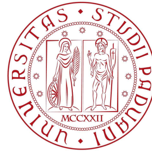


Università degli Studi di Padova

FACOLTÀ DI INGEGNERIA
Corso di Laurea in Ingegneria dell'Informazione

Tesi di Laurea Triennale

Utilizzo di un toolkit per applicazioni telefoniche su Android

Candidato:
Alberto Lazzarin
Matricola 575544

Relatore:
Prof. Michele Moro

Anno Accademico 2010-2011

Indice

1 Introduzione al sistema Android **2**
- 1.1 Cenni Storici 2
- 1.2 Caratteristiche generali 3
- 1.3 Controversie sulla privacy 4

2 L'applicazione Android **6**
- 2.1 Activity 6
- 2.2 Intent 8
- 2.3 Service 11
- 2.4 Content Provider e Resolver 12
- 2.5 BroadCast Receiver 13

3 Utilizzo del tookit **15**
- 3.1 Operazioni standard 15
- 3.2 Operazioni Avanzate 30

4 Conclusioni **43**

Capitolo 1

Introduzione al sistema Android

Il definitivo decollo del settore di telefonia mobile negli anni Zero ha portato ad un (prevedibile) aumento della complessità dell'hardware e software presenti nei telefoni cellulari. Il secondo aspetto in particolar modo ha raggiunto una evoluzione tale negli ultimi anni da farlo rientrare a tutti gli effetti nella categoria di sistema operativo, ricco di funzioni che ormai hanno poco a che fare o prescindono totalmente dalle classiche operazioni di chiamata o invio messaggi.

Il seguente elaborato si propone di prendere in esame uno dei più recenti e rivoluzionari OS disponibili per i cosiddetti "smartphone", Android, mediante un'analisi generica dell'architettura di una sua applicazione[1] seguita da un test, a titolo d'esempio, di un toolkit che comprende alcune tra le più comuni attività telefoniche.

1.1 Cenni Storici

L'ingegnere Andy Rubin, già noto per aver lavorato presso la Apple Inc., per aver sviluppato Magic Cap (un sistema operativo per PDA della General Magic) e per essere stato mananger presso la compagnia Danger (ditta specializzata in software e servizi per dispositivi mobili, poi acquisita da Microsoft), lasciò la stessa nel 2004 per poi concentrarsi in un progetto covato da tempo. Utilizzando il dominio di Android.com, Rubin concentrò tutte le sue risorse per riunire uno staff di ingegneri e "product planner" volto a sviluppare una nuova e innovativa piattaforma per cellulari. La startup rimase in ombra fino all'acquizione dalla parte di Google Inc. nel 2005. L'ormai celebre azienda nel 2007 costituì, assieme a colossi come ASUS, HTC, Intel, Motorola, Qualcomm, T-Mobile, e NVIDIA, un'alleanza finanziaria volta a stabilire nuovi standard per l'open source: l'Open Handset Alliance (OHA). Fu proprio essa a presentare ufficialmente nel 2007 la prima versione di Android, sistema operativo per smartphone basato su Linux con rilascio del relativo SDK. Ancora acerbo, Android dovette scontrarsi con

altri OS già consolidati e corredati quindi di un nutrito campionario di applicazioni, come Symbian, molto apprezzato nella sua versione 9.4 per le elevate prestazioni e il supporto nativo del Wi-Fi, oppure Windows Mobile. Nonostante questo, il nuovo OS targato Google ebbe un successo straordinario dovuto principalmente alla sua natura "open source" (come detto prima è basato sul kernel Linux) e alla elevata adattabilità per dispositivi mobili. Contrariamente al Symbian e ad altri sistemi proprietari, Android è offerto gratuitamente da Google ai produttori[1], basando il guadagno sui servizi online e sulla pubblicità ed è in continua evoluzione grazie alla presenza di una nutrita community alle spalle. Ad oggi Android, giunto alla versione 2.3, viene adottato da aziende come HTC, Acer, Samsung, Motorola, LG, Sony Ericsson, è la piattaforma mobile più diffusa negli USA mentre nel mondo è il terzo sistema più "venduto" superando anche Apple [3]. Inoltre viene utilizzato anche per dispositivi non inerenti alla telefonia, come tablet pc o ebook reader. Vista la conseguente perdita di mercato del suo OS, Nokia ha reso disponibile il codice del Symbian^3 (versione successiva al 9.5), ora anch'esso open source, nel Febbraio 2010 [2], e ha rilasciato nel Maggio 2010 il "MeeGo" nato in collaborazione con Intel, nuovo OS destinato ai telefoni di ultima generazione, ai netbook e ai Tablet PC, anch'esso basato su Linux [4]. Samsung dal canto suo ha recentemente sviluppato il Bada, OS proprietario limitato però al settore di telefonia mobile [5] .

1.2 Caratteristiche generali

Scaricando il source code (http://android.git.kernel.org/) di Android e dando un'occhiata al codice di un qualsiasi frammento è possibile cogliere la sua prima caratteristica significativa: Android non ha un linguaggio di programmazione proprio ma è scritto principalmente in Java. In realtà le librerie native contenute nel kernel sono realizzate in C e C++, tuttavia ciò non toglie che per utilizzare correttamente l'SDK è sufficiente conoscere discretamente il linguaggio di Sun Microsystem, nonchè un pò di sintassi XML[2] (vedasi il capitolo successivo). Questo aspetto molto positivo, considerando la notorietà di Java, va apparentemente in contrasto con il fatto che l'utilizzo della JVM per eseguire bytecode Java comporta il pagamento di una royalty. Da qui l'esigenza di compilare il codice in un bytecode diverso e di eseguirlo in una nuova Virtual Machine, detta "Dalvik", creata ad hoc per l'esecuzione in sistemi a risorse limitate, gli smartphone per l'appunto.

