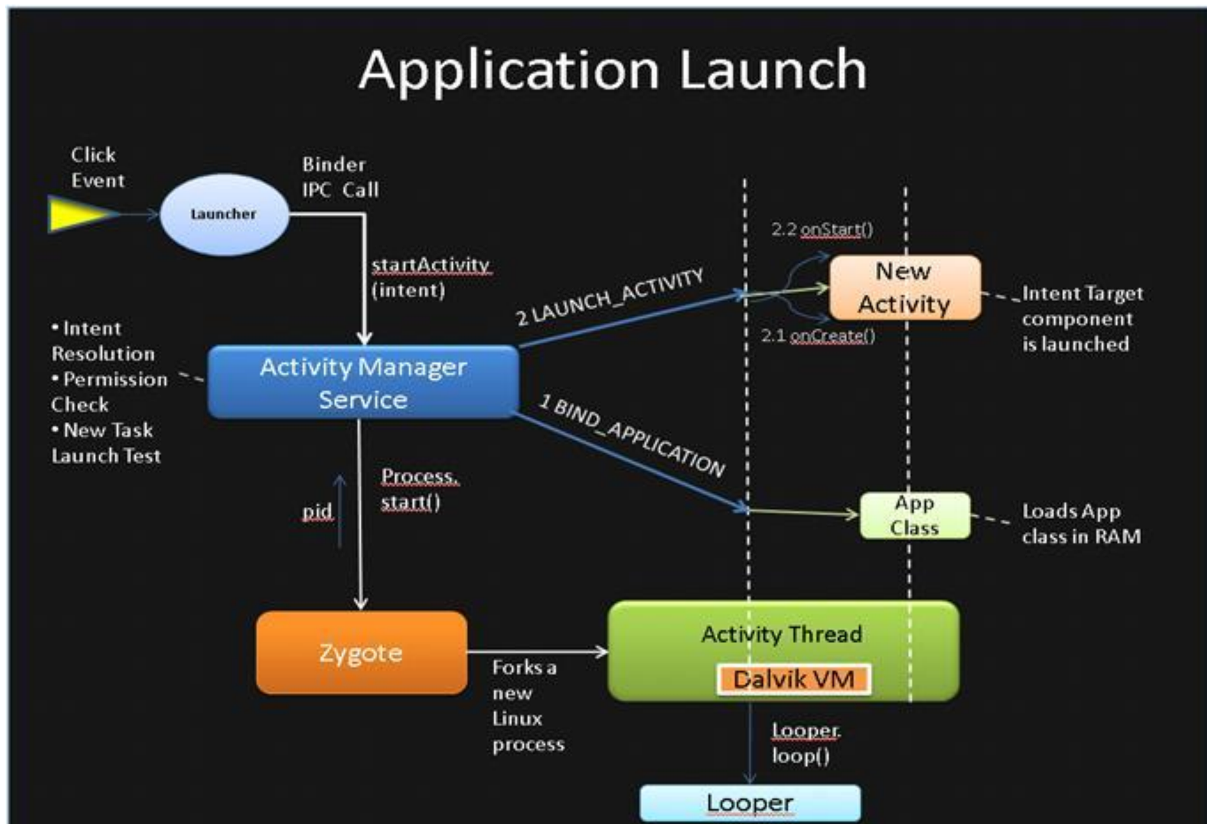
2. Le composant désiré à l'intérieur de l'application aurait besoin d'être activé.

Notez que dans le cas où l'application est déjà lancée, le nouveau composant sera instancié par défaut dans le même processus.

Comme mentionné plus haut, l'IPC passe à travers un objet « Context ». Alors, quand un composant « A » à l'intérieur d'une application doit activer un autre composant « B » dans une application différente, ou lui donner quelque chose de nouveau à faire, il utilise essentiellement l'objet « Context » pour envoyer un message à l'autre composant. Dans le cas d'une activité, de service ou un BroadcastReceiver, cela prend la forme de ce qu'on appelle une intention « Intent ».

C’est une structure passive de données qui définit l'opération à effectuer pour une activité et un service, et pour un BroadcastReceiver, c’est une définition de la déclaration en cours de diffusion.

