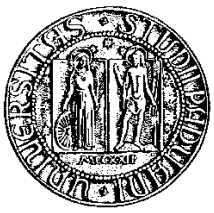

Università degli Studi di Padova

Facoltà di Ingegneria

Corso di Laurea Triennale in Ingegneria Informatica

tesi di laurea

Android 4.0: overview, novità e funzionamento del sistesma operativo targato Google

Relatore: Ing. Michele Moro

Laureando: Michael Masiero

26 marzo 2012

Autore: Michael Masiero

Indice

Capitolo 1

Introduzione

Il seguente elaborato nasce dallo stimolo di voler approfondire uno spicchio di un panorama del mondo dell'informatica che sempre più prepotentemente sta crescendo e sta prendendo parte alla vita quotidiana delle persone: quello degli smartphone, e più precisamente della loro 'anima', il sistema operativo. Nel corso degli anni le aziende si sono avvicinate sempre più a fornire un rapporto personale fra telefono cellulare e end-user, inserendo nuove funzionalità e raccogliendo ed elaborando dati personali per un'interazione sempre più radicata.
Nel panorama attuale i sistemi operativi più affermati sono sicuramente Android, IOs e di recente Windows Phone, realtà ereditata da versioni di Windows CE precedenti alla nascita di Android. Di seguito ci si focalizzerà solo sul primo dei tre citati.

1.1 I sistemi embedded

I sistemi embedded sono sistemi con caratteristiche particolari, di seguito vengono elencate le principali:

- sono pensati per fare una o poche funzioni limitate;
- hanno risorse limitate;
- vi sono dei limiti legati al tempo reale;
- sono programmati con linguaggi di basso livello.

A fronte di queste caratteristiche, si potrebbe pensare che gli smartphone non siano sistemi embedded poichè:

- non svolgono una o poche funzioni: le applicazioni che vi girano possono svolgere un'infinità di compiti, gli unici limiti sono la creatività del programmatore e le funzionalità messe a disposizione dall'hardware;

- il concetto di risorse limitate è 'relativo': i moderni smartphone possono essere considerati come dei piccoli computer, essendo provvisti ormai di processori potenti (di recente dual e quad core), di molta memoria RAM (fino a 2GB) e di molta memoria fisica (32GB) e molto altro;
- limiti tempo reale: vengono svolte operazioni dinamiche in tempo reale, come nel caso del garbage collector;
- non sono usati solo linguaggi di basso livello: i moderni smartphone sono programmati con linguaggi di alto livello come C++ e Java.

L'architettura porta comunque con sè delle implicazioni che distinguono un sistema operativo per computer da un sistema operativo per smartphone e tablet:

- solo un'applicazione alla volta è visibile ed interagisce con l'utente. Il sistema operativo gestisce gli scambi da un'applicazione ad un'altra;
- il ciclo di vita di un'applicazione è ottimizzato per il sistema embedded;
- il sistema operativo può decidere di terminare un'applicazione che non interagisce con l'utente per liberare risorse;
- in Android non è presente alcun page file.

All'utente non importa inoltre se le applicazioni in background hanno delle limitazioni, anche perché non conosce nulla dello stato delle applicazioni in pausa, ferme o distrutte. Le app sono viste tutte come disponibili (ovvero in running). E' quindi compito dello sviluppatore mantenere l'illusione del multitasking (per esempio nel caso di pagine web caricate in background, durante l'ascolto di musica, ecc). La piattaforma mette così a disposizione API apposite per lo sviluppatore.

1.2 Che cos'è Android e nascita del sistema

Android è un sistema operativo open source per dispositivi mobili, attualmente basato su kernel 2.6 di Linux.

Figura 1.1: Logo Android.

L'idea alla base di Android nasce per mano di un'azienda della California nel lontano 2003 dal nome Android Inc., fondata da Andy Rubin, Rich Miner, Nick Sears e Chris White. All'epoca il mercato degli antenati dei moderni smartphone era dominato da telefoni prodotti da Palm o montanti Windows Mobile. L'idea di Rubin, partendo da questo scenario, era quella di creare un sistema operativo aperto basato su Linux, conforme agli standard e con un'interfaccia semplice, immediata e funzionale. Inoltre doveva poter mettere a disposizione degli sviluppatori validi strumenti per la creazione di nuove applicazioni e sopratutto l'adozione doveva essere gratuita.
La svolta nello sviluppo di Android arriva nel luglio del 2005 quando Google acquista Android Inc., trasformandola nella Google Mobile Division con a capo sempre Andy Rubin. L'acquisizione fornì a Rubin i fondi e gli strumenti per portare avanti il suo progetto. Il passo successivo fu la fondazione nel novembre dello stesso anno della Open Handset Alliance (OHA), unione, con a capo Google, di operatori telefonici, produttori di dispositivi mobili, produttori di semiconduttori, compagnie di sviluppo software e di commercializzazione, avente come scopo quello di creare standard aperti per dispositivi mobili.
Nel novembre del 2007 L'Open Handset Alliance viene istituita ufficialmente e presenta il sistema operativo Android, seguito dal rilascio del primo Software Development Kit (SDK) per gli sviluppatori (il quale includeva gli strumenti di sviluppo, le librerie, un emulatore del dispositivo, la documentazione e alcuni progetti di esempio). Nel giugno dello stesso anno era arrivato sul mercato il primo iPhone di Apple che ha rivoluzionato il modo di concepire gli smartphone. In molti si aspettavano che Google rispondesse con un proprio smartphone, per cui la sorpresa fu grande quando presentò un intero ecosistema, un sistema operativo capace di funzionare su molti dispositivi diversi tra loro.

Figura 1.2: Logo Open Handset Alliance.

Le versioni del sistema sono indicate a livello ufficiale da un numero di versione seguito sempre da un nome in codice per tradizione ispirato a prodotti dolciari sempre in ordine alfabetico.
Di seguito, sotto forma di pseudo roadmap, vengono riportate le versioni sviluppate fin'ora:

- Android 1.0 e 1.1: prime due versioni del sistema rilasciate la prima a fine 2007 e la seconda ad inizio 2009. Con il primo update non vennerò però introdotte novità significative.
- Android 1.5 - Cupcake (aprile 2009): primo major update, introduce importanti novità sopratutto a livello utente, come una miglior esperienza con la tastiera a schermo, animazioni fra schermate e miglioramenti a lato multimediale.

Figura 1.3: Logo Android Cupcake.

- Android 1.6 - Donut (settembre 2009): corregge errori di riavvio del sistema, migliora ulteriormente il lato delle interfaccie multimendiali, vi è una miglior integrazione della ricerca e si introduce il supporto a diverse

risoluzioni. Inoltre è la prima versione che offre una navigazione definita come turn-by-turn legata alle funzionalità Gps e di navigazione.

Figura 1.4: Logo Android Donut.

- Android 2.0,2.1 - Eclair (ottobre 2009): aggiunge funzionalità come supporto per bluetooth 2.1, flash e zoom digitale per la fotocamera, multitouch, live wallpapers e supporto a HTML5.

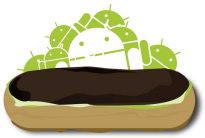

Figura 1.5: Logo Android Eclair.

- Android 2.2 - Froyo (maggio 2010): principalmente aumenta la velocità del sistema adottando il sistema di compilazione Javascript 'just in time' ripreso dal browser Chrome. Viene inoltre aumentato il supporto del browser, implementato il supporto al Flash Player, il supporto al tethering USB e WiFi e la possibilità di installare app nella memoria esterna (microSD).

Figura 1.6: Logo Android Froyo.

- Android 2.3 - Gingerbread (dicembre 2010): l'interfaccia è stata modificata e viene definita da Google come la versione di Android più veloce di sempre. Sono migliorati la tastiera, la funzione di copia e incolla e la gestione della batteria.

Figura 1.7: Logo Android Gingerbread.

- Android 3.0,3.2 - Honeycomb (febbraio 2011): versione creata appositamente per dispositivi tablet con miglioramenti sia a livello di sistema sia specifici per il diverso ambiente di utilizzo.

Figura 1.8: Logo Android Honeycomb.

- Android 4.0 - Ice Cream Sandwich (ottobre 2011): ultima versione rilasciata del sistema, introduce importanti novità su cui ci si soffermerà succesivamente, ma in particolare nasce come fusione delle due versioni precedenti così da poter funzionare in ogni tipo di supporto (smartphone o tablet).

Figura 1.9: Logo Android Ice Cream Sandwich.

- Android 5.0 - Jelly Bean (non ancora rilasciata): per il momento vi sono solo rumors, i principali legati al voler collegare l'ambiente Mobile con quello Desktop.

Figura 1.10: Logo Android JellyBean.

1.3 Overview sulla struttura del sistema

La struttura del sistema è sostanzilmente uno stack di software, in cui sono inclusi il sistema operativo, applicazioni middleware (intermediarie fra app e componenti software) e applicazioni chiave (fondamentali per il corretto funzionamento delle funzionalità di base del sistema offerte all'utente finale).
Fra le features troviamo:

- Application framework, che consente il riutilizzo e la sostituzione dei componenti;
- Dalvik virtual machine, macchina virtuale ottimizzata per dispositivi mobili;
- Browser integrato basato sul motore open source WebKit;

- Grafica ottimizzata, supportata da un'apposita libreria grafica per il 2D, la grafica 3D è basata sulla specifica OpenGL ES 1.0;
- SQLite, motore per la memorizzazione di strutture di dati;
- Supporto multimediale per i più comuni formati audio, video e di immagine;
- Telefonia GSM (legata all'hardware);
- Bluetooth, EDGE, 3G e WiFi (legati all'hardware);
- Fotocamera, GPS, bussola e accelerometro (legati all'hardware);
- Ricco ambiente di sviluppo che include un emulatore, strumenti per il debug, profili adatti alla gestione della memoria e delle prestazioni e un plugin per l'IDE Eclipse.

Nel seguente schema vengono mostrati i componenti principali del sistema Android, descritti poi di seguito.

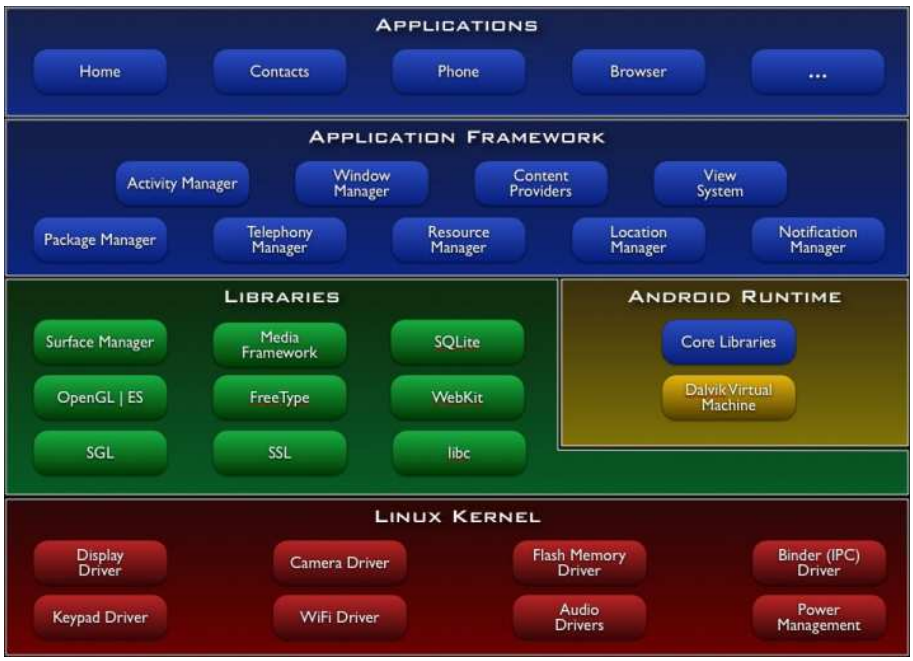

Figura 1.11: Architettura del sistema operativo Android.

Procedendo dall'alto verso il basso (essendo uno stack) si passa da livello utente (Applicazioni) fino a basso livello (Kernel Linux).

1.3.1 Applicazioni

Qui si trovano i software per l'interazione con l'utente. Il sistema fornisce un insieme di 'core applications' per lo svolgimento delle funzionalità di base alle quali è possibile aggiungerne di nuove (via market o altri canali). Ogni applicazione è scritta in Java.