Durante la fase di compilazione, il codice sorgente subisce due trasformazioni: la prima nel classico bytecode java e la successiva conversione in file eseguibile DEX (Dalvik Executable). E' da notare come nella prima fase tutte le classi utilizzate o importate[3] nel codice vengano compilate in file .class separati men-

[1] Android è distribuito con la licenza "Open Source Apache License 2.0" senza la necessità di pagare alcuna royalty

[2] la gestione dei database richiede anche padronanza del linguaggio SQL

[3] Lo sviluppo di un'applicazione Android prevede la possibilità di utilizzo di tutte le classi presenti nel Java Development Kit

tre nella successiva trasformazione esse vengano raggruppate in un unico file (classes.dex). Il file .dex infatti comprende ottimizzazioni per la riduzione della ridondanza (attraverso la condivisione di risorse), che comporta minori dimensioni del file finale da eseguire, e un'architettura non più stack-based (come il bytecode Java) ma register-based (basato su registri). Questa scelta seppure renda più complessa la fase di compilazione, snellisce di molto l'esecuzione dei suddetti file nella Dalvik Virtual Machine. La DVM presenta molte analogie con la JVM (ad esempio il Garbage Collector) ma allo stesso tempo migliora alcune sue caratteristiche multithreading, assegnando ad ogni applicazione un processo Linux differente.

L'IDE ufficialmente supportato per l'utilizzo dell'ADK è Eclipse, al quale può essere integrato un apposito plugin (comprensivo di emulatore), che permette di compilare i progetti e renderli archivi installabili, in formato .apk ("Android package"), direttamente nel dispositivo Android [4].

1.3 Controversie sulla privacy

Nelle sue prime versioni (fino alla 1.6) Android subì alcune critiche inerenti alla stabilità e performance. Giunto ora alla versione 2.3 (detta "Gingerbread"), l'OS Google ha di certo corretto queste pecche, tuttavia resta ancora aperta una questione inerente allo scarso controllo svolto da Google sulle applicazioni presenti nell'App Store e alla poca trasparenza di informazioni e modalità con cui le stesse effettivamente operano. Jaeyeon Jung, ricercatore di Intel, e William Enck, dottorando alla Penn State University, hanno condotto uno studio orientato su questo aspetto i cui esiti sollevano più di qualche perplessità.

"Taintdroid" [6], creata appositamente dagli stessi, è un'estensione della piattaforma Android[5] che effettua un tracking mirato a segnalare il tipo di dati elaborato dalle applicazioni prese in esame e la loro destinazione. Il sistema di tracciamento consiste in una modifica al livello dell'interprete della VM e nell'uso di una libreria Taint (letteralmente "macchia"). Ad ogni invocazione da parte di una applicazione presa in esame (sospetta) di un metodo nativo (come visto prima scritto in C++ e considerato "affidabile") per il prelevamento di dati sensibili (ad esempio accesso ad un content provider[6]), viene eseguito un processo (cosiderato affidabile) in cui si estraggono i dati richiesti e si raggruppano in parcelle poi inviate tramite il Binder IPC[7] verso l'applicazione sospetta. A questo punto il binder, opportunamente modificato, "macchia" ogni singola parcella attraverso un tag (diverso in base al tipo di dati richiesti). Alla ricezione della parcella, la macchia viene "recuperata" dalla parcella e "propagata" in ogni

[4] A differenza di Android ad esempio, l'SDK dell'ultima versione di Symbian prevede principalmente l'utilizzo del linguaggio C++ usando i tool di sviluppo Qt, ma è possibile scrivere applicazioni anche in Java ME o Python.

[5] essa è disponibile per il download nel sito ufficiale segnalato, ma il funzionamento è garantito solo sul dispositivo Nexus One con Android 2.1

[6] si veda il capitolo successivo per maggiori informazioni

[7] IPC sta per "Inter Process Communication". Mentre il Binder è una classe che permette l'invocazione di metodi per via remota

valore letto dall'applicazione in questione. Il tracking ovviamente deve proseguire all'interno dell'applicazione stessa: viene modificata l'allocazione degli stack frame per metodi (a cui sono stati passati come parametri variabili macchiate) o dati "sporcati" e viene aggiunto un taint tag per ogni file salvato su memoria secondaria. Una volta che l'applicazione chiama un "taint sink" (ad esempio per inviare informazioni ad un server), la libreria Taint toglie la macchia dai dati eventualmente sporcati, ne permette l'invio, e registra su un file log il nome dell'applicazione, il tipo di dati inviati e la destinazione. E' da notare che questo approccio si differenzia dall'instruction-level tracking (analisi di ogni istruzione) per un minore overhead[8] (diminuzione della performance del 14% rispetto ad un rallentamento di più del doppio) ma per una maggiore presenza di falsi positivi (ad esempio: se una variabile macchiata viene inserita in un array, tutto l'array viene considerato come "sporco" assieme al suo contenuto).

Su 358 applicazioni richiedenti permessi per l'accesso a Internet (l'Android Market vanta ad oggi più di 1000 applicazioni in totale), ne sono state esaminate 30 (tra cui antivirus, meteo, servizi news, giochi). 15 di esse hanno inviato coordinate geografiche a server pubblicitari (ad esempio "admob.com" o "ads.mobclix.com"), e alcuni anche il codice IMSI (codice unico utilizzato per identificare un utente in una rete GSM), IMEI e l'ICC-ID (numero seriale della SIM). Il test, nonostante i rischi prima descritti, non ha generato falsi positivi. Delle 105 connessioni sospette, 37 sono poi state riconosciute come legittime (vedi il download delle mappe da parte di google maps), ma le altre vanno considerate come violazioni della privacy, aggravate dal fatto che buona parte delle applicazioni colpevoli non presentavano una EULA (End User License Agreement) al momento dell'installazione.

Lo studio lascia aperti dunque molti interrogativi sui rischi derivanti da una piattaforma open-source, rischi messi ancora più in luce dall'interesse rinvigorito da parte delle principali aziende telefoniche verso questa tipologia di software.