Les fournisseurs de contenu ne sont cependant pas activés par Intentions. Au lieu de cela, l'activation se produit sur une demande d'un ContentResolver, qui agit comme un médiateur entre le composant demandeur et le fournisseur de contenu.



Le Back-Stack

Considérons le scénario suivant:

1. Vous êtes sur l'écran d'accueil. Ceci est l’activité 1

2. Vous cliquez sur l'icône de l’application Mail, qui active l’activité principale dans l'application Mail qui apparait au premier plan. C'est l'activité 2

3. Vous cliquez maintenant sur Composez et qui active l'activité de composition dans l'application Mail. C'est l'activité 3

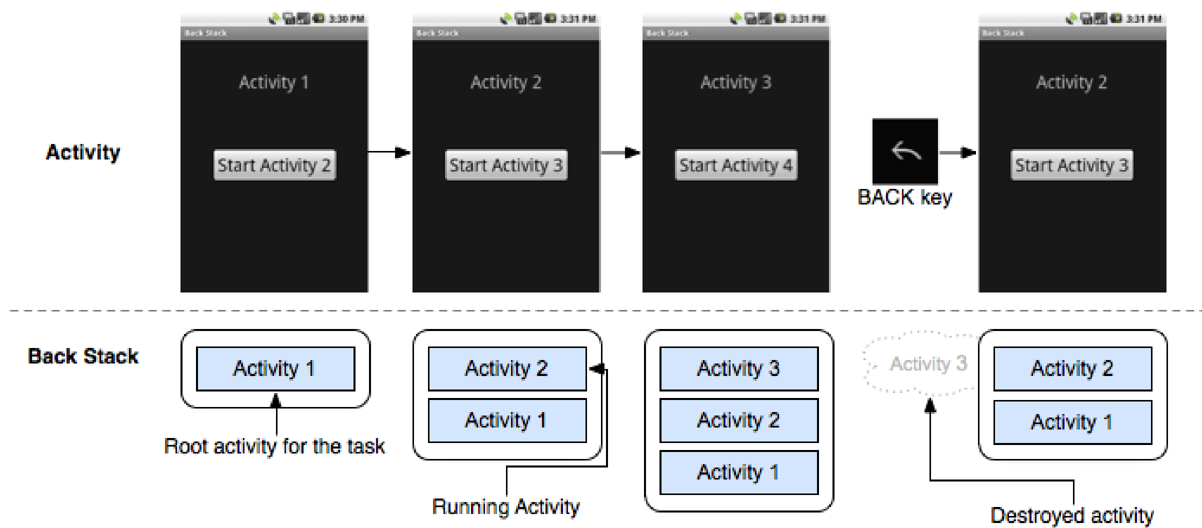
4. Vous décidez d'annuler de composer un nouveau message et appuyez sur le bouton de retour. Vous revenez à l'activité 2

Voici ce qui arrive dans le fond:

• Quand une activité démarre une autre, elle s'arrête et son état est sauvegardé. Alors, quand l'activité 1 commence Activité 2, activité 1 est arrêtée, son état sauvegardé et ainsi de suite

• Le système maintient une pile (appelée un Back-Stack) avec la dernière activité sur le dessus et la plus ancienne activité à la base.

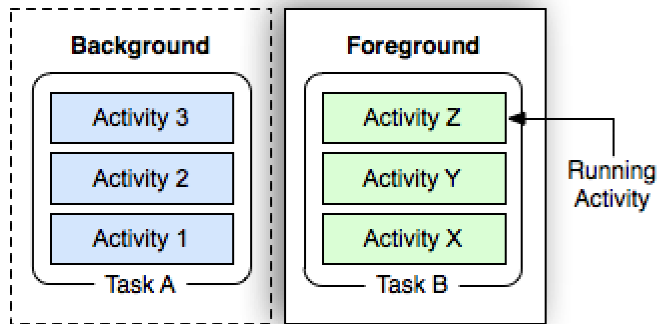
• Lorsque l'utilisateur appui sur le bouton retour de l'appareil, l'activité 3 est sorti de la pile et l'Activité 2 est démarré à partir de son état sauvegardé