1.3.2 Application Framework

Piattaforma per lo sviluppo, la quale offre infinite possibilità per la creazione di nuove applicazioni da parte degli sviluppatori, i quali possono accedere a tutte le funzionalità del sistema (dall'hardware, alle informazioni memorizzate, ai servizi in background e altro ancora). La piattaforma dà anche pieno accesso alle API utilizzate dal sistema. L'architettura è così strutturata per poter semplificare il riutilizzo di ogni singolo componente (ricondivisibile e utilizzabile fra applicazioni diverse).
Oltre alle varie funzionalità è fornito anche un insieme di servizi, il quale include un insieme di View per costruire le applicazioni, Content Providers (per comunicare fra applicazioni), Resource Manager (per l'accesso alle risorse grafiche), Notification Manager (per la gestione dei messaggi e delle notifiche a video) e Activity Manager (per la gestione dei cicli di vita delle applicazioni).

1.3.3 Librerie

Android include un insieme di librerie C/C++ utilizzate da diversi componenti del sistema. Le varie funzionalità sono fornite agli sviluppatori attraverso l'Application Framework. Fra le librerie troviamo una libreria C che equivale all'implementazione della standard libc ma adattata a sistemi Linux embedded, librerie per la multimedialità, librerie per la gestione della grafica, una libreria dedicata per il web browser (LibWebCore), una libreria per la gestione di database relazionali (SQLite) e altre ancora.

1.3.4 Android Runtime

Insieme di librerie derivanti direttamente da Java e funzionalità legate all'avvio delle applicazioni. Ogni applicazione che viene lanciata crea un proprio processo con associata un'istanza della Dalvik virtual machine (macchina virtuale creata appositamente per la gestione efficente di più applicazioni contemporanee). Quest'ultima fa riferimento al kernel sottostante per funzionalità di basso livello come gestione di thread e della memoria.

1.3.5 Kernel Linux

La piattaforma si basa attualmente sulla versione 2.6 del kernel Linux, usato per i servizi di base del sistema, quali sicurezza, gestione della memoria, gestione dei processi, di rete, e dei driver. Esso funziona anche come un livello di astrazione tra l'hardware e il resto dello stack software.

Capitolo 2

Linux

Linux è il più famoso sistema operativo open source. Ha origine da UNIX nei lontani anni settanta, il suo sviluppo concreto inizia però vent'anni più tardi grazie a Linus Torvalds. Da progetto di un singolo si evolve rapidamente a livello mondiale, coinvolgendo migliaia di sviluppatori. Uno dei più importanti passi per il sistema fu la scelta di adottare la licenza GPL. Sotto questa licenza il sistema venne protetto da fini commerciali, beneficiando così anche di un ampio sviluppo a livello utente.

L'architettura di un sistema operativo basato su GNU/Linux può essere vista come grandi blocchi:

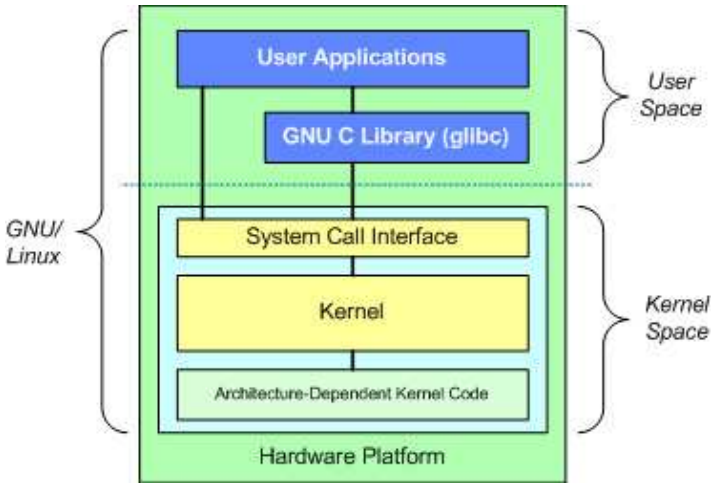

Figura 2.1: Struttura del sistema operativo Linux.

La parte superiore è il livello utente (o applicazione). In questo livello sono eseguite le applicazioni del sistema operativo. E' presente inoltre una libreria

che fornisce un'interfaccia al sistema per interagire con il livello sottostante, ovvero il livello kernel (kernel space). Quest'ultimo è diviso internamente in tre ulteriori sotto livelli:

- System Call Interface (SCI), nel quale sono implementate funzioni di base per l'interazione con la macchina;
- Kernel, ovvero dove risiede il codice del nucleo del sistema, indipendente dalla macchina;
- Board Support Package (BSP), livello in cui è presente codice specifico legato alla piattaforma.

2.1 Introduzione al Kernel Linux

Come visto precedentemente, Android ha come fondamenta la versione 2.6 del kernel Linux. Il kernel è la parte centrale, il nucleo, la più importante dell'intero sistema.
Concretamente il kernel è un pezzo di software che provvede a definire un livello di collegamento fra hardware e applicazioni di alto livello. Il kernel mette così a disposizione delle applicazioni che girano nella macchina i suoi servizi attraverso quelle che vengono definite chiamate di sistema (come per esempio le chiamate open, read, write, close, ecc.). Viene così creato un livello di astrazione sulle funzioni dell'hardware.

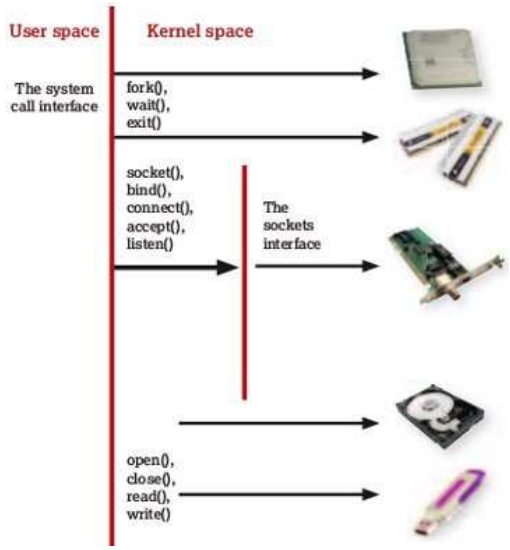

Figura 2.2: Interazione fra kernel e dispositivi hardware.

Il filesystem è una delle principali astrazioni fornite dal kernel, ovvero la riunione delle funzionalità legate all'accesso a file su supporto generico aventi un particolare formato.
Il kernel è responsabile inoltre di funzionalità non direttamente visibili, come lo scheduling dei processi. Oltre allo scheduling il kernel provvede anche alla gestione della memoria. Anche in questo caso vi è l'illusione che ogni programma avviato abbia della memoria dedicata. In realtà la memoria fisica è condivisa con gli altri processi e in base alle esigenze vengono operate varie scelte. Inoltre gestendo la memoria il kernel si preoccuopa di prevenire l'accesso da parte di altri processi ad un indirizzo già assegnato ad un programma, così da mantenere e preservare l'integrità del sistema.
Nel kernel sono implementanti anche protocolli di rete come l'IP, il TCP e l'UDP, fondamentali per la comunicazione fra più macchine o con la rete. Anche in questo caso vi è l'illusione che le connessioni che si stabiliscono siano senza interruzioni, quando invece a basso livello non vi è mai nulla di permanente.
Per come è organizzato, il kernel Linux tiene traccia dell'ID (un numero identificativo) dell'utente e del gruppo a cui appartiene, così da sfruttare i permessi associati per prendere delle decisioni durante l'esecuzione dei processo. Il controllo degli accessi è quindi strettamente legato alla sicurezza del sistema.
Infine il kernel fornisce un ampio insieme di moduli che 'conoscono' come gestire operazioni di basso livello legate strettamente all'hardware.

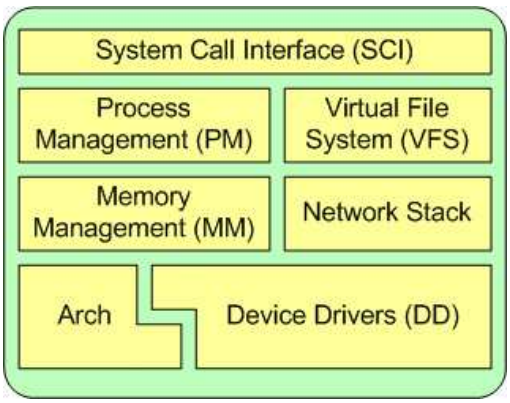

Figura 2.3: Struttura del kernel Linux.

2.2 Kernel e modularità

A differenza delle prime versioni del kernel Linux che adottavano una struttura di tipo assolutamente monolitico, ovvero ogni singola parte era collegata alle altre staticamente in un unico file eseguibile, i moderni kernel sono modulari.

La modularità permette di suddividere le varie funzionalità in più moduli caricabili dinamicamente. Questo consente al nucleo del kernel, caricato all'avvio, di rimanere piccolo e con la possibilità di caricare e sostituire i moduli anche da avviato senza che sia necessario un riavvio.
Linux nella maggiorparte dei casi gestisce senza alcun bisogno di interventi esterni i vari moduli, vi sono però delle istruzioni che permettono l'ispezione e la gestione manuale.
L'istruzione *lsmod* per esempio permette di avere una lista dei moduli attualmente caricati.

```
# lsmod
pcspkr              4224  0
hci_usb            18204  2
psmouse            38920  0
bluetooth          55908  7 rfcomm,l2cap,hci_usb
yenta_socket       27532  5
rsrc_nonstatic     14080  1 yenta_socket
isofs              36284  0
```

Figura 2.4: Esempio comando lsmod.

Per ogni modulo viene visualizzato il nome, la dimensione, il numero attuale di utilizzi e una lista delle dipendenze. Il numero di utilizzi è importante poiché non è possibile levare moduli attivi. Se si tentasse di rimuovere un modulo in uso sarà lo stesso kernel a ritornare un messaggio di errore segnalante l'impossibilità di portare a termine la richiesta.

2.3 Approfondimento sullo scheduling

Lo scheduling dei processi ricopre un ruolo fondamentale nell'ottimizzazione e nel buon e corretto funzionamento di un sistema.

Lo scheduler ha subito nel tempo un'evoluzione, a pari passo con lo sviluppo dei computer e dell'aumento di versione del kernel. Con il kernel 2.6 (quello su cui si basa attualmente Android) si hanno avuto due diversi tipi di scheduler. In un primo momento, per risolvere problematiche legate al kernel precedente e per motivi di varia ottimizzazione, Linux ha adottato uno scheduler O(1), nel quale i processi erano organizzati in liste a seconda della priorità. Erano inoltre presenti due code, una per i processi attivi e una per i processi sospesi, nelle quali erano mantenute le liste di priorità. In ogni istante, data una priorità, si poteva accedere con una complessità O(1) alla lista corrispodente. Vi erano implementati anche diverse ottimizzazioni di tipo euristico per la determinazione di quali processi erano di tipo I/O bound o CPU bound. Il codice però per

effettuare questi calcoli era di difficile gestione e mancava di una sostanza algoritmica. Si ebbe così un'evoluzione (nella stessa versione del kernel) arrivando ad uno scheduler di tipo CFS (Completely Fair Scheduler).
L'idea principale dietro a questo scheduler è di mantenere equilibrato il tempo di occupazione della CPU concesso ai processi. Quando non vi è questo bilanciamento, si sceglie di concedere ai processi non bilanciati del tempo per essere eseguiti. Per determinare il bilanciamento, lo scheduler mantiene memorizzato il tempo dato ad un processo in quello che viene chiamato 'virtual runtime'. Più è piccola questa struttura (e quindi meno tempo è stato concesso al processo), più avrà priorità per il controllo del processore. Viene implementato inoltre il concetto di equità anche per i processi in attesa (per esempio per quelli in attesa di I/O), concedendogli una giusta porzione di tempo di esecuzione quando ne hanno bisogno.
A differenza di ciò che accadeva con i precedenti tipi di scheduler (i processi erano mantenuti in code d'esecuzione), il CFS mantiene i processi in un albero rosso-nero con ordinamento temporale. La struttura rosso-nera porta con sè importanti proprietà: è auto-bilanciata (non ci sono percorsi nell'albero più lunghi di due elementi rispetto agli altri) e ogni operazione sull'albero ha un costo di O(log n) (con n nodi dell'albero), quindi inserimenti e cancellazioni sono molto rapidi e efficienti.