[8] e questo permette a TaintDroid di essere meglio testato anche da comuni utenti seppur con certi limiti

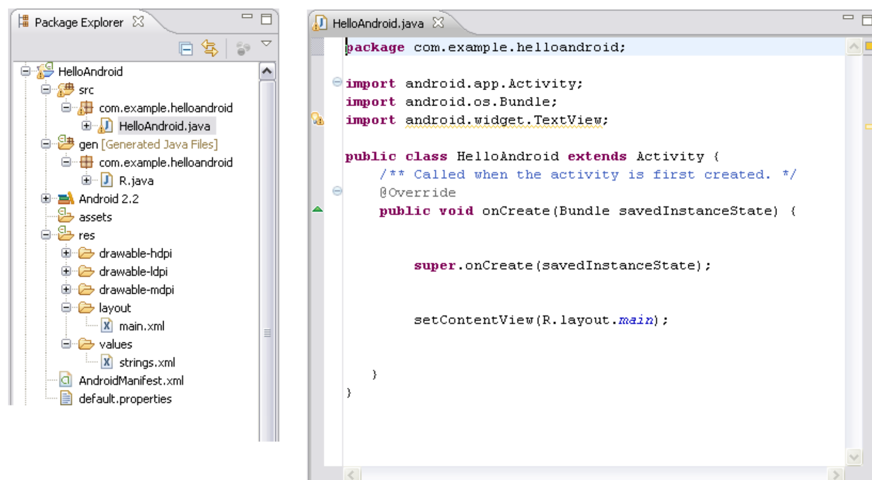
Capitolo 2

L'applicazione Android

Nel secondo capitolo vengono discusse le principali componenti di una applicazione Android. Ogni "app" comprende almeno una di queste ma non per forza tutte (ad esempio un programma banale può consistere di solo un'Activity o un Intent). L'analisi delle stesse verrà effettuata mediante l'utilizzo dell'IDE Eclipse con l'intento di abbinare la loro descrizione teorica a semplici esempi implementativi.

2.1 Activity

Creando un nuovo progetto Android (sempre tramite Eclipse), viene fornita la possibilità fin dall'inizio di creare un'Activity principale. Acconsentendo, una volta terminato il processo di creazione, la struttura iniziale del codice sorgente appare analoga a quella mostrata nella figura sottostante.



Osservando il codice di "HelloAndroid.java", presente nella cartella "src" (contenente i codici sorgenti), si intuisce subito che l'Activity è una classe. La class "Activity" descrive in particolar modo un'interfaccia grafica, o meglio una UI con la quale l'utente visualizza messaggi o interagisce con l'app stessa che ne fa uso. E' evidente quindi la fondamentale importanza di un simile componente, tant'è che la maggior parte delle applicazioni presenti per Android può essere vista come una serie di Activity[1] (o più semplicemente una sequenza di schermate). Nel codice sorgente evidenziato si nota l'implementazione (obbligatoria) del metodo "onCreate", chiamato alla creazione della Activity stessa. In questo caso ci si limita a impostare l'organizzazione grafica della schermata da visualizzare, la quale però non è descritta nel codice java, bensì nel file "main.xml", invocato attraverso il metodo "*setContentView(R.layout.main);*". Esso chiama appunto una risorsa contenuta nella cartella layout ("R" sta per "resources", cioè viene invocata la risorsa main, contenuta nella directory "layout", a sua volta contenuta in "res"), che descrive la struttura dell'interfaccia, ossia la presenza di TextView (caselle di testo), Button (pulsanti), ListView ecc...[2] con le corrispettive caratteristiche (trasparenza, colore, dimensioni, id, gerarchia). Queste possono al loro volta essere chiamate e modificate dinamicamente nel sorgente java utilizzando le omonime classi e agganciandosi agli elementi xml attraverso il loro id.

Si sorvola sulle altre risorse come i drawable (immagini e icone associate all'applicazione) e string.xml (contenente oggetti String).

La classe "R"

E' bene spendere due parole sul file "R" per una maggiore comprensione dell'elaborato. Aprendo il seguente file ci si trova di fronte ad una serie di sottoclassi "final" (cioè non estendibili) e statiche che contengono a loro volta costanti. Quelle costanti sono indirizzi alle risorse contenute dentro la cartella "res".

[1] se ancora non si è capito, ad ogni singola schermata corrisponde una singola Activity

[2] in pratica tutti gli oggetti di tipo "View"

```java
/* AUTO-GENERATED FILE.  DO NOT MODIFY.

package com.example.helloandroid;

public final class R {
    public static final class attr {
    }
    public static final class drawable {
        public static final int icon=0x7f020000;
    }
    public static final class id {
        public static final int textview=0x7f050000;
    }
    public static final class layout {
        public static final int main=0x7f030000;
    }
    public static final class string {
        public static final int app_name=0x7f040001;
        public static final int hello=0x7f040000;
    }
}
```

L'argomento non verrà approfondito in quanto non indispensabile per comprendere i test del capitolo successivo ma è interessante notare due cose. La prima è che il file è auto-generato: in sostanza ogni volta che l'utente crea, ad esempio, un nuovo file "xml" dentro la cartella layout, oppure una nuova View nel relativo XML di layout, o ancora un valore in "string.xml", "R" viene automaticamente aggiornato da un tool dell'ADK con una nuova costante (dentro, rispettivamente, alle sottoclassi "layout", "id" e "string") che indirizza opportunamente l'elemento. "R" non va dunque modificato manualmente. La seconda è intuibile dall'esempio prima esposto: l'unico modo per chiamare una particolare risorsa (all'interno appunto di "res") in un componente dell'applicazione, è riferirsi alla relativa costante dentro R.

2.2 Intent

Ogni applicazione viene eseguita in un processo Linux differente (salvo opportune eccezioni), fatto per cui va considerata indipendente dalle altre. Considerando che qualsiasi applicazione non banale richiede solitamente più di una schermata, si rivela necessario un meccanismo di interazione tra le Activity, con cui un'attività può magari chiamarne un'altra trasferendo dati da elaborare. Gli Intent rispondono a queste (e ad altre) esigenze. Il codice qui sotto descrive un'applicazione che gestisce oggetti di tipo "Person".

Attivita1.java

```
package com.example.testFraAttivita;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;

public class Attivita1 extends Activity {
        Intent intent;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.act1);


        int i=5;
        String s="ciao";
        Person p=new Person("Erich", "Gamma");

        intent=new Intent(getApplicationContext(), Attivita2.class);

        intent.putExtra(pkg+".myString", s);
        intent.putExtra(pkg+".myInt", i);
        intent.putExtra(pkg+".myPerson", p);


        Button button=(Button)findViewById(R.id.startAct2);
        button.setOnClickListener(new OnClickListener() {

                        @Override
                        public void onClick(View v) {
                                startActivity(intent);

                        }
                });
    }
}
```

"Attivita1" ha il compito di creare un oggetto Person, e al click del pulsante "startAct2", presente nel act1.xml, di passare alla classe "Attivita2" la quale effettuerà elaborazioni sull'oggetto stesso. Si notino alcuni elementi:

intent=new Intent(getApplicationContext(), Attivita2.class); la creazione dell'Intent che attiverà la seconda attività. Il primo parametro è un oggetto di tipo "Context" associato all'applicazione. Il Context è un'interfaccia in Android che descrive, come il nome suggerisce, l'ambiente e lo stato

dell'applicazione. Il Context dell'app è un parametro richiesto spesso, soprattutto per il "broadcast" di Intent, ed è accessibile nella maggior parte dei casi via "this". Si sappia che, ad esempio, la classe "Activity" e "Service" sono una sua particolare implementazione.

intent.putExtra(String nomeInformazione, Object valore); all'intent vengono correlate informazioni che potranno poi essere prelevate dalla seconda attività una volta attivata.

startActivity(intent); l'Attivita1 attiva l'Attivita2 mediante l'Intent (l'azione viene eseguita al click del "Button" descritto).