Les Tâches

Dans le scénario décrit ci-dessus, on suppose que lorsque l'utilisateur a été dans l'application Nouveau Mail (Activité 3), il a décidé d'appeler quelqu'un et a appuyé sur le bouton Accueil. Ce ne va pas dépiler le back-stack, mais commencer une nouvelle pile. Pour ce faire, la collection d'activités dans la première pile a besoin de passer en arrière plan. Ceci est réalisé grâce à la notion de « tâche »

Une tâche est une unité cohérente d'Activités. Quand une tâche passe vers l'arrière-plan, toutes les activités dans cette tâche sont arrêtés, mais le back-stack de la tâche demeure intact, de sorte que lorsque l'utilisateur retourne à la tâche, elle peut reprendre où elle s'était arrêtée. Cependant, pour économiser de la mémoire, le back-stack d'une tâche de fond n'est pas retenu pour une très longue période et si l'utilisateur ne revient pas à la tâche, le back-stack est effacé, à l'exception de l'activité root.



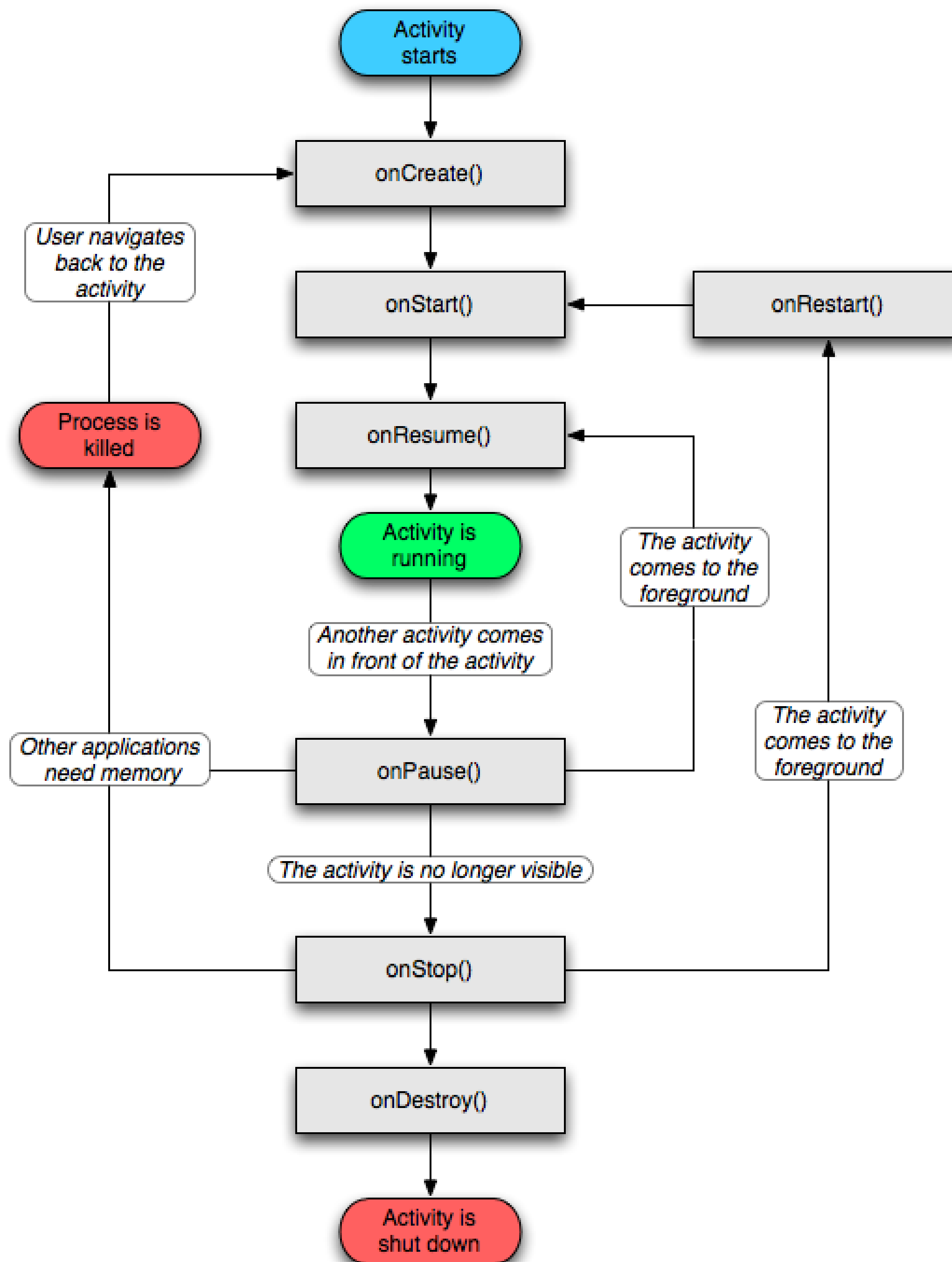
Cycle de vie d’une activité

Le cycle de vie d'une activité est affecté par son association avec d'autres activités, sa mission et son back-stack. Il ya trois états dans lesquels une activité peut exister:

• Reprise / En execution: L'activité est au premier plan et a le focus utilisateur. Une telle activité n'est jamais tuée par le système.

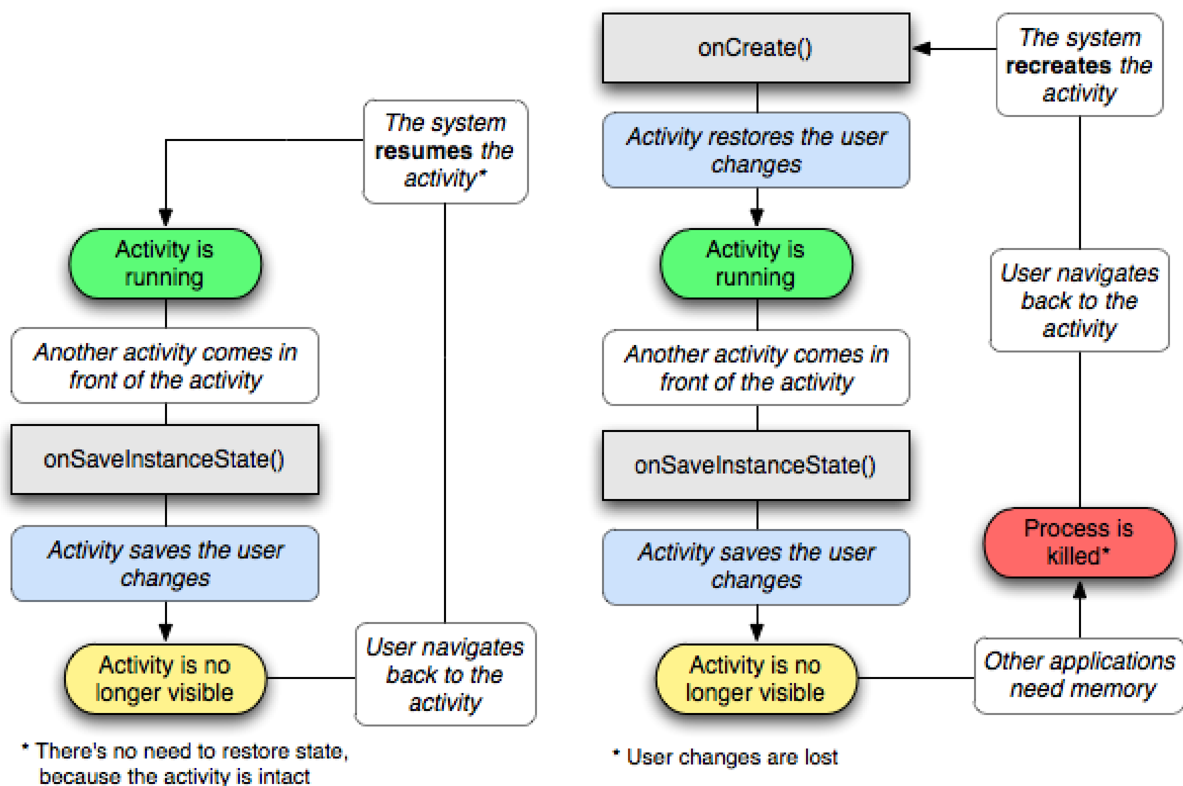
• En Pause: L'activité est visible, mais une autre activité est au premier plan et a le focus utilisateur. Cela pourrait arriver si l'autre activité est au sommet, mais elle est translucide, ou parce qu'elle ne couvre pas tout l'écran. Dans l'état de pause, l'activité est conservée en mémoire, maintient toutes les infos d'État et reste attaché au gestionnaire de fenêtres. Cependant, il peut être tué dans des conditions de mémoire extrêmement faible.