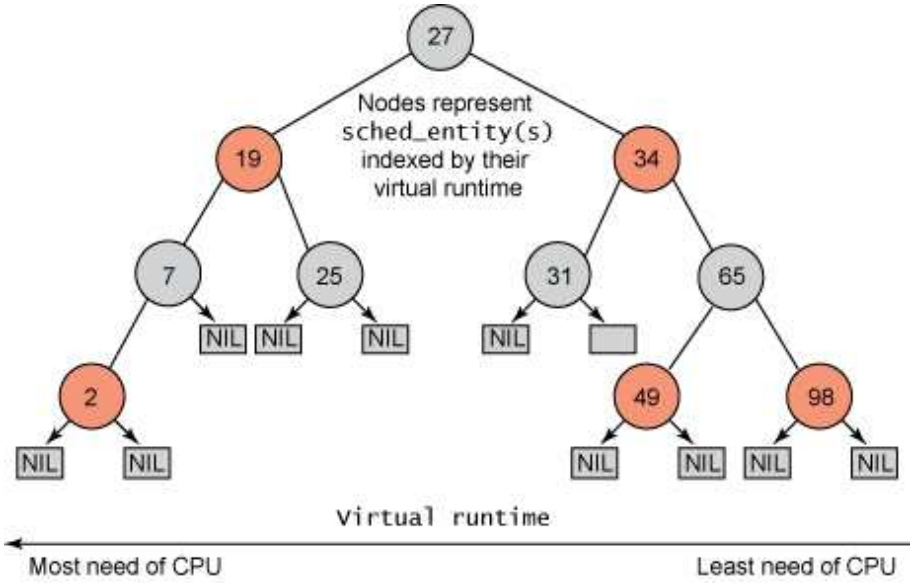

Figura 2.5: Struttura dati utilizzata nello scheduling CFS.

Processi con maggior bisogno di utilizzare la CPU (minor virtual runtime) sono mantenuti nella parte sinistra dell'albero, viceversa processi con meno bisogno

(maggior virtual runtime) sono mantenuti nella parte destra. Lo scheduler, seguendo il principio di equità, prende di volta in volta il processo più a sinistra. Concluso l'instante di tempo concessogli, viene sommato tale tempo al virtual runtime del processo (che verrà reinserito nell'albero se non si è ancora completato). Estraendo un nodo, l'albero, per auto-bilanciarsi, tenderà a spostare via via i nodi più a destra verso sinistra, così da mantenere equità nelle successive scelte dei processi che dovranno essere eseguiti.
A differenza di altre versioni di scheduler, il CFS non utilizza il concetto di priorità direttamente, bensì un fattore di 'decadimento' legato al tempo che un processo può rimanere in esecuzione. Processi a bassa priorità hanno un più alto valore di tale fattore, viceversa per quelli ad alta priorità. Questo significa che il tempo che un processo può rimanere in esecuzione diminuisce più velocemente per un processo a bassa priorità piuttosto che per uno ad alta priorità.
Un'ulteriore novità introdotta con il CFS è il concetto di gruppo di scheduling. Anche in questo caso l'obiettivo è l'equità ed è particolarmente indicato nei casi di processi che ne avviano molti altri. In questi casi invece di trattare separatamente ogni processo li si prende in gruppo. Il processo che li ha lanciati condivide con il gruppo (gestito in una gerarchia) il proprio virtual runtime, mentre ogni singolo processo continua a mantenere il proprio virtual runtime in modo indipendente.

2.4 /proc filesystem

Il kernel di Linux, fra le altre cose, mette a disposizione una grande quantità di informazioni a cui si può accedere attraverso il /proc filesystem.
Tutto ciò che si trova all'interno di /proc riesce a fornire una vista a svariate strutture di dati interne al kernel. Un esempio è dato dalla presenza della lista dei moduli attualmente caricati (/proc/modules) oppure dalle informazioni sullo stato attuale della memoria virtuale (/proc/meminfo). Di particolare interesse sono i file in /proc/sys dato che, oltre a fornire utili informazioni sul sistema (per esempio se è attiva o meno una determinata funzionalità), c'è la possibilità di scrivervi direttamente (per esempio per modificare lo stato di un elemento).

Capitolo 3

Il sistema operativo Android

Il funzionamento del sistema operativo ruota quasi completamente attorno alle applicazioni in esecuzione e alla loro interazione con l'utente. Di seguito si approfondiranno proprio questi aspetti.

3.1 Le applicazioni

Le applicazioni di sistema non utilizzano API 'speciali': come per le applicazioni sviluppate dagli utenti, sfruttano le stesse API pubbliche. Il sistema operativo offre delle applicazioni standard che possono essere sostituite su scelta dell'utente in caso di bisogno. Questo significa che si possono sostituire intere applicazioni come il browser di sistema o anche semplicemente avere una tastiera diversa.
Questa struttura aperta e modulare è una degli elementi che contraddistinguono l'ecosistema Android.
In Android si possono distinguere due tipi di applicazioni:

- le 'normali' applicazioni, ovvero le applicazioni che funzionano a schermo intero;
- i widget, applicazioni che occupano solo una piccola e definita porzione della home (il 'desktop' di sistema).

Ogni applicazione lanciata ha associato un proprio processo (nel kernel Linux) con l'ID dell'utente. Come visto precedentemente, Linux stesso si preoccupa di assicurare la protezione di dati sensibili o l'accesso a componenti privilegiati del sistema.
Ogni applicazione inoltre ha la propria copia di Dalvik (la macchina virtuale). Questa scelta evita, nel caso di errori e malfunzionamenti di un'applicazione, che essi si propaghino verso altre.

Le applicazioni in Android possono essere formate da 4 tipologie di 'componenti':

- Activity: componenti costituiti da una singola vista, possiedono un'interfaccia utente;
- Servizi: componenti eseguiti in background per svolgere operazioni lunghe e/o non interattive (per esempio il playback della musica), non possiedono un'interfaccia utente;
- Content provider: componenenti di supporto alla comunicazione fra applicazioni (gestiscono dati da condividere), non possiedono un'interfaccia utente;
- Broadcast receiver: componenti la cui funzione è volta a 'reagire' a eventi di sistema che influenzano l'intero dispositivo (es.: batteria scarica) , non possiedono un'interfaccia utente.

Ogni applicazione è costituita da un insieme di questi quattro componenti, ognuno con un ciclo di vita separato dagli altri. Inoltre ogni applicazione può chiedere ad Android di lanciare un altro componente di un'altra applicazione (per esempio si può chiedere di scattare una foto appoggiandosi alla funzionalità offerta dall'applicazione della fotocamera).
Per ogni applicazione è importante sottolineare che non esiste un 'primo' componente, di conseguenza per quanto è possibile definire un'activity principale non vi sono specifici entry point.
Ogni applicazione è provvista di un file di manifesto (manifest.xml), dove vengono dichiarati i componenti della stessa, i requisiti software e hardware per il suo utilizzo e la lista dei permessi che si richiedono per eseguire il codice.

3.1.1 Intent

Le activity, i servizi e i broadcast receiver vengono tutti attivati da intent. Un intent è un messaggio asincrono che richiede l'avvio di una determinata azione. Attraverso una richiesta tramite intent è possibile chiedere al sistema un componente specifico o anche solo un tipo di componente. In quest'ultimo caso sarà compito del sistema associare alla richiesta un componente disponibile in grado di soddisfarla.
Ci sono più meccanismi per inviare intent ad ogni tipo di componente:

- un oggetto Intent viene passato a *Context.startActivity()* o *Activity.startActivityForResult()*, due metodi per il passaggio ad una nuova activity per compiere una nuova operazione. E' possibile ricevere un oggetto Intent anche come risultato di *Activity.setResult()*, il metodo che serve a ritornare le informazioni richieste da *Activity.startActivityForResult()* quando l'activity richiamata viene terminata;

- un oggetto Intent viene passato a *Context.startService()* per lanciare un nuovo servizio o inviare nuove istruzioni ad un servizio già attivo. Analogamente a *Activity.startActivityForResult()* è possibile creare un collegamento via intent fra un componente e un servizio attraverso *Context.bindService()* (creabile anche sul momento se non già preesistente);

- oggetti Intent passati un qualsiasi metodo di broadcast (come *Context.sendBroadcast()*, *Context.sendOrderedBroadcast()*, o *Context.sendStickyBroadcast()*) e recapitati ai destinatari interessati.

Tutte queste meccaniche avvengono in modo trasparente grazie ad Android, il quale si preoccupa di trovare il corretto ricevente per ogni richiesta, istanziandolo nel caso non esistesse già. E' inoltre importante sottolineare che non vi sono sovrapposizioni fra queste metodologie: un intent lanciato in broadcast verrà ricevuto solo da broadcast receiver e non da activity o servizi. Analogamente accade lo stesso nel caso di activity o servizi.

Un oggetto Intent è un aggregato di informazioni. Contiene sia informazioni di interesse per il componente ricevente (legate per esempio all'azione da effettuare) sia informazioni di interesse per il sistema Android (come la categoria del componente a cui è indirizzato l'intent e le istruzioni su come lanciare una determinata activity).
Concretamente è formato da:

- nome del component a cui è indirizzato l'intent (campo opzionale);

- azione che verrà svolta (identificata da una stringa);

- URI dei dati su cui verrà svolta l'azione e il tipo di dati;

- categoria dell'intent, ovvero una stringa con informazioni legate al tipo di componente a cui è indirizzato l'intent;

- un campo extra per l'inserimento di informazioni aggiuntive;

- flag di diverso tipo, molti sfruttati da Android per sapere come agire in fronte all'intent.

Gli Intent possono inoltre essere divisi in due gruppi:

- intent espliciti, ovvero indirizzati direttamente verso un componente identificato dal proprio nome. Vengono utilizzati solitamente in modo trasparente all'utente per le comunicazioni interne ad un'applicazione;

- intent impliciti, ovvero quelli per i quali non viene specificato un componente (il campo nome rimane vuoto). Vengono spesso utilizzati per attivare componenti di altre applicazioni.

Nel primo caso Android provvede solo a recapitare al componente esplicitato l'intent, nel secondo caso si preoccupa di trovare il migliore o i migliori componenti a cui indirizzarlo, basandosi su dei filtri associati ai componenti stessi. Nel caso in cui un componente non è provvisto di appositi filtri potrà ricevere solo intent espliciti.

3.1.2 Activity

Un'activity è un componente delle applicazioni che fornisce un'interfaccia tramite la quale l'utente può interagire. Un'applicazione invece è costituita da più activity, assieme ad altri componenti. Solitamente, in un'applicazione, un'activity è specificata come principale e viene visualizzata a schermo appena l'utente lancia l'applicazione per la prima volta.
Avendo bisogno di un'interfaccia visuale, ogni classe implementata (in cui viene estesa Activity) deve o creare sul momento gli elementi visuali su cui poggia o (scelta più comune) deve aver associato un file .xml nel quale sono dichiarati gli elementi grafici che saranno visualizzati. Il file in questione è inserito fra le risorse dell'applicazione.
Ogni applicazione è provvista di un file di 'manifesto' nel quale sono riportate direttive per fornire al sistema informazioni per il corretto funzionamento della stessa. Tra le informazioni da riportare deve essere presente una lista composta dalle activity utilizzate dall'applicazione.

Ogni activity segue un determinato ciclo vitale e in ogni istante è caratterizzata da uno dei seguenti stati:

- active/resumed (ovvero running), l'activity è visibile e può ricevere input dall'utente;
- paused, parzialmente visibile a causa di un'altra activity in foreground, non può ricevere input dall'utente (e.s.: è in attesa a causa di un alert dialog presente al di sopra di essa);
- stopped, non visibile (rimane in Background).

In qualsiasi istante un'activity non in esecuzione (stato paused o stopped) può essere costretta da Android a liberare risorse (chiamando il suo metodo finish() o terminando direttamente il processo). Ci si può riferire a questo stato come destroyed. Nel caso venisse riaperta dovrà essere ricreata completamente.
Ogni singolo cambio di stato è dovuto a decisioni prese dall'utente o dal sistema operativo. Le transizioni di stato non sono evitabili da un'activity, è possibile solo agire in funzione di essi prendendo decisioni appropriate. Ad ogni transizione corrispondono uno o più metodi mediante i quali è possibile compiere operazioni aggiuntive.