La seconda attività, una volta attivata dalla prima, dovrà per forza prelevare le informazioni create da "Attivita1" e "trasportate" dall'intent, attraverso le chiamate qui riportate...

```
Intent intent=getIntent(); // prelevo l'intent che ha attivato Attivita2

String pkg=getPackageName();

String s=intent.getStringExtra(pkg+".myString");
int i=intent.getIntExtra(pkg+".myInt", -1);
Person p=(Person)intent.getSerializableExtra(pkg+".myPerson");
```

... prima di elaborarle.

E' fondamentale notare come gli Intent non si limitino ad attivare altre Activity, ma possano chiamare qualsiasi tipo di applicazione, se questa dispone di alcune funzioni richieste. Questo aspetto mette in risalto uno dei pregi delle applicazioni Android cioè la loro modularità: se una certa applicazione ha la necessità di compiere un'operazione o funzione già eseguita in maniera efficiente da altre app presenti nel dispositivo, non serve ricopiarne il codice, ma è sufficiente utilizzare un intent per richiamare la funzione direttamente dall'applicazione che già la implementa.

Si torni un attimo alla prima immagine mostrata nell'elaborato, e si consideri il file AndroidManifest.xml. Esso è fondamentale nell'architettura di una applicazione in quanto ne costituisce la sua "carta d'identità".

TestFraAttivita Manifest

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.example.testFraAttivita"
      android:versionCode="1"
      android:versionName="1.0">
    <application android:icon="@drawable/icon" android:label="@string/app_name">
        <activity android:name=".Attivita1"
                  android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

    <activity android:name="Attivita2" android:label="la seconda attività">
    </activity>
    </application>
    <uses-sdk android:minSdkVersion="3" />
</manifest>
```

Manifest | Application | Permissions | Instrumentation | AndroidManifest.xml

Nel Manifest in particolar modo dell'applicazione presa in esame si possono vedere alcuni elementi come nome dell'applicazione, versione, dichiarazione dell'icona da utilizzare ecc... Si presti attenzione all'elemento "intent-filter": è proprio questo che descrive il tipo di azioni utilizzabili da un intent, che può compiere l'applicazione. Un'Activity può dunque chiedere all'OS di eseguire una qualsiasi app che possa compiere la "action" specificata, attraverso le le seguenti righe di codice

- intentGenerico.setAction(CUSTOM_ACTION);
- startActivity(intentGenerico);

A seguito di ciò l'OS del dispositivo avvia un processo detto "Intent Resolution", che avrà il compito di attivare un componente il cui manifest presenta nella sezione "intent-filter", l'action CUSTOM_ACTION. La scelta dell'Activity viene svolta runtime generando un errore di esecuzione nel caso che nessun componente installato nel dispositivo presenti quell'action nel suo Intent Filter[3].

2.3 Service

Un "Service" è un concetto già riscontrabile in ormai qualsiasi sistema operativo di una certa complessità. Esattamente come gli Intent non ha interfaccia grafica, ma esso non si occupa di stabilire una comunicazione tra attività e applicazioni ma esegue per un tempo indeterminato operazioni in background. Ovviamente un Service, che è a tutti gli effetti una classe, dispone di un'interfaccia che può essere chiamata da un'applicazione per modificare il suo flusso esecutivo: l'esempio più ovvio è un riproduttore multimediale. Attraverso una Activity l'utente

[3] nell'esempio presentato, l'unica action presente nell'Intent Filter è "MAIN", cioè la più generica possibile. Essa viene generata in automatico per ogni applicazione nel suo manifest.

può selezionare il brano da ascoltare, selezionando il tasto "Play" viene di fatto eseguito un Service (attraverso un Intent) che esegue il file audio permettendo all'utente stesso di cambiare activity, nonchè applicazione, senza fermare la musica in sottofondo (comunque interrompibile tornando ad esempio all'Activity di prima e premendo il tasto "Pause" o "Stop"). Ecco alcuni metodi utili per una maggiore comprensione:

public abstract ComponentName startService (Intent intent) E' chiamato solitamente da una Activity per "generare" il servizio.

public abstract boolean stopService (Intent intent) E' chiamato per interrompere il servizio.

public abstract boolean bindService (Intent service, ServiceConnection

conn, int flags) Metodo molto interessante. Permette di stabilire una connessione (comunicazione) con un servizio già attivo (il campo "flags" indica le modalità di interazione). Il servizio in questione stabilirà se accettarla o meno in base all'implementazione del suo metodo onBind().

public abstract void unbindService (ServiceConnection conn) Interrompe la connessione con il Service prima "agganciato".

Non ci si soffermerà ulteriormente su questo componente in quanto non indispensabile per le prove effettuate nel capitolo 3.

2.4 Content Provider e Resolver

La condivisione delle risorse o dei componenti (attraverso l'uso degli Intent) è un aspetto fondamentale dello sviluppo in Android come si è visto. Considerando che non è possibile per un'applicazione leggere o modificare dati, come database SQLite o file salvati, privati di un'altra applicazione, si fa necessaria la definizione di una classe che rappresenti un contenitore di risorse accessibile da più app. Il Content Provider è quindi un repository di informazioni, solitamente database, accessibile attraverso l'uso di URI (Uniform Resource Identifier). Si consideri il seguente frammento di codice...