• Arrêté: l'activité n'est pas visible à l’utilisateur. Tandis que l'activité est conservée en mémoire, maintient toutes les infos d'État, il ne reste pas attaché au gestionnaire de fenêtres. L'activité peut être tuée par le système pour récupérer de la mémoire si nécessaire.



Enregistrement de l'État de l'activité

Afin de récupérer de la mémoire notez qu'il est tout à fait possible que le système, peut détruire une activité, ou même le processus dans lequel l'activité était en marche. Toutefois, lorsque l'utilisateur revient à l'activité (par le back-stack), on reprend au point où l'utilisateur a quitté. Pour ce faire, une activité doit enregistrer son état. Cela se passe via la méthode onSaveInstanceState ().



Cycle de vie de processus

Le système Android peut avoir besoin de tuer un processus, afin de récupérer de la mémoire. Afin de s'assurer que cela crée un impact minimal sur l'expérience utilisateur, Android range les processus dans un ordre de priorité:

• **Processus de premier plan**: Un processus qui est requis pour ce que l'utilisateur est en train de faire. Un tel processus est tué seulement comme un dernier recours

• **Processus visibles:** Ce processus n'est pas au premier plan, mais peut affecter ce que l'utilisateur voit sur l'écran. Par exemple, il peut accueillir une activité en pause. Un tel processus n'est pas tué à moins que cela est nécessaire pour maintenir tous les processus du premier plan, en execution

• **Procressus Service:** Un processus qui exécute un service et n'est pas l'un des deux types ci-dessus. Par exemple, le service peut jouer de la musique ou faire du téléchargement de quelque chose. Le système peut tenir un tel processus en exécution tant qu'il y ait assez de mémoire pour garder les processus de premier plan et visibles en marche.

Processus d'Arrière plan: Un processus qui gère une activité qui n'est pas actuellement visible à l'utilisateur (l'activité est arrêtée). Un tel processus n'a pas d'impact sur l'expérience utilisateur (si le cycle de vie de l'activité est correctement mis en œuvre et l'état d'activité est correctement sauvegardé et restauré). Le système peut tuer un tel processus à tout moment. Généralement, il existe plusieurs processus d'arrière-plan et le système maintient une liste LRU qui est utilisé pour tuer ces processus.

• **Processus vides:** un processus vide ne détient aucun composant actif. La seule raison pour qu'un tel processus soit maintenu en vie, en premier lieu est pour la mise en cache et l'amélioration du temps de démarrage. Le système tue souvent ces processus afin de maintenir l'équilibre entre les caches processus et les caches du noyau sous-jacent.

Bien sûr, il peut donc arriver qu'un processus de forte priorité soit dépendant d'un processus de faible priorité. Dans un tel cas, le classement des processus servant augmente au même niveau que celui du processus dépendant.

Une situation où ce classement a un impact direct est la suivante:

• Votre application doit télécharger quelque chose de grand qui peut prendre du temps et l'utilisateur est susceptible de se déplacer hors de l'activité de l'application à autre chose.

• Si vous créer un thread pour cette tâche et l'utilisateur se déplace, le processus devient un processus de fond.

• Cependant, si vous créer un service à la place, le processus serait un processus de service et sera moins susceptibles d'être tués