Di seguito viene riportato un esempio di activity dove vengono dichiarati i metodi fondamentali legati al suo ciclo di vita:

```
public class ExampleActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // The activity is being created.
    }
    @Override
    protected void onStart() {
        super.onStart();
        // The activity is about to become visible.
    }
    @Override
    protected void onResume() {
        super.onResume();
        // The activity has become visible (it is now "resumed").
    }
    @Override
    protected void onPause() {
        super.onPause();
        // Another activity is taking focus
        //(this activity is about to be "paused").
    }
    @Override
    protected void onStop() {
        super.onStop();
        // The activity is no longer visible (it is now "stopped")
    }
    @Override
    protected void onDestroy() {
        super.onDestroy();
        // The activity is about to be destroyed.
    }
}
```

Implementandoli è possibile gestire tre diversi cicli concatenati al ciclo di vita dell'activity:

- l'intero tempo di vita, compreso fra il lancio di *onCreate()* (dove avviene il setup dello stato globale e dell'ambiente) e di *onDestroy()* (dove vengono rilasciate le risorse utilizzate);

- il tempo di vita nel quale l'activity è visibile, compreso fra i metodi *onStart()* e *onStop()*. In questo frangente l'activity è visualizzata a schermo dall'utente, il quale potrà interagirvi;

- il tempo di vita nel quale l'activity è in foreground (è sopra ad ogni altra activity e riceve direttamente input dall'utente), compreso fra i metodi *onResume()* e *onPause()*. Poiché l'activity può frequentemente entrare e uscire dallo stato di foreground, il codice all'interno dei metodi di questo sotto ciclo non dovrà impegnare troppo il sistema per evitare rallentamenti a livello utente.

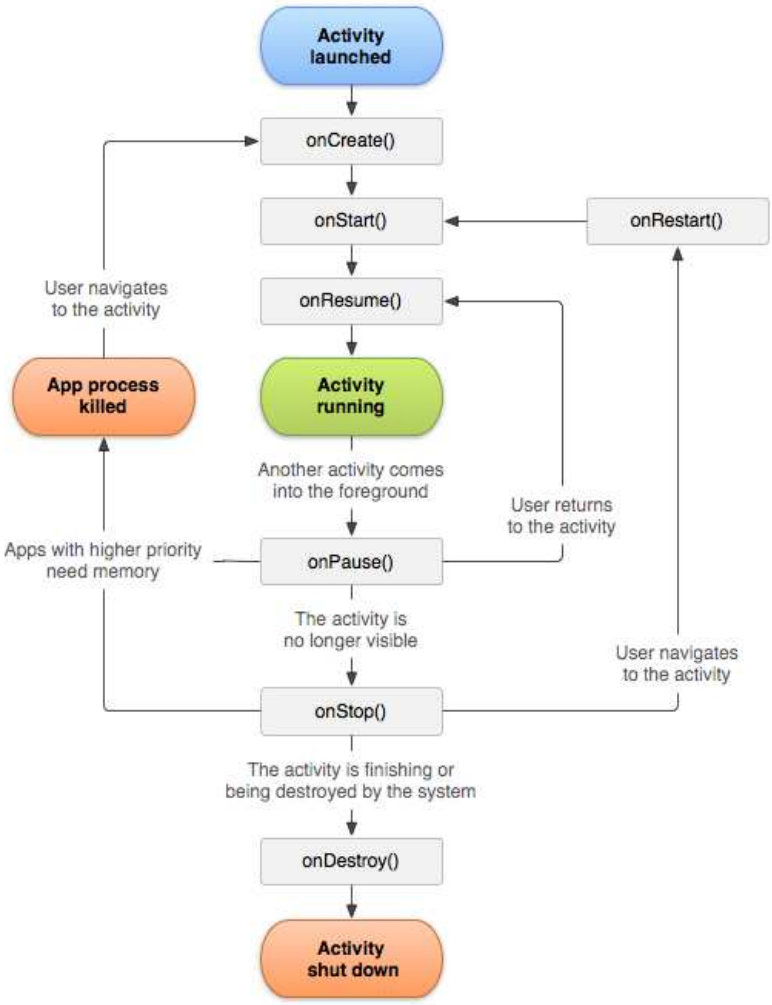

Figura 3.1: Ciclo di vita di un'Activity.

In Android viene associata un'activity ad ogni orientazione (portrait e landscape). Di conseguenza ogni cambiamento fra una e l'altra causa un transizione degli stati paused, stopped e destroyed per quella attiva, mentre viene istanziata quella nuova.
E' importante sottolineare che, in base a quanto detto, dato che non è garantito che durante il ciclo di vita di un'activity vengano richiamati i metodi *onStop()* e *onDestroy()*, è buona cosa salvare lo stato ogni qual volta assume lo stato di paused (e quindi viene richiamato *onPause()*).
Lo stato di un'activity è l'insieme delle informazioni degli elementi di cui è composta in un dato istante. Quando un'activity ritorna in foreground, il suo stato non è garantito permanga nel tempo: per questo è necessario implementare un ulteriore metodo (*onSaveIstanceState()*) per essere sicuri che venga ripristinata correttamente.
Se implementato, il sistema lo richiama prima che l'activity entri in uno stato in cui potrebbe essere distrutta. Attraverso metodi specifici per il salvataggio di diversi formati di dati, si inseriscono all'interno di un oggetto Bundle (contenitore) le informazioni che si vogliono preservare. Nel caso poi il sistema liberasse memoria, nel momento in cui l'utente riapre l'activity viene ricreata come la si era lasciata alla precedente chiusura, passando l'oggetto Bundle sia a *onCreate()* sia a *onRestoreInstanceState()*. Durante la prima apertura tale oggetto sarà nullo. Vi sono comunque alcuni elementi che vengono salvati automaticamente dall'implementazione di default del metodo, come modifiche visibili all'interfaccia utente.

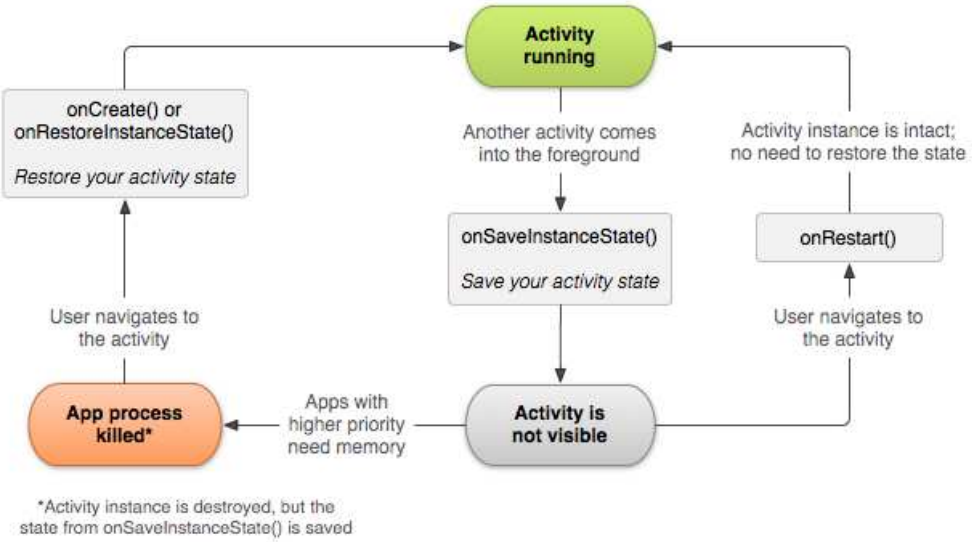

Figura 3.2: Schema di ripristino dello stato di un'activity.

3.1.3 Servizi

Un servizio è un componente di un'applicazione che si occupa di svolgere in background operazioni che richiedono lunghi tempi di esecuzione. Per sua natura non è provvisto di interfaccia utente.
Una qualsiasi applicazione può lanciare un servizio che continuerà a rimanere in esecuzione anche nel momento in cui l'utente deciderà di eseguirne un'altra. In aggiunta, un qualsiasi componente può legarsi ad un servizio per interagirvi e eventualmente comunicare con altri processi (IPC).
Un servizio può assumere due forme:

- 'Standard' o Unbounded, ovvero quando un componente dell'applicazione (come un'activity) lo lancia attraverso *startService()*. Da avviato, un servizio può rimanere in background per un tempo indefinito, anche nel momento in cui chi l'ha lanciato viene distrutto. Di solito le funzioni di servizi di questo tipo sono singole, ovvero ogni servizio svolge una singola operazione e non ritorna nessun valore al chiamante. Sono servizi standard, per esempio, quelli che si occupano dell'upload o del download di file. Ad operazione conclusa è lo stesso servizio a fermarsi.

- Bound, ovvero quando un componente lo lega a se stesso attraverso *bindService()*. Questo legame offre un'interfaccia client-server che permette al componente di interagirvi, mandando per esempio richieste, acquisendo informazioni o comunicando con altri processi. Il tempo di vita di un servizio di questo tipo è strettamente legato alla vita del componente che l'ha lanciato: alla distruzione del componente viene distrutto anche il servizio. Allo stesso servizio inoltre possono legarsi più componenti e in questo caso continuerà a rimanere in background fintanto che almeno uno vi è ancora collegato.

Indipendentemente da chi ha avviato il servizio, esso può essere utilizzato da un qualsiasi componente dell'applicazione (e anche da altre applicazioni). Come nel caso di lancio di un'activity, un servizio può essere utilizzato attraverso il lancio di un Intent. E' possibile comunque dichiarare il servizio di tipo privato, bloccando così l'accesso da parte di altre applicazioni.
E' da porre particolare attenzione al fatto che un servizio avviato è legato al thread principale del processo dell'applicazione, non crea (se non è specificato) un proprio thread separato e non viene avviato in un altro processo. Questo significa che se il servizio è di tipo CPU bound, è preferibile crearvi un nuovo thread associato per diminuire il rischio che l'applicazione non risponda ad alto livello, consentendo così sempre un'interazione fluida col dispositivo da parte dell'utente.
Per poter utilizzare un servizio, dopo averlo dichiarato opportunatamente nel manifest.xml e aver implementato in una classe apposita la classe Service, può

essere creato via *onCreate()* dopo aver istanziato la classe stessa. In base poi alla forma che si vuole fargli assumere, 'normale' o 'bind', si richiamano rispettivamente *onStartCommand()* o *onBind()*. Un servizio, nel primo caso, viene poi distrutto quando si ferma o è il client a fermarlo. Nel secondo invece la distruzione avviene quando non è più legato alcun client.

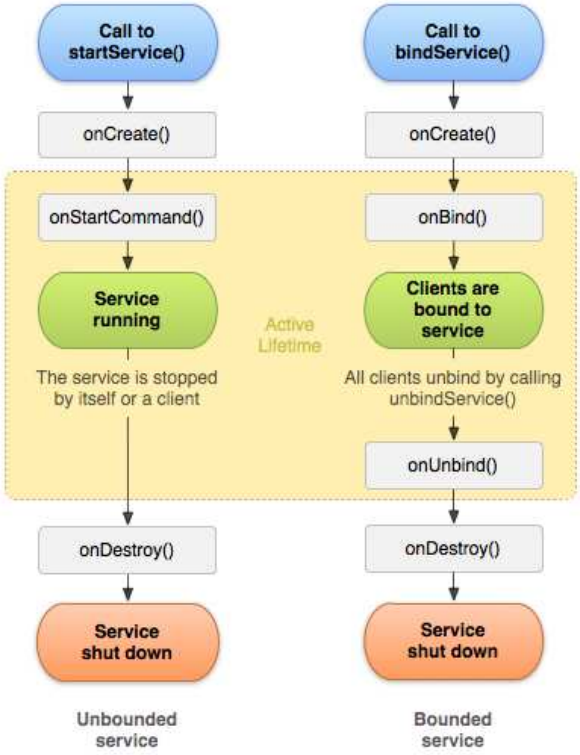

Figura 3.3: Ciclo di vita di un Servizio.