```
Uri simUri = Uri.parse("content://icc/adn");
c = getContentResolver().query(simUri, null, null, null, null);
startManagingCursor(c);
```

Qui è mostrato un modo per accedere ai contatti della Sim Card. Questi sono appunto interfacciabili attraverso un Content Provider che se chiamato (tramite la seconda riga di codice) restituisce un oggetto Cursor (puntatore) riferito ad un database SQLite (la cui implementazione viene nascosta, per un'esigenza di incapsulamento). La terza riga di codice dichiara l'attivazione del Cursor.

Tornando alla seconda si noti che il metodo "query" è chiamato non tanto dal Content Provider, ma da un oggetto generico di tipo **Content Resolver** (ottenuto tramite la chiamata "getContentResolver()"), oggetto che fa da tramite per "interrogare" un Content Provider. Un'applicazione normale infatti non può utilizzare i metodi del Content Provider direttamente.

Gli URI utilizzati per accedere ad un Content Provider sono tutti del tipo "*content://<authority>/path*" in cui *authority* identifica univocamente il provider e *path* la risorsa richiesta. La realizzazione esatta di un Content Provider viene omessa, basti sapere che per l'implementazione di un oggetto simile è necessario riscrivere opportunamente alcuni metodi ("delete", "insert", "getType", il già visto "query", "update" e il classico "onCreate", in cui verranno generati i dati da condividere) e modificare l'AndroidManifest.xml aggiungendo l'elemento dichiarativo

```
<provider android:name="name_of_ContentProvider"
    android:authorities="AUTHORITY"></provider>
```

Per la lista dei vari Content Provider e dei loro URI è possibile consultare il seguente link: *http://developer.android.com/reference/android/provider/package-summary.html*.

2.5 BroadCast Receiver

L'ultimo componente analizzato in questo capitolo si limita a "reagire" ad eventi esterni (all'applicazione), come l'arrivo di una telefonata, di un sms o di particolari avvertenze del dispositivo (ad esempio, un livello basso di batteria). La diffusione di notifiche (i broadcast) avviene tramite l'OS, e l'applicazione, attraverso l'implementazione di BroadCast Receiver, può stabilire in base a quali eventi reagire ed in che modo.

La classe "BroadcastReceiver" non dispone di una UI particolare, inoltre "elabora" gli eventi esterni in maniera simile alla gestione di Intent pendenti con la possibilità di attivare un'Activity oppure utilizzando semplicemente la classe "NotificationManager". Essa permette di avvertire l'utente dell'evento attraverso un suono, la visualizzazione di una particolare icona nella status bar oppure facendo vibrare il dispositivo.

Processi e Sicurezza

Pur essendo open source è evidente che un sistema operativo dalla portata commerciale di Android debba disporre di opportuni vincoli di sicurezza per garantire affidabilità all'utente medio.

Come si è accennato in precedenza, per default ad ogni applicazione è associato un processo con un particolare UserID. Questo comporta una certa indipendenza tra di loro e permette l'interazione solo attraverso strumenti quali gli Intent. La procedura tuttavia non è sempre immediata in quanto certi servizi di sistema o funzioni particolari (lettura ad esempio di identificativi del telefono) richiedono che il processo chiamante abbia a disposizione alcuni "per-

messi" definiti nell'applicazione da interrogare. I permessi, in gergo android "Permission", possono essere definiti dall'utente anche nell'SDK per la propria applicazione nell'AndroidManifest. A questo punto, se tale applicazione è un Content Provider, un'altra applicazione (con userID diverso) non può accedere alla prima normalmente[4], ma deve dichiarare nel proprio manifest l'elemento "uses-permission", specificando la Permission della prima app. In tale modo viene permesso all'applicazione di accedere ai contenuti protetti, ma viene generata in fase di installazione (dell'app richiedente) un'avvertenza, impostata come attributo della Permission, che segnala all'utente il fatto che la suddetta applicazione, se installata, effettuerà operazioni "sensibili" (in questo caso, l'accesso ad un Content Provider protetto).

Esistono Permission standard definite in ambiente Android, liberamente consultabili nella documentazione ufficiale[5], ma è doveroso far notare che, come si vedrà in seguito, attraverso le permission non è possibile accedere a tutte le funzionalità o API disponibili nel sistema operativo. Certi metodi critici infatti, prima di procedere, effettuano un controllo non tanto del permesso che l'applicazione esterna vuole acquisire, ma del suo UserID: se non è uguale ad uno prefissato (ad esempio, di sistema), esso genera una particolare SecurityException che nega l'utilizzo della risorsa all'applicazione in questione.

Di fatto è possibile definire un particolare UserID (detto "Shared") nel manifest. Esso permette a più applicazioni di essere eseguite nello stesso processo, con tutto ciò che ne consegue. La procedura però ha un prezzo: i file-archivio apk, le cui applicazioni nel manifest esibiscono lo stesso Shared User Id, devono essere firmati con la medesima chiave e certificato[6] [7]. Da questo ne consegue che se una certa applicazione esibisce nel manifest un "android.uid.system" (cosa che in pratica le permette di eseguire qualsiasi operazione), essa dovrà essere firmata solo con la stessa chiave e certificato utilizzati per firmare i componenti del sistema operativo, cosa praticamente impossibile in una distribuzione standard di Android, i cui certificati e chiavi sono in possesso esclusivo del vendor.

Ciò in ogni caso risulta molto utile per uno sviluppatore che desidera aggiornare, sostituire o aggiungere funzionalità ad un'applicazione già pubblicata. Se la nuova versione del software è firmata allo stesso modo di quella precedente, essa, una volta installata, andrà a sostituire automaticamente quella ormai obsoleta, altrimenti verrà installata come applicazione diversa (o in alternativa genererà errore in fase di install se l'applicazione precedente apparteneva ad un particolare UserId).

---

[4]come si vedrà in seguito, viene sollevata una SecurityException

[5]si controlli in particolare la documentazione relativa alla classe android.Manifest.permission

[6]La documentazione ufficiale recita chiaramente "only two applications signed with the same signature (and requesting the same sharedUserId) will be given the same user ID".

Capitolo 3

Utilizzo del tookit

Nel terzo e ultimo capitolo, verranno sfruttate le conoscenze basilari esposte nei primi due per effettuare una breve analisi e test delle API che Android dispone per interagire con il telefono e più in particolare con la SIM.

3.1 Operazioni standard

Utilizzo di TelephonyManager

Tra le API pubbliche liberamente consultabili nel sito ufficiale è presente il package "android.telephony", il quale, come suggerisce il nome, offre una discreta gamma di operazioni nell'ambito su cui si sta focalizzando l'attenzione.

package

android.telephony

Since: API Level 1

Classes | Description

Provides APIs for monitoring the basic phone information, such as the network type and connection state, plus utilities for manipulating phone number strings.

more...

Classes

CellLocation	Abstract class that represents the location of the device.
NeighboringCellInfo	Represents the neighboring cell information, including Received Signal Strength and Cell ID location.
PhoneNumberFormattingTextWatcher	Watches a `TextView` and if a phone number is entered will format it using `formatNumber(Editable, int)`.
PhoneNumberUtils	Various utilities for dealing with phone number strings.
PhoneStateListener	A listener class for monitoring changes in specific telephony states on the device, including service state, signal strength, message waiting indicator (voicemail), and others.
ServiceState	Contains phone state and service related information.
SignalStrength	Contains phone signal strength related information.
SmsManager	Manages SMS operations such as sending data, text, and pdu SMS messages.
SmsMessage	A Short Message Service message.
SmsMessage.SubmitPdu	
TelephonyManager	Provides access to information about the telephony services on the device.

Enums

SmsMessage.MessageClass	SMS Class enumeration.

Except as noted, this content is licensed under Apache 2.0. For details and restrictions, see the Content License.
Android 2.3 r1 - 07 Dec 2010 14:28
Site Terms of Service - Privacy Policy - Brand Guidelines

Si noti innanzitutto in alto a destra la voce "Since: API Level 1", la quale indica che il package da analizzare è presente dalla versione 1.0 di Android (API di livello 1 appunto), fatto abbastanza ovvio considerata la tipologia dei dispositivi su cui l'os viene adottato più spesso.

Android offre agli sviluppatori che utilizzano l'SDK un completo accesso alle funzionalità SMS-MMS, con cui è possibile creare ex novo un nuovo client SMS, gestione delle chiamate, e lettura di alcuni stati del telefono fondamentali.

Si può iniziare con la classe più generica, cioè TelephonyManager. Questa classe di fatto permette un'interazione con il Telephony Service, servizio di sistema che utilizza strumenti implementati a basso livello ma che può essere facilmente interrogato tramite la chiamata **getSystemService(String nome-**

Servizio)[1], la quale appunto restituisce il riferimento ad un oggetto di tipo TelephonyManager (senza utilizzare il costruttore). Il metodo evidenziato, appartenente alla classe “Context”, è estremamente utile in quanto permette di chiamare molti servizi di sistema, quali il “Power Service” o il “Notification Service”, senza utilizzare ad esempio il metodo più esplicito “bindService” visto in precedenza[2].

A questo punto dunque si può creare un nuovo progetto, chiamato “TelephonyTestApp”: il build verrà effettuato con il source della versione 2.2 di Android, e verrà creata “ActivityOne” come Activity iniziale. La prima activity deve solamente mostrare una serie di dati, ossia stringhe, dunque, il relativo “main.xml”, sarà costituito da 2 TextView (titolo e risultato delle chiamate) e 1 Button per passare all’attività successiva...

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
<TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:gravity="center_horizontal"
    android:text="Phone Status"
    android:textColor="#00FF00"/>
<TextView android:id="@+id/dialog"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content"
 android:gravity="center_horizontal"
 android:layout_marginTop="15sp"
 android:text="&lt;Valore&gt;"
 android:layout_gravity="center_horizontal">
 </TextView>
<Button android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:text="Clicca qui per continuare"
  android:id="@+id/Pulsante"
  android:layout_gravity="center_horizontal">
    </Button>
</LinearLayout>
```