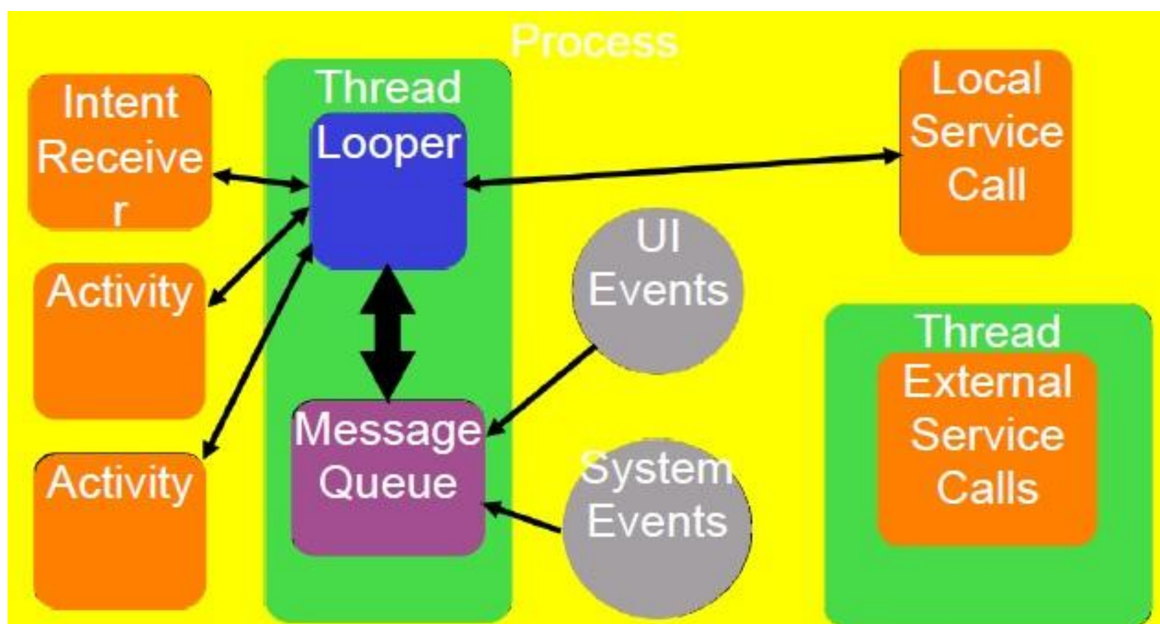
Main / UI Thread

Quand une application est lancée et un nouveau processus est créé pour l'accueillir, l'application reçoit un seul thread appelé thread principal ou le thread d'interface puisque ce thread est responsable du service de l'interface utilisateur.

La façon dont cela fonctionne est la plupart du temps la même chose pour presque n'importe quelle implémentation d'interface utilisateur UI - que ce soit les systèmes d'exploitation de bureau ou mobile OS.

Fondamentalement, le thread d'interface utilisateur exécute une boucle infinie qui vérifie une file d'attente pour voir s'il ya des événements de l'interface utilisateur en cours. Dans le cas d'Android, ce concept est formalisé sous la forme d'un **looper**.

Le looper traite une MessageQueue (file de messages) qui contient les messages à être expédiés. La tâche réelle de la gestion de la file d'attente se fait par un gestionnaire qui est responsable de la manipulation (ajout, suppression, dispatching) des messages dans la file de messages.



Pour le thread principal, un Looper est configuré par le système. Cependant, vous pouvez également associer un Looper avec votre propre thread.

Notez que le Looper peut être associé avec un seul tread et que l'association ne peut pas être changée.

La façon dont Android assure ceci (l'association ne peut pas être changée) est en mettant le looper sur le « thread local storage » du thread et ne pas exposer un constructeur pour le Looper.

Alors bien évidemment, on ne doit pas exécuter une opération de blocage sur le thread d'interface utilisateur. Si vous le faites, les messages de la file d'attente de cessent de se traiter et votre application ne répondait plus.

Les Services

Comme mentionné précédemment, un service est un type de composant qui est utilisé pour réaliser une opération d'arrière-plan, et ne fournit pas une interface utilisateur. Un composant d'une application peut démarrer un service, et même si l'utilisateur passe de l'application, le service continue à fonctionner.

Un cas d'utilisation typiques est de jouer de la musique et télécharger un fichier. Android assure pour un processus d'exécution d'un service qu'il ne soit pas tué autant que possible (voir cycle de vie du processus ci-dessus).

Notez que par défaut, un service s'exécute sur le thread UI principal dans le même processus. Cela signifie que si on effectue une opération de blocage, votre interface ne répondait plus, parce que le thread ne serait pas disponible pour exécuter l'activité. Pour contourner cela, vous devez lancer un thread de travail à l'intérieur du service.