3.1.4 Content Provider

I content provider si occupano di gestire l'accesso a insiemi di dati strutturati, provvedendo ad occuparsi anche dell' incapsulamento e della sicurezza degli stessi. Rappresentano inoltre l'interfaccia standard per la condivisione di dati fra più processi.
Nel momento in cui vi è la necessità di accedere a dati gestiti da un content provider, viene richiamato un oggetto di tipo ContentResolver all'interno del Context dell'applicazione stessa per la comunicazione con il provider (istanza di una classe che implementa ContentProvider). Ricevuta la richiesta dal client, l'oggetto provider esegue l'azione richiesta e ritorna il risultato dell'operazione. Un'applicazione che non necessita di condividere i propri dati con altre non ha

bisogno di implementare un proprio content provider, è necessario però implementarlo nel caso si eseguano trasferimenti di dati complessi o file.
Il sistema operativo mette comunque a disposizione un set di provider di base come per la gestione di oggetti multimediali (audio, video, immagini), sfruttabili da ogni applicazione.
I dati sono organizzati in tabelle (come in un database). Un content provider può gestire più tabelle. Per ogni tabella è presente una colonna ID (chiave primaria), la quale associa ad ogni record un numero unico identificativo.
Android mette a disposizione due classi astratte di base (appartenenti al pacchetto android.content) per l'uso dei content provider:

- ContentProvider, per l'incapsulamento dei dati;
- ContentResolver, per accedere ai dati memorizzati attraverso la classe precedente.

Per identificare i dati, nello specifico le tabelle e le loro righe, viene sfruttata la classe Uri (del pacchetto android.net). URI è l'acronimo di Uniform Resource Identifier e non è altro che una stringa riconosciuta da standard internazionali che segue uno schema fisso per identificare delle risorse.
Ogni stringa URI segue il seguente schema:

- nome dello schema (content nel caso si utilizzino content provider);
- '://';
- autorità (in questo caso il nome del content provider);
- path di dove si trova la risorsa.

content://com.example.prova.AddressBookProvider/People/56

In questo caso, dopo il nome dello schema (content), troviamo l'autorità (nel nostro esempio la path completa di dove risiede il content provider com.example.prova.AddressBookProvider) e il path della risorsa. Quest'ultima, nel caso d'esempio, risiede nella tabella People alla riga 56.
All'interno di Android sono definiti URI standard legati a funzionalità di sistema (per esempio per la lista contatti).
Ogni content provider deve essere dichiarato nel file manifest.xml per poter essere utilizzato, sarà poi comunque il sistema ad istanziare l'oggetto. Ogni istanza inoltre può comunicare con più content resolver. Le modifiche e le query ai dati vengono fatte solo attraverso di essi. Ogni content resolver non viene istanziato, ma ottenuto invocando il medoto *getContentResolver()*, opportunamente implementato.

3.1.5 Broadcast receiver

Un broadcast receiver è un componente che si attiva rispondendo ad un annuncio fatto in broadcast dal sistema stesso. Viene gestito attraverso la classe astratta BroadcastReceiver e in aggiunta attraverso Intent (per la comunicazione in broadcast).
Come per i componenti prececedenti, un broadcast receiver deve essere registrato nel file manifest.xml dell'applicazione. Inoltre, la registrazione avviene anche dinamicamente durante l'esecuzione dell'applicazione attraverso il metodo *registerReceiver()*.
Il ciclo di vita di un broadcast receiver può essere riassunto nei seguenti punti (dopo ovviamente essere stato implementato estendendo la classe BroadcastReceiver):

- il receiver viene registrato;
- quando viene mandato un intent in broadcast che può attivare il ricevitore, viene richiamato il metodo *onReceive(Context context,Intent intent)*, anche nel caso in cui è contenuto in un processo non in esecuzione;
- quando *onReceive()* si conclude, l'oggetto ricevitore non è più attivo (e il corrispondente processo può venire fermato).

Al metodo *onReceive()* Android impone di completare la sua esecuzione entro 10 secondi. Allo scadere del timer il ricevitore viene considerato bloccato e il processo può venire distrutto. Un broadcast receiver è quindi valido solo per il tempo della durata di tale metodo. Come componente non dispone di un'interfaccia utente, gli è consentito però comunicare con l'utente attraverso la creazione di notifiche nella status bar.

3.2 Multitasking

Prima di focalizzarsi su quello che è il multitasking per Android, è importante dare la definizione generale di task (che assume, come vedremo a breve, un significato diverso per Android) e thread. Si definisce processo o task un'istanza di un programma attualmente avviato dal sistema operativo. Processi diversi non condividono risorse.
Si definisce thread un'unità di un processamento gestita dallo scheduler del sistema operativo.
Un task può contenere più thread che condividono le stesse risorse.
Multitasking (o più propriamente multithreading) è l'abilità di gestire più thread concorrentemente. Il numero di thread può essere superiore al numero di unità d'esecuzione (parte del processore formata da ALU e registri). Il numero di unità di esecuzione è legato al numero di processori (o core) presenti su un dispositivo.

Il multitasking utilizzato da Android è di tipo preemptive, ovvero i thread vengono rimossi forzatamente dall'unità di esecuzione quando lo decide il sistema operativo. Questo accade per esempio quando un thread ad altra priorità necessità di risorse bloccate da altri thread a più bassa priorità. Ogni thread riceve una porzione del tempo d'esecuzione proporzionale alla sua 'importanza'. A differenza però di quanto accade nei moderni sistemi operativi per computer, in Android non possono essere eseguite più applicazioni (ovvero interi programmi) contemporaneamente (tradotto per sistemi mobili, si intendono più applicazioni visibili a schermo contemporaneamente). Quello che invece è possibile fare è gestire più thread concorrentemente.
In Android è possibile quindi avere solo un'applicazione in foreground. Durante l'esecuzione non è necessario che richieda permessi specifici per compiere determinate azioni, a differenza di quel che accade per le applicazioni in background. Le activity messe in background (ovvero, facendo riferimento al ciclo di vita, dopo che il metodo *onPause()* è stato chiamato) continuano ad essere schedulate, in qualsiasi momento però possono essere fermate o distrutte da Android senza alcun preavviso. Di conseguenza non vi è la certezza che un' activity in tale stato completi sempre il suo lavoro.
Queste limitazioni sono legate strettamente alla natura dell'ambiente: ogni applicazione utilizza delle risorse (come la memoria, la batteria nel caso sia in esecuzione, ecc.) che sono limitate nel caso dei sistemi embedded. Se si necessitano delle risorse non è possibile toglierle all'applicazione in foreground, in quanto l'utente lo noterebbe immediatamente. E' invece più ragionevole richiederle a tutte le altre applicazioni.
Come visto in precedenza, in background possono essere eseguiti solo broadcast receiver e servizi. Nel primo caso all'applicazione viene consentito di rimanere in esecuzione in background per un breve periodo di tempo in conseguenza ad un evento che si è verificato. Nel secondo caso invece l'applicazione può rimanere in background per un periodo di tempo illimitato.
In Android il multitasking è strettamente legato al ciclo di vita delle activity e alle operazioni che compie l'utente. Ogni activity può a sua volta iniziarne un'altra (anche di un'altra applicazione), così da svolgere diverse azioni. Indipendentemente dal fatto che le activity provengano da più applicazioni, Android mantiene l'insieme di activity in uno stesso task. Si definisce task (per Android) una collezione di activity con il quale l'utente interagisce quando deve compiere una determinata operazione. Questo significa che un task è composto da tutte le schermate, di una o più applicazioni, che l'utente attraversa mentre sta utilizzando un'applicazione. La schermata Home è il punto di partenza per la maggior parte dei task. Quando l'utente decide di lanciare un'applicazione (tocca l'icona collegamento corrispondente), il task corrispondente viene messo in foreground. Nel caso non ne esistano per l'applicazione appena avviata, viene creato un nuovo task e l'activity principale dell'applicazione inserita come radice di uno stack, chiamato 'back stack'. Ogni qual volta un'activity viene lanciata

viene subito inserita in cima a tale stack dal sistema. Chi l'ha lanciata (l'activity precedente) viene preservata nello stack ma viene messa in stato di 'stop'. Lo stack, in quanto tale, è organizzato secondo una logica LIFO (Last In First Out). Quando l'activity sul top dello stack viene conclusa dall'utente premendo il tasto Back, essa viene estratta (pop) e distrutta e il controllo dello schermo torna all'activity che ora si trova in cima allo stack stesso. Nel caso l'utente continui a premere il tasto Back, le activity presenti nel task verranno volta per volta distrutte e lo stack liberato. Rimasto vuoto, l'utente viene riportato nella schermata di Home e il task smette di esistere.

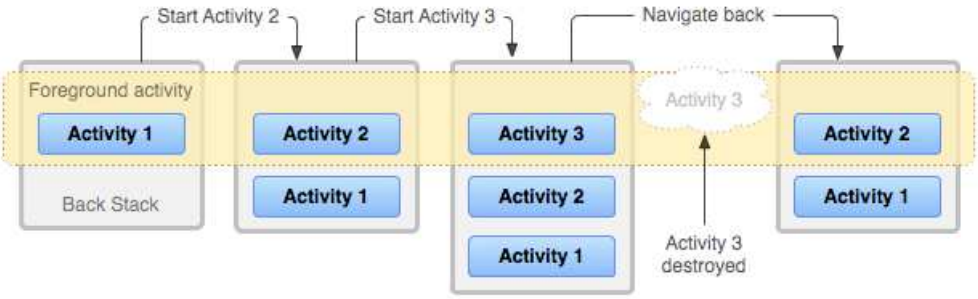

Figura 3.4: Esempio del funzionamento del backstack di un task.

Un task, essendo un insieme di più activity, viene messo in background assieme all'intero stack quando l'utente inizia un nuovo task o ritorna all'Home screen attraverso l'Home button. Rimanendo in background, tutte le activity dello stack rimangono in stato di stopped ma lo stack rimane intatto. In background possono essere mantenuti più task, se però l'utente ne inserisce molti, Android potrebbe essere costretto a rimuoverne qualcuno per recuperare memoria, causando così la perdita dello stato dell'activity. Per riprendere un task, l'utente può tenere premuto il tasto Home e selezionare dalla lista che apparirà l'applicazione (il task relativo) da riprendere o rilanciare direttamente la stessa applicazione
Dato che in uno stack le posizioni non vengono mai riordinate, se un'applicazione lo permette è possibile avere nello stesso stack più istanze della stessa activity.
Essendo Android fondato su kernel Linux, è utile focalizzare l'attenzione anche su quello che accade a basso livello: quando un componente di un'applicazione viene lanciato e non vi sono altri componenti di tale applicazione in esecuzione, Android avvia un nuovo processo di Linux per l'applicazione con un singolo thread di esecuzione. Di default, tutti i componenti della stessa applicazione sono eseguiti nello stesso processo e thread (definito come thread 'principale'). Se un componente di un'applicazione viene lanciato ed esiste già un processo per tale applicazione (esiste già almeno un altro componente in esecuzione), esso viene

eseguito nello stesso processo e utilizza lo stesso thread di esecuzione. E' comunque possibile decidere di far eseguire diversi componenti di un'applicazione in processi separati o creare più thread per lo stesso processo.

3.3 Gestione della memoria

Android è provvisto di un proprio gestore interno della memoria che provvede, sempre in modo trasparente per l'utente, ad assegnare e liberare porzioni di memoria alle applicazioni del dispositivo.
In particolare, svolge un ruolo fondamentale nel momento in cui le risorse iniziano a scarseggiare. Essendo la fluidità a livello utente una priorità, il gestore (in questo caso nelle vesti di task killer) provvede a iniziare a chiudere tutte le applicazioni classificate come a più bassa priorità. Normalmente vengono considerate a priorità minore le applicazioni che vengono usate meno e/o che non sono fondamentali per il corretto funzionamento del dispositivo.
Per assegnare la priorità ad un processo, Android fa affidamento ad un servizio (ActivityManagerService.java), il quale tiene traccia della loro importanza assegnando loro un valore sulla base di più fattori. Questo meccanismo è strettamente legato al kernel Linux su cui si fonda Android. L'implementazione è però 'personalizzata' in quanto sistema embedded. Ogni processo ha così associata una variabile 'oom_adj' (dove oom sta per out of memory), la quale più assume un valore alto e più è facile che il dato processo venga selezionato per essere terminato.
Una particolarità di Android sta nel fatto che è possibile, attraverso un modulo, impostare più valori di soglia: se la memoria libera scende sotto un valore X il sistema può rimuovere processi con valore di oom_adj maggiore di Y, se continua a scendere fino ad un valore W potranno poi essere rimossi processi con oom_adj maggiore di Z (con Z minore di Y), e via dicendo.
Android raggruppa i processi in esecuzione in sei diverse categorie, di seguito riportate in ordine di importanza (decrescente):

- foreground_app;
- visible_app;
- secondary_server;
- hidden_app;
- content_provider;
- empty_app;

Per ogni categoria Android impone una diversa soglia di memoria libera. le soglie si trovano nel file /sys/module/lowmemorykiller/parameters/minfree e sono di default rispettivamente 1536, 2048, 4096, 5120, 5632, 6144 (l'ordine potrebbe

variare in relazione al tipo di firmware del dispositivo). Questi valori corrispondono a pagine di memoria di 4 kB l'una. Tramite privilegi di root (maggiori dettagli più avanti) è possibile modificare tale file scegliendo opportunamente la configurazione desiderata.