Osservando la figura si può vedere a sinistra il contenuto del file “main.xml” mentre a destra il suo relativo layout (mostrato direttamente da Eclipse anche senza dover eseguire l’applicazione nell’emulatore)[3]. Si faccia attenzione la prima volta all’elemento “android:id” di ogni View: il valore “@+id/Nuovo_Nome”

[1]La documentazione ufficiale chiarisce che non va creata una nuova istanza di TelephonyManager, ma è necessario ottenere il riferimento a quella già istanziata dal sistema. “getSystemService” ha la funzione di ottenere, runtime, il riferimento.

[2]Similmente a quanto visto prima, getSystemService ha il compito di prelevare per riferimento l’istanza dei relativi “Manager” creati dal sistema, con i quali è possibile interagire con i servizi.

[3]La figura sovrastante è a titolo d’esempio. Non verranno più mostrati i contenuti dei file “layout” xml, in quanto non necessari alla comprensione del resto dell’elaborato.

indica l'attribuzione di un nuovo id alla View, parametro fondamentale per fare riferimento ad essa nel codice java. Si può passare ora al codice della prima parte dell'applicazione.

```
1  package com.example.telephonyTestApp;
2
3  import com.example.telephonyTestApp.R;
4
5  import android.app.Activity;
6  import android.content.Context;
7  import android.os.Bundle;
8  import android.telephony.TelephonyManager;
9  import android.widget.TextView;
10
11 public class ActivityOne extends Activity {
12     /** Called when the activity is first created. */
13     @Override
14     public void onCreate(Bundle savedInstanceState) {
15         super.onCreate(savedInstanceState);
16         setContentView(R.layout.main);
17
18
19         //Accedo al servizio
20         String telSrvc = Context.TELEPHONY_SERVICE;
21
22         TelephonyManager telManager = (TelephonyManager) getSystemService(telSrvc);
23
24
25         String dettagli = "";
26
27
28         int phoneType = telManager.getPhoneType();
29         switch (phoneType) {
30
31         case (TelephonyManager.PHONE_TYPE_CDMA):
32         {dettagli = dettagli + "Il dispositivo è di tipo CDMA"+"\n"; break;}
33
34         case (TelephonyManager.PHONE_TYPE_GSM) :
35         {dettagli = dettagli + "Il dispositivo è un telefono GSM"+"\n"; break;}
36
37         case (TelephonyManager.PHONE_TYPE_NONE):
38         {dettagli += "Il dispositivo è un telefono di tipo sconosciuto"+"\n"; break;}
39
40         default: break;
41         }
42
43         // Legge il codice IMEI per il GSM o il MEID per il CDMA
44         String deviceId = telManager.getDeviceId();
45
46         // Numero di telefono
47         String phoneNumber = telManager.getLine1Number();
48
49         // Versione Software del telefono
50         String softwareVersion = telManager.getDeviceSoftwareVersion();
51
52         // "Aggancia" la TextView dei risultati
53         TextView primoDato = (TextView) findViewById(R.id.dialog);
54
55
56         dettagli += "IMEI: " + deviceId + "\n";
57         dettagli += "Software Version: " + softwareVersion + "\n";
58         dettagli += "Numero di telefono: " + phoneNumber + "\n";
59
60         primoDato.setText(dettagli);
```

La riga 22 è già stata spiegata: si noti solamente che la chiamata “getSystemService” restituisce un Object generico, per cui è necessario effettuare un

cast. Alla riga 53 si può notare come all'oggetto TextView, definito nel codice java, si associ quello definito nel file xml precedente attraverso l'id, in modo da poterlo modificare dinamicamente. Tutto il resto dovrebbe essere perfettamente comprensibile: il pezzo di app stampa su schermo il codice IMEI, la versione software, e il numero telefonico. Provando ad eseguire il codice (sull'emulatore o su un dispositivo, è indifferente), l'applicazione termina con un errore Runtime ("The application ... stopped unexpectedly. Please try again") non specificato. Non è un errore casuale, e può essere smascherato utilizzando il classico debug di eclipse o più semplicemente lo strumento "LogCat" fornito dall'SDK. Si prenda in considerazione questo metodo, più immediato, e si controlli il log: salta subito agli occhi una segnalazione di "Fatal Exception", sotto riportata...

```
FATAL EXCEPTION: main
java.lang.RuntimeException: Unable to start activity ComponentInfo{com.example.tel…
    at android.app.ActivityThread.performLaunchActivity(ActivityThread.java:2663)
    at android.app.ActivityThread.handleLaunchActivity(ActivityThread.java:2679)
    at android.app.ActivityThread.access$2300(ActivityThread.java:125)
    at android.app.ActivityThread$H.handleMessage(ActivityThread.java:2033)
    at android.os.Handler.dispatchMessage(Handler.java:99)
    at android.os.Looper.loop(Looper.java:123)
    at android.app.ActivityThread.main(ActivityThread.java:4627)
    at java.lang.reflect.Method.invokeNative(Native Method)
    at java.lang.reflect.Method.invoke(Method.java:521)
    at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:868
    at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:626)
    at dalvik.system.NativeStart.main(Native Method)
Caused by: java.lang.SecurityException: Requires READ_PHONE_STATE: Neither user 10041
    at android.os.Parcel.readException(Parcel.java:1247)
    at android.os.Parcel.readException(Parcel.java:1235)
    at com.android.internal.telephony.IPhoneSubInfo$Stub$Proxy.getDeviceId(IPhone…
    at android.telephony.TelephonyManager.getDeviceId(TelephonyManager.java:187)
    at com.example.telephonyTestApp.ActivityOne.onCreate(ActivityOne.java:44)
    at android.app.Instrumentation.callActivityOnCreate(Instrumentation.java:1047)
    at android.app.ActivityThread.performLaunchActivity(ActivityThread.java:2627)
    ... 11 more
```

E' in effetti una SecurityException la causa del problema. Non serve analizzare meticolosamente l'errore perchè la soluzione viene suggerita dalla stessa exception: i metodi invocati sono protetti da una Permission ("READ_PHONE_STATE"). E' necessario dunque specificare nell'AndroidManifest che la app "TelephonyTestApp" "usa" il permesso (già definito nelle permission di sistema) in questione...

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.example.telephonyTestApp"
      android:versionCode="1"
      android:versionName="1.0">
    <application android:icon="@drawable/icon" android:label="@string/app_name">
      <activity android:name=".ActivityOne"
       android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
      </activity>

    </application>
    <uses-sdk android:minSdkVersion="5" />

<uses-permission android:name="android.permission.READ_PHONE_STATE">
</uses-permission>
</manifest>
```

L'elemento è aggiunto (penultima riga) come "figlio" solo di "manifest".
A questo punto è possibile eseguire il codice per ottenere l'output desiderato.

Prima di proseguire è opportuno fare alcune considerazioni. I tre valori mostrati in figura risultano sballati per il semplice fatto che il test è stato eseguito sull'emulatore, il quale offre un'emulazione molto limitata della scheda SIM e di altri hardware telefonici. Il pulsante, se premuto, non sortisce alcun effetto e questo è ovvio perchè esso, pur essendo stato definito nel main.xml, non è stato ancora nominato nel codice, e dunque non è stata impostata l'azione da eseguire in caso di pressione. E' infine interessante notare come la modifica del manifest non abbia sortito alcun effetto sulla sua esecuzione nell'emulatore se non quello di correggere la SecurityException. Tuttavia se il progetto viene

esportato come apk[4], prima di confermare la sua installazione in un dispositivo appare su schermo il seguente messaggio: "Consenti all'applicazione di: Telefonate - lettura di stato e identità del telefono". Quando un'applicazione chiede l'utilizzo di un certo permesso, è l'utente, durante la fase di installazione, che ha la possibilità eventualmente di negarlo. Questo procedimento viene in effetti criticato nello studio condotto con TaintDroid, descritto prima, in quanto non fornisce comunque abbastanza informazioni sulle modalità di utilizzo di queste funzioni "sensibili": in assenza di un EULA adeguato, in caso di installazione di un'app analoga a quelle prese in considerazione nel primo capitolo, colpevoli di inviare dati personali a server pubblicitari, verrebbe segnalato solamente che l'app "legge lo stato e l'identità del telefono" e "accede a internet".

Detto questo si può proseguire utilizzando altri metodi della stessa classe, per prelevare altre informazioni, questa volta sulla SIM Card, aggiungendo il seguente codice (sopra la riga 60):

[4]Per compiere la seguente azione, Eclipse, correlato con l'estensione dell'Android SDK, dispone di un tool molto semplice che permette di compilare il codice sorgente, comprimerlo in un apk, e firmarlo con un certificato personalizzato in modo da renderlo installabile su qualsiasi dispositivo (Android non permette l'installazione di un apk non "signato")