Le applicazioni in esecuzione dispongono di una cache di rapido accesso dove inserire dati temporanei (memoria heap dinamica). Più è grande la dimensione di questa memoria e più il sistema sarà veloce. Si avranno inoltre dei benefici in termini di consumi energetici, dato che l'accesso alla memoria esterna è più dispendioso per la batteria. Normalmente Android sceglie di dedicare 24 Mb per questa memoria. Nel momento in cui tale memoria sarà piena, il sistema è costretto ad iniziare un'operazione di garbage-collection, tramite la quale viene liberata parte della heap. Un appropriato valore della memoria consente quindi di ridurre la frequenza di questa operazione e di conseguenza di occupare meno la CPU e il numero di operazioni di I/O fra memoria flash e RAM (e come sempre quindi vengono ridotti anche i consumi). Anche in questo caso è possibile settare un valore diverso da quello di default attraverso i privilegi di root.

Capitolo 4

Android 4.0

La versione Ice Cream Sandwich porta con sè numerose novità, sia a livello utente che per gli sviluppatori. La novità più importante è legata al fatto che nasce come unione e miglioramento delle due versioni di firmware precedenti (2.3 e 3.0). Ricordiamo che la prima è adatta esclusivamente a smarthphone, la seconda invece è ottimizzata per tablet. Di conseguenza questa versione è pensata per essere montata su dispositivi di qualsiasi natura, unificando le diverse realtà.

4.1 Novità a livello utente

Android 4.0 fa tesoro delle esperienze prencedenti e delle opinioni degli utilizzatori per creare un ambiente sempre più confortevole, intuitivo e alla portata di tutti. In modo particolare ci si è focalizzati nel rendere più semplici e immediate funzioni già presenti, magari implementate in modo ancora 'oscuro' per l'utente medio. Inoltre molte sono le funzionalità che sono diventate native, poichè già presenti ma solo sotto forma di applicazione di terze parti.
Con questa versione scompaiono i tasti fisici dai terminali e il quarto tasto di ricerca, lasciando spazio a tre tasti direttamente nella parte bassa dello schermo del dispositivo.
Di seguito viene riportata una lista delle principali features introdotte e/o aggiornate con questa versione:

- Interfaccia utente completamente ridefinita, volta a rendere più visibili le azioni che si svolgono più comunenmente. Sono ridefinite anche le animazioni e funzionalità legate all'interazione. E' inoltre introdotto un nuovo font chiamato Roboto pensato per una visione più definita in schermi ad alta risoluzione;

- Multitasking reso di più facile accesso, più semplice e visuale. Precedentemente si poteva accedere solo ad una lista delle applicazioni (visualizzate come icone) aperte di recente (pressione lunga Home button, metodo poco

immediato), ora per mostrare la nuova lista è stato inserito un tasto dedicato. E' inoltre possibile interagirvi, scorrendola per riaprire o rimuovere le applicazioni utilizzate di recente. Per ognuna di esse è presente una thumbnail che mostra lo stato nella quale è stata lasciata;

- Miglioramento delle notifiche e dell'interazione con queste ultime. In base alle dimensioni dello schermo del terminale la loro posizione cambia per sfruttarlo nel miglior modo possibile;
- Cartelle nella schermata Home e icona di accesso ai preferiti. La prima feature era già disponibile attraverso applicazioni specifiche, offre la possibilità di aggregare shortcut in un'unica cartella. La seconda invece è pensata per device con uno schermo più piccolo per favorire l'accesso ad applicazioni e funzionalità di uso comune senza dover navigare per i vari menù;
- Widget ridimensionabili. Anche questa feature era già disponibile, permette di riorganizzare nel migliore dei modi l'Home screen;
- Nuova lock screen (la schermata che appare alla riaccensione dello schermo a terminale bloccato). Sono stati inseriti shortcut veloci per diverse funzionalità (chiamate, messaggi, camera) e la possibilità di accedere alle notifiche senza entrare nella Home;
- Miglioramento della tastiera nativa del sistema. Ogni versione ormai porta con sè questo tipo di aggiornamenti, Android 4.0 punta a rendere l'inserimento del testo più veloce e accurato, aumentando la precisione della correzione degli errori durante la digitazione;
- Miglioramento dell'acquisizione audio di un testo;
- Maggior integrazione con aspetti 'social' e di condivisione di contenuti;
- Miglioramento e aggiunta di funzionalità per la fotocamera;
- Galleria contenuti multimendiali completamente rivisitata, con la possibilità di modificare nativamente per esempio un'immagine;
- Possibilità di modificare in tempo reale video che si stanno registrando con effetti di vario tipo;
- Possibilità di scattare degli screenshoot (prima era possibile solo attraverso applicazioni terze e root del dispositivo);
- Miglioramento della web experience, in particolare con miglioramenti ed ottimizzazione della navigazione via web browser e aumento delle funzionalità email;

- Inserimento di nuove funzionalità (Android Beam) per la condivisione istantanea di applicazioni, contatti, contenuti multimediali e molto altro fra due terminali.

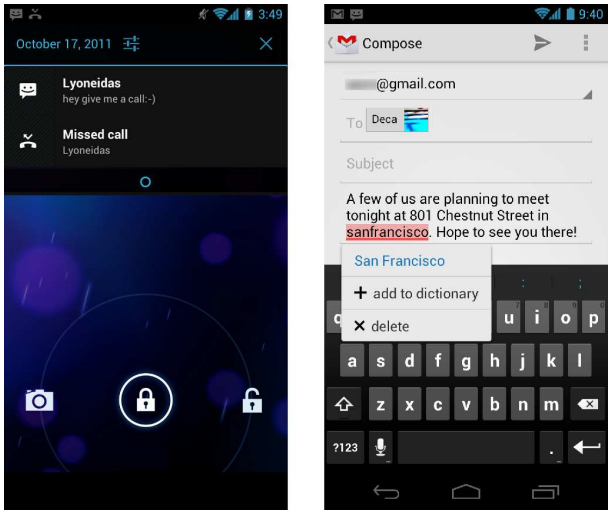

Figura 4.1: Schermate d'esempio di Android 4.0 - 1.

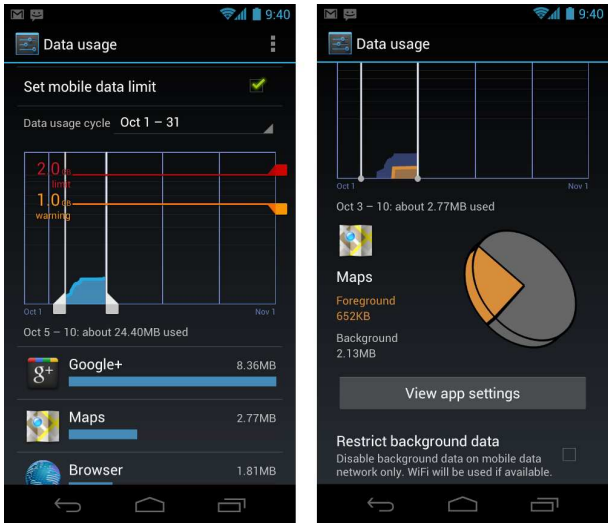

Figura 4.2: Schermate d'esempio di Android 4.0 - 2.

4.2 Novità per gli sviluppatori

- Interfaccia unificata per smartphone e tablet: Android 4.0 possiede un'unica struttura per l'interfaccia utente. L'Unified UI framework include tutti gli elementi dell'interfaccia già presenti in Android 3.x e le sue API (assieme alle novità della versione). Per gli sviluppatori questa scelta si traduce come semplificazione del codice e delle risorse utilizzate, e uno sviluppo diretto su tutti i dispositivi con Android, indipendentemente dalla loro natura.

- Comunicazione e condivisione: con Android 4.0 l'integrazione delle caratteristiche sociali e di condivisione è estesa a tutte le applicazioni di sistema fornite all'utente. E' così messa a disposizione un insieme di API di supporto come Social API, Calendar API, Visual Voicemail API, Android Beam e Modular sharing widget.

- Low-level streaming multimedia: si stabilisce un diretto ed efficiente percorso per il controllo completo sui media data, prima che vengano passati al sistema perché vengano visualizzati. Viene, anche in questo caso, introdotta una nuova API, implementata sugli stessi servizi di base della piattaforma esistente, OpenSL ES API, così che gli sviluppatori possano usufruire di entrambe le API in caso di bisogno.

- Nuove caratteristiche per la fotocamera: gli sviluppatori possono usufruire di una serie di nuove funzionalità offerte dal sistema, anche per un miglior controllo della periferica da parte delle applicazioni che ne faranno uso.

- Media effects per trasformazione di video e immagini: vengono forniti filtri di trasformazione per l'applicazione di effetti e per la modifica di immagini. Ogni trasformazione viene poi elaborata dalla GPU, di conseguenza l'elaborazione dei frame diviene molto rapida.

- Audio remote controls: Android 4.0 aggiunge una nuova API di controllo audio remoto che consente alle applicazioni multimediali di integrarsi con i controlli di riproduzione che vengono mostrati in una visualizzazione remota. Le applicazioni multimediali possono essere integrate con un remote control per la riproduzione di musica che è incluso nella schermata di sblocco, permettendo agli utenti di controllare la selezione e la riproduzione di canzoni senza dover aprire l'applicazione musicale.

- Nuovi codec e contenitori multimediali.

- Nuovi tipi di connettività: viene aggiunto il supporto per connessione WiFi 'diretta' e Bluetooth Health Device Profile (HDP).

- Nuovi UI component: Android 4.0, oltre ad aggiungere nuovi elementi grafici, impone che il device supporti l'accelerazione hardware per il disegno

2D. Questo garantisce agli sviluppatori di mantenere prestazioni ottimali su schermi ad alta risoluzione anche utilizzando diverse combinazioni di componenti.

- Nuovi metodi di inserimento e servizi testuali: vengono aggiunte novità legate al supporto testuale, come la possibilità da parte delle applicazioni di interrogare i servizi disponibili, come dizionari, spell checker, suggerimenti e correzioni.

- Aumento accessibilità API: vengono aggiunte nuove caratteristiche di accessibilità alle API consentendo agli sviluppatori di migliorare l'esperienza utente nelle proprie applicazioni, sopratutto nel caso di dispositivi senza pulsanti hardware. In casi particolari è anche possibile possibile facilitare la navigazione, migliorare il feedback e avere interfacce utente più ricche.

- Uso più efficente della rete e monitoraggio: in Android 4.0, gli utenti possono visualizzare la quantità di dati in rete delle applicazioni in esecuzione, eventualmente impostando secondo necessità i limiti per l'utilizzo dei dati per tipo di rete o disabilitando l'utilizzo dei dati per applicazioni specifiche. In questo contesto, gli sviluppatori devono progettare le loro applicazioni per funzionare in modo efficiente e seguire buone pratiche per la verifica della connessione di rete. Android 4.0 fornisce inoltre network API per consentire alle applicazioni di raggiungere questi obiettivi. Come gli utenti passano da reti o limiti stabiliti su dati di rete, la piattaforma permette di interrogare le richieste di tipo di connessione e la disponibilità. Gli sviluppatori possono utilizzare queste informazioni per gestire in modo dinamico le richieste di rete per garantire una migliore esperienza per gli utenti (fornendogli eventualmente i dati di utilizzo per mezzo di un nuovo system Intent).

- Management delle credenziali: Android 4.0 rende più facile, per le applicazioni, gestire l'autenticazione e le sessioni protette. Una nuova API e lo storage criptato permettono alle applicazioni di memorizzare e recuperare le chiavi private e le loro catene di certificati corrispondenti. Qualsiasi applicazione può utilizzare l'API per installare e memorizzare i certificati utente e CA in modo sicuro.

- Address Space Layout Randomization: Android 4.0 fornisce questa nuova modalità (ASLR) per proteggere applicazioni di sistema e di terze parti dallo sfruttamento a causa di problemi di gestione della memoria.

- VPN client API: gli sviluppatori possono ora creare o estendere proprie soluzioni VPN alla piattaforma utilizzando un nuovo VPN API (fondata sulla memorizzazione sicura delle credenziali). Con il permesso dell'utente, le applicazioni possono configurare gli indirizzi e le regole di routing, gestire i pacchetti in uscita e in entrata, e stabilire tunnel sicuri su un

server remoto. Le imprese possono inoltre usufruire di un client VPN standard integrato nella piattaforma che fornisce l'accesso ai protocolli L2TP e IPSec.

Capitolo 5

Approfondimenti

5.1 Android oggi e frammentazione

Android, come sistema operativo per smartphone e tablet, negli ultimi anni ha subito una crescita esponenziale: attualmente si contano 850000 attivazioni giornaliere di nuovi terminali che lo montano. Ci sono però delle questioni chiave sulle quali è necessario porre particolare attenzione.
Essendo un sistema nato per essere montato non su un solo o pochi hardware, vi sono diverse problematiche che si devono affrontare. I primi a subire le conseguenze di questa scelta sono sicuramente gli utenti finali, costretti - anche magari per scelte di mercato - ad avere aggiornamenti con il contagocce o dopo molto tempo dal rilascio del sorgente ufficiale. Le case di produzione, oltre a dover adattare e ottimizzare il sistema per i propri terminali, modificano o aggiungono delle funzionalità al sistema stesso. Ci si trova così con terminali poco o non più supportati anche solo dopo un anno dal rilascio ufficiale, o con aggiornamenti pendenti a mesi dal rilascio della nuova versione del sistema. La principale controparte di Android, rappresentata da IOs, anche se ha a che fare con un numero limitato di hardware, non è comunque esente, anche se in misura minore, da queste problematiche. I terminali Apple sono supportati sicuramente per un tempo più lungo, ciò però non dà comunque la sicurezza di avere un sistema sempre performante e completo di tutte le ultime funzionalità.
Parlando di numeri, attualmente si osserva una crescita molto lenta della distribuzione della versione 4.0, mentre la versione 2.3 rimane ancora la più diffusa, con oltre il 60% di share. Dal grafico (fig. 5.1) si può anche notare che, se da una parte la diffusione dell'ultima versione è ancora limitata, lentamente stanno scomparendo anche le versioni più vecchie.
La realtà della frammentazione non si riflette solo nell'ambito dell'esperienza con il sistema, ma anche nel momento in cui uno sviluppatore vuole creare una nuova applicazione. La scelta di un' API piuttosto che un'altra, di una determinata risoluzione dello schermo o altro ancora, dà luogo ad una miriade di diverse combinazioni, pagate poi in tempo speso, ad esempio per eseguire i vari test per

controllare che ogni dispositivo sul mercato funzioni correttamente. Uno degli obiettivi della versione 4.0, in fronte a queste problematiche, è quello di unificare l'ecosistema Android. Peccato però che come si è visto sopra, non sempre è possibile.

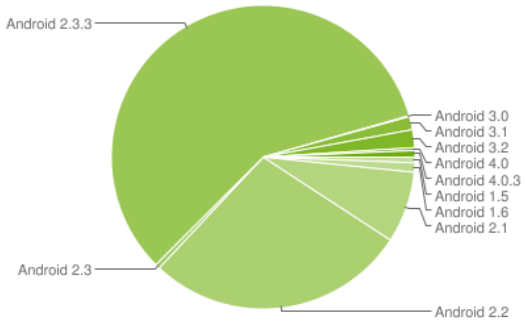

Figura 5.1: Distribuzione del mondo delle versioni di Android a marzo 2012.

Nel panorama dello sviluppo troviamo anche la questione della separazione dei kernel Linux e di Android. Oltre al non avanzamento di versione del kernel su cui si basa Android (fermo ancora al 2.6), rimangono ancora 'slegati'. Recenti annunci, anche da parte dello stesso Linus Torvalds, e patch per Android implementate nei kernel Linux, fanno presagire ad una probabile riunificazione dei due mondi, un po' come appena accaduto per il mondo smartphone e il mondo tablet.

5.2 Rooting

Fare il 'rooting' di un dispositivo significa ottenere i privilegi di 'superuser' all'interno del sistema Android. Aumentando i privilegi utente si aprono possibilità per incrementare le capacità di utilizzo del dispositivo, per esempio potendo installare un software personalizzato (ROM, ovvero Read Only Memory e sinonimo di custom software per Android) o semplicemente applicazioni che richiedono permessi particolari per essere avviate, installare un tema personalizzato, aumentare le performance e la durata della batteria e altro ancora.
Il termine root deriva direttamente dal mondo Linux, dove viene utilizzato per indicare chi ha privilegi da 'superuser', cioè la possibilità di cambiare e modificare una qualsiasi porzione di codice del sistema. Un dispositivo, per come viene venduto, fornisce all'utente privilegi solo di tipo guest. Questo viene fatto per evitare che, anche incosciamente, si tocchino parti del software con il rischio di malfunzionamenti vari. Di contro, utilizzando privilegi così bassi si chiude la porta a numerose funzionalità utili.

Di seguito vengono riportati quelli che sono i principali vantaggi e svantaggi di questa pratica.

5.2.1 Vantaggi del rooting

- Custom Software (ROM): disponendo dei privilegi necessari, è possibile installare sulla memoria ROM del dispositivo un software diverso, spesso basato su una versione pulita (stock) di Android. L'installazione di software diversi, oltre a semplici cambi di aspetto, può portare ad un aumento notevole delle prestazioni (e secondariamente delle funzionalità). Questa pratica diviene molto utile nel momento in cui, per esempio (causa anche motivi visti precedentemente con la frammentazione) sul terminale sia montata una vecchia versione di Android, non più supportata dalla casa produttrice. La comunità di sviluppatori sopperisce così a questa mancanza allungando la vita del dispositivo.

- Custom theme: senza dover necessariamente utilizzare una ROM è possibile modificare quasi ogni aspetto del sistema.

- Kernel, velocità e batteria: sia tramite ROM, che agendo direttamente, o tramite applicazioni è possibile agire su prestazioni e batteria (maggiori dettagli in un successivo approfondimento). Tramite inoltre dei tweak si può agire direttamente sul kernel (che ricordiamo essere il livello che provvede a gestire la comunicazione fra hardware e software).

- Baseband: il rooting di un device permette di aggiornare le baseband, ovvero ciò che controlla la trasmissione radio nel dispositivo. L'aggiornamento porta con sè migliorie sia nel segnale che nella qualità, per esempio nelle chiamate.

- Aggiornamenti della versione del firmware: oltre all'installazione di ROM è possibile aggiornare la versione di Android, sia per anticipare il rilascio ufficiale, sia nel caso non sia stata creata una versione apposita perchè il dispositivo non è più supportato.

- Backup del sistema: tra le funzionalità degne di nota troviamo sicuramente la possibiltà di creare copie di backup (con applicazioni apposite), di tutti i dati personali, anche relativi alle applicazioni installate. Questa funzionalità è molto utile nel momento in cui il device venga spesso aggiornato o vengano fatte diverse prove con le varie ROM disponibili.

5.2.2 Svantaggi del rooting

- Bricking: con il root un device può potenzialmente 'brickare', ovvero rischiare di scombinare le cose a livello software, tanto da far diventare il device inutilizzabile (come un mattone).

- Garanzia: acquisire i privilegi di root per molti produttori equivale a perdere la garanzia.

- Sicurezza: abbattendo delle barriere di sicurezza vi è la possibilità di installare malware con il rischio di intaccare il dispositivo. Il rischio rimane comunque potenziale, anche perchè spesso le applicazioni devono richiedere esplicitamente all'utente i permessi per essere eseguite e un utente informato sa bene che non deve abilitarle ad occhi chiusi.

La procedura di root è comunque reversibile, non è però confermato se comunque la casa produttrice possa o meno accorgersi delle modifiche dei privilegi.

5.3 Task killer

Come visto precedentemente, Android è già provvisto di un gestore di processi che in caso di bisogno funge anche da task killer. In rete (e nel Market) sono presenti comunque numerose applicazioni che svolgono le medesime funzioni e con le quali l'utente può decidere in qualsiasi momento di killare determinati processi. L'utilizzo scorretto e l'abuso di questo tipo di funzionalità porta con sè solo potenziali problemi di instabilità e scarse performance del terminale. Purtroppo spesso, per scarse informazioni o conoscenze, si tende ad aderire a svariati luoghi comuni:

- I task killer rendono il device più veloce: come appena detto, l'effetto prodotto è l'opposto di quello desiderato. Questo perchè, oltre a rendere il sistema potenzialmente instabile e scattoso, c'è sempre il rischio di chiudere processi condivisi o comunque utilizzati da applicazioni o dal sistema stesso, il quale si vede costretto, in caso di bisogno, a ricreare completamente l'attività, perdendo tempo e risorse (cicli di clock, batteria) e rischiando di creare rallentamenti anche a livello utente;

- I task killer aumentano la durata della batteria: come appena accennato capita l'esatto contrario. C'è spesso il rischio di chiudere un'applicazione che verrà riaperta dopo poco nuovamente dallo stesso Android, causando continue operazioni di I/O inutili e dispendiose;

- I task killer si sostituiscono ad ipotetici pulsanti di uscita dalle applicazioni non presenti: per la filosofia del sistema Android, le applicazioni non hanno bisogno di tasti di chiusura. L'utente stesso deve essere sollevato da qualsiasi preoccupazione di questo tipo, anche perchè il sistema svolge già ottimamente questo compito;

- I task killer non servono a nulla: sulla base di quanto appena detto si potrebbe essere portati a pensarlo, purtroppo però vi sono delle situazioni in cui sono indispensabili. Può capitare che per motivi di compatibilità o mal

programmazione, alcune applicazioni possano generare incompatibilità e malfunzionamenti. Un'azione mirata consente così di terminare la fonte del problema e ripristinare il normale funzionamento del terminale.

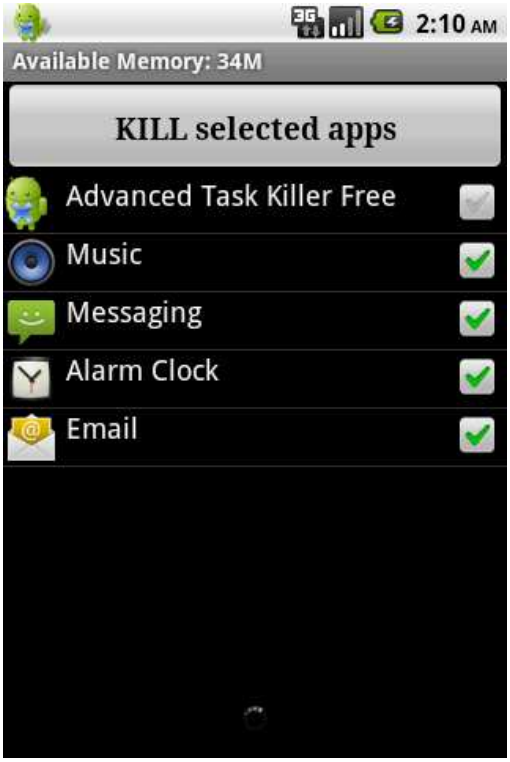

Figura 5.2: Schermata d'esempio di un task killer.

5.4 SetCPU

SetCPU è un'applicazione utilizzabile solo su dispositivi rootati che consente di gestire la frequenza della CPU del terminale, impostando degli intervalli di lavoro. Oltre ad essere uno strumento per ottimizzare le prestazioni, in alcuni terminali vi è la possibilità di overclock della CPU.

5.4.1 Benefici

La CPU consuma più energia maggiore è la frequenza di lavoro con cui opera. Un processore veloce permette al sistema di essere più veloce a sua volta, per contro la batteria si scaricherà più velocemente. Viceversa se la frequenza di lavoro è molto bassa, il terminale risulterà più lento (e magari scattoso), la batteria però durerà di più.

5.4.2 Profili

Grazie a questo tool è possibile controllare manualmente le frequenze della CPU, o eventualmente impostare dei profili tramite i quali è possibile modificare in automatico la velocità della CPU in base allo stato del sistema.
Le condizioni di trigger proposte dall'applicazione sono:

- Ricarica: il dispositivo è collegato ad una qualsiasi fonte di ricarica, anche nel caso di batteria completamente carica;
- Ricarica AC o USB: il dispositivo si sta caricando tramite presa a muro o via usb, anche nel caso di batteria completamente carica;
- Spegnimento dello schermo;
- Batteria: il livello di carica del dispositivo è sceso sotto una soglia impostata;
- Temperatura o temperatura CPU: la temperatura della batteria, del sistema o della CPU (solo se supportato) ha superato un valore limite precedentemente impostato;
- Orario: il funzionamento della CPU è legato all'ora della giornata.

Se nessuno dei requisiti è soddisfatto, SetCPU utilizzerà valori di default impostabili nella main activity.
Ad ogni profilo impostato, scelte le condizioni di funzionamento, viene associato un livello di priorità. Se le condizioni di più profili sono soddisfatte contemporaneamente, verrà scelto il profilo con più alta priorità.

5.4.3 Controllo scaling CPU

Ogni CPU lavora su due valori di frequenza. Grazie a SetCPU è possibile impostare come variano le frequenze nel passaggio dal valore minimo al valore massimo e viceversa. Anche in questo caso l'applicazione offre delle modalità preimpostate (non sempre tutte disponibili per ogni kernel):

- ondemand: quando la CPU raggiunge un determinato livello di carico (impostabile), la frequenza viene rapidamente scalata per sopperire alle richieste. Quando poi non diminuisce, il bisogno viene riabassata gradualmente;
- interactive: funzionamento simile a 'ondemand', azione più rapida nel momento del bisogno;
- conservative: funzionamento simile a 'ondemand', ma con un aumento più graduale nel momento del bisogno. Controllo meno rapido, ma meno dispendioso per la batteria;

- performance: la CPU funzionerà sempre al valore massimo di frequenza impostato. Questa impostazione è più efficente del semplice set della frequenza massima e minima nei profili, poiché si evita lo spreco di risorse nella scansione del carico della CPU;
- powersave: esatto opposto di 'performance', la CPU funzionerà sempre al valore minimo di frequenza impostato;
- smartass: viene impostato un profilo tramite il quale si mantiene la CPU al minimo quando è in idle (dispositivo con schermo spento).

Figura 5.3: Logo e schermata di SetCPU.

5.5 Esempio di implementazione thread in Android

L'esempio analizzato di seguito consiste essenzialmente in un'applicazione basata sul multi-threading, la quale disegnerà dei contatori su schermo in risposta a eventi generati dal tocco dello schermo da parte dell'utente. Ad ogni tocco corrisponderà un nuovo numero su schermo e l'aggiornamento dei numeri già presenti (i numeri fanno riferimento ad un contatore che si auto-aggiorna in background).
Il lavoro svolto dall'applicazione è frutto principalmente di una classe derivata da *View* che implementa *Runnable*. Grazie a questa implementazione può lanciare un proprio thread, semplicemente implementando il metodo *run()* e creando succesivamente una sua istanza.

Il lancio di più thread paralleli avviene tramite la chiamata del metodo *thread.start()*, il quale avvierà il codice presente in *run()*.

```
public void run(){
  while (!Thread.currentThread().isInterrupted()) {
    try {
      Thread.sleep(250);
    } catch (Exception e) {}
    mTicker++;
  }
}
```

Come si può osservare il ciclo generato è infinito, in una logica però di applicazione gestita dall'utente, esso può essere interrotto in qualsiasi momento.
L'esecuzione del thread causa l'incremento ogni 0.25 secondi di un contatore. Per ogni chiamata del metodo *onDraw()* viene estratto il valore corrente e riportato su schermo.

```
//lista import per il corretto funzionament dell'applicazione

public class Threading1Activity extends Activity {
    public static final int MENU_ABOUT = Menu.FIRST;

    private Threading1View mThreading1View;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        mThreading1View = new Threading1View(this);
        setContentView(mThreading1View);

        //lancio del thread in background per l'incremento
        //del contatore
        Thread thread = new Thread(mThreading1View);
        thread.start();
    }

    //view principale per il disegno dell'ambiente su schermo
    private static class Threading1View extends ImageView
                implements Runnable {
        private final Paint mPaint = new Paint();
        private List<Point> mCounterPositions = new
                                            ArrayList<Point>();
```

```
private int mTicker = 0;

public Threading1View(Context context) {
    super(context);
    setFocusable(true);

    mPaint.setTextSize(16);
    mPaint.setTextAlign(Paint.Align.CENTER);
}

//metodo richiamato alla pressione su schermo
@Override
protected void onDraw(Canvas canvas) {
    canvas.drawBitmap(mBmBackground, 0, 0, mPaint);

    if (mCounterPositions.size() > 0) {
        int counter = 0;
        for (Point position: mCounterPositions) {

            //disegno del valore del contatore nel cerchio
            mPaint.setColor(Color.WHITE);
            mPaint.setStyle(Paint.Style.FILL);
            mPaint.setStrokeWidth(0);
            String count = ""+((counter++) + mTicker);
            canvas.drawText(count, position.x,
                                position.y + 6, mPaint);
        }
    }
}

//metodo richiamato al tocco dello schermo
@Override
public boolean onTouchEvent(MotionEvent event) {
    switch (event.getAction()) {
        case MotionEvent.ACTION_DOWN:
            Point position = new Point (
                (int) event.getX(), (int) event.getY() );

            //aggiunge la posizione del punto
            mCounterPositions.add(position);
            invalidate();
            break;
    }
```

```
            return true;
        }

        public void run() {
            //...  vedi sopra
            }
        }
    }
}
```

5.6 Intent e passaggio dati fra activity

Come visto precedentemente la comunicazione fra activity avviene attraverso intent. Un'activity che ne avvia una seconda tramite intent può richiedere all'altra di svolgere un compito e, una volta conclusosi, di ritornarle un risultato. Alternativamente può passare direttamente all'activity che avvia un valore necessario per il suo corretto funzionamento. Vengono così messi a disposizione dei metodi per inserire o estrarre le informazioni direttamente dall'intent stesso.

5.6.1 I metodi per il passaggio dei dati

Per l'inserimento di valori in un intent, Android mette a disposizione il metodo polimorfo putExtra() (utilizzabile dall'oggetto intent in cui si vuole inserire l'informazione). Il metodo accetta come parametri una stringa che identifica in modo univoco il valore e il valore stesso. Quest'ultimo può assumere i seguenti tipi di dato:

- dato di tipo primitivo (boolean, char, byte, short, int, long, float, double, String);
- array di dati di tipo primitivo;
- CharSequence o array di CharSequence;
- Bundle;
- Parcelable o array di Parcelable;
- Serializable;

Dichiarato l'intent, sia che lo si passi a *startActivity()* che a *onActivityResult()*, si deve prima procedere con l'inserimento del dato via *putExtra()*:

```
tipo val = valore;
Intent i = new Intent(this, Activity2.class);
i.putExtra(com.android.example.prova"', val);
```

L'estrazione del valore passato avviene in Activity2 tramite l'utizzo del metodo *getExtra()*. Per poter acquisire correttamente il valore bisogna sapere a priori il suo tipo, dato che il metodo ritorna un dato di tipo Bundle (contenitore). Nel caso di valori interi, per esempio, si utilizza:

```
int val = getIntent().getExtras().getInt(com.android.example.prova);
```

5.6.2 Serializable e Parcelable

Nel caso in cui vi fosse la necessità di passare istanze di oggetti di tipo diverso dai tipi primitivi o da strutture di dati da loro formate, Android mette a disposizione due interfacce: Serializable permette il passaggio dell'istanza attraverso degli stream di comunicazione, Parcelable permette di serializzare gli oggetti per uno scambio efficente fra le activity.

Per utilizzare Serializable è necessario implementarla nella definizione della classe che si vuole passare (e in tutte le classi utilizzate al suo interno se non sono di tipo primitivo), aggiungendovi inoltre un campo *serialVersionUID* di tipo final static long. Quest'ultimo serve in fase di estrazione (deserializzazione) per verificare la compatibilità delle classi. Per recuperare poi l'oggetto passato via intent si utilizza sempre *getExtra()* seguito da *getSerializable*. Il meccanismo di serializzazione offerto da Serializable viene messo a disposizione però da Java e non da Android, è quindi pensato per l'ambiente desktop e non ottimizzato per altri terminali.

A differenza di quanto appena visto, Parcelable è offerto direttamente da Android. Si ha una maggior leggerezza a discapito di una maggior complessità di utilizzo. Per implementare l'interfaccia è necessario:

- fare l'override di due metodi da inserire nella classe da serializzare: *describeContents()* e *writeToParcel(Parcel dest, int flags)*. Il primo è utilizzato per la restituzione, se necessaria, di un FileDescriptor, il secondo invece si occupa di scrivere lo stato dell'oggetto sull'istanza di Parcel passata come parametro;
- implementare l'interfaccia Parcelable.Creator tramite classe anonima e assegnazione di questa alla costante pubblica CREATOR;
- implementare un costruttore che prenda in ingresso un Parcel per la rigenerazione dello stato.

Per una maggior chiarezza viene riportato di seguito un esempio:

```
public class Persona implements Parcelable {
  private String nome, cognome;
  private int age;
  public Persona(String nome, String cognome) {
  this.nome = nome;
  this.cognome = cognome;
}
public Persona(Parcel p) {
  this.nome = p.readString();
  this.cognome = p.readString();
  this.age = p.readInt();
}
@Override
public int describeContents() {
  return 0;
}
@Override
public void writeToParcel(Parcel p, int flags) {
  p.writeString(nome);
  p.writeString(cognome);
  p.writeInt(age);
}
public static final Parcelable.Creator CREATOR =
  new Parcelable.Creator() {
    public Persona createFromParcel(Parcel in) {
      return new Persona(in);
    }
    public Persona[] newArray(int size) {
      return new Persona[size];
    }
  };
}
```

A questo punto si procede analogamente a quanto visto con Serializable, ovvero utilizzando *putExtra()* per il passaggio e *getExtra()* seguito da *getParcelable()* per l'estrazione. Anche in questo caso ogni tipo non primitivo inserito nella classe deve implementare Parcelable.

Capitolo 6

Conclusioni

Questa tesi si è posta come punto di riferimento quello di illustrare il funzionamento del sistema operativo per dispositivi mobili Android, cercando di porre l'attenzione su aspetti specifici della piattaforma che ne definisco il 'carattere'.
Sfruttando la novità del momento, ovvero l'ultima versione del firmware (4.0), per quanto la percentuale di diffusione è ancora molto bassa, si è così potuto approfondire quella che può essere considerata un'importante tappa per lo sviluppo di questo sistema, segnata in particolare dall'unificazione sotto un unico firmware della versione per smartphone e per tablet.
Spero quindi che la riunificazione in un unico elaborato degli elementi principali che costituisco la struttura portante del sistema (con accanto i vari approfondimenti illustrati), oltre a dare una chiara idea delle sue potenzialità, possa essere un'utile risorsa sia per chi si affaccia per la prima volta a questo mondo sia per chi vuole anche solo far luce su semplici ma importanti meccanismi che lo governano.

Bibliografia

[1] "Sito dedicato agli sviluppatori android." http://developer.android.com/index.html.

[2] C. Fantozzi, "Slide del corso sistemi embedded 2010/11."

[3] "Sito dedicato a linux." http://www.tuxradar.com/.

[4] "Sito dedicato allo sviluppo." http://www.ibm.com/developerworks/.

[5] "Sito dedicato all'applicazione per android setcpu." http://www.setcpu.com.

[6] "Blog dedicato al mondo dei dispositivi mobili." http://www.batista70phone.com.

[7] "Blog dedicato al mondo android." http://www.androidiani.com.

[8] "Sito dove reperire esempi e spiegazioni su parti di software." http://www.simplesoft.it.

Elenco delle figure