```
int simState = telManager.getSimState();
switch (simState) {
  case (TelephonyManager.SIM_STATE_NETWORK_LOCKED):
  {dettagli = "SIM Bloccata \n"; break;}
  case (TelephonyManager.SIM_STATE_ABSENT):
  {dettagli = "SIM Assente \n"; break;}
  case (TelephonyManager.SIM_STATE_PIN_REQUIRED):
  {dettagli = "SIM - richiede PIN \n"; break;}
  case (TelephonyManager.SIM_STATE_PUK_REQUIRED):
  {dettagli = "SIM - richiede PUK \n"; break;}
  case (TelephonyManager.SIM_STATE_UNKNOWN):
  {dettagli = "SIM - Stato Sconosciuto \n"; break;}

  case (TelephonyManager.SIM_STATE_READY): {

   dettagli += "\nDettagli SIM Card\n" + "\n";

   // Numero seriale della SIM
   String simSerial = telManager.getSimSerialNumber();

   // Codice Operatore della SIM (MCC + MNC)
   String simOperatorCode = telManager.getSimOperator();

   // Nome dell'Operatore della SIM
   String simOperatorName = telManager.getSimOperatorName();

   // Codice ISO della SIM
   String simCountry = telManager.getSimCountryIso();

//Riunisco i dettagli
dettagli += "Numero Seriale: " + simSerial + "\n";
dettagli += "Nome Operatore: " + simOperatorName + "\n";
dettagli += "Codice Operatore: " + simOperatorCode + "\n";
dettagli += "Codice ISO: " + simCountry + "\n";
```

Vista la banalità non è necessario commentarlo, ma viene sfruttata l'occasione per introdurre una caratteristica Java fondamentale che è possibile sfruttare anche in ambito Android: la "**reflection**". Questa è un'insieme di API contenuto nel omonimo package (java.lang.reflection) che permettono di effettuare un'interrogazione dinamica di istanze, metodi, attributi, e costanti appartenenti ad una particolare classe. La reflection in buona sostanza può essere vista come un "import" runtime. Le applicazioni di questa peculiarità sono molte e vengono lasciate al lettore, mentre in questa sede verrà presa in considerazione la possibilità di invocare ed utilizzare API di Android non pubbliche. Si consideri il metodo "getSimOperator()" e si consulti la sua implementazione direttamente dal source della classe TelephonyManager.java (nel package "android.telephony"). Il corpo consiste semplicemente in una invocazione di un

altro metodo:

return SystemProperties.get

(TelephonyProperties.PROPERTY_ICC_OPERATOR_ALPHA);

La classe SystemProperties si trova nel package "android.os" e non è pubblica, tant'è che, se si tenta di importarla, Eclipse segnala immediatamente un errore di sintassi fittizio ("classe non esistente"). Di fatto essa è presente nell'os, ma l'SDK non ne permette l'utilizzo diretto. Proprio in questo ambito è possibile applicare a titolo d'esempio la reflection. Si sostituisca il metodo in questione con il seguente codice.

```
       // Nome dell'Operatore della SIM
       String simOperatorName = "";

try {
     Class systemPropertiesClass = Class.forName("android.os.SystemProperties");

     Class telefonoProprietà = Class.forName
     ("com.android.internal.telephony.TelephonyProperties");

     Method metodo = systemPropertiesClass.getMethod("get", String.class);

     String costante = (String) telefonoProprietà.getField("PROPERTY_ICC_OPERATOR_ALPHA")
     .get(telefonoProprietà);

// Corrisponde a "SystemProperties.get(TelephonyProperties.PROPERTY_ICC_OPERATOR_ALPHA);"
     simOperatorName = (String) metodo.invoke(systemPropertiesClass, costante);

     } catch (ClassNotFoundException e) {
       Log.e("ERROR", e.toString());
       e.printStackTrace();
     } catch (SecurityException e) {
       Log.e("ERROR", e.toString());
       e.printStackTrace();
    } catch (NoSuchMethodException e) {
       Log.e("ERROR", e.toString());
       e.printStackTrace();
    } catch (IllegalArgumentException e) {
       Log.e("ERROR", e.toString());
       e.printStackTrace();
    } catch (IllegalAccessException e) {
       Log.e("ERROR", e.toString());
       e.printStackTrace();
    } catch (NoSuchFieldException e) {
       Log.e("ERROR", e.toString());
       e.printStackTrace();
    } catch (InvocationTargetException e) {
       Log.e("ERROR", e.toString() + "Caused by: " + e.getCause().toString());
       e.printStackTrace();
    }
```

Le due classi (la già nominata SystemProperties e TelephonyProperties) vengono cercate per nome e salvate nell'oggetto di classe generico. Successivamente viene prelevato il metodo, specificando stavolta non solo il nome ma anche il tipo di parametro (se sono più di uno, è necessario un array di "Class") ricevuto

in ingresso. Si faccia attenzione alle righe successive: la costante viene prelevata come campo dalla classe TelephonyProperties, il ".get" finale sembrerebbe ridondante ma il parametro in ingresso corrisponde all'oggetto da cui si vuole prelevare il "field" (essendo in questo caso statico, lo si invoca su se stesso). Siccome viene restituito un generico Object è necessario effettuare il cast. Importante è infine l'invocazione finale che riceve 2 parametri in ingresso: l'istanza (Object) "ricevente" da cui viene invocato il metodo (in questo caso statico, per cui si utilizza la classe stessa, oppure "null", altrimenti è necessario prelevare il costruttore e utilizzare "newInstance()") e il parametro (o parametri, se si inserisce un array di Object) in ingresso del metodo che si sta invocando. La chiamata produce l'effetto equivalente a "ricevente.metodo(parametro/i)". Si noti come ultima cosa la presenza di numerose exception da gestire obbligatoriamente, le cui cause sono tutte intuibili dal nome[5].

L'esecuzione del codice modificato, seppur un pò più complesso (è a titolo d'esempio), produce lo stesso output di prima.

E' venuto il momento di utilizzare il pulsante, per ora definito solamente in "main.xml". Il Bottom al suo click potrebbe passare ad un'altra Activity, così come si è già visto nel primo esempio inerente agli Intent. Si faccia attenzione che la seconda Activity dovrà disporre di un altro xml grafico (situato nella stessa posizione di "main") e dovrà "attestare" la sua presenza nell'AndroidManifest (altrimenti "startActivity(intent)" termina in maniera errata): è necessario aggiungere...

```
<activity android:name="ActivityTwo"></activity>
```

come figlio di "application".

Lettura dei contatti (Content Provider) e invio di SMS (Intent)

La classe TelephonyManager permette di interrogare la SIM Card per leggere parecchie informazioni, ma in ogni caso espone uno scarso livello di interazione con essa, limitato a funzioni di tipo "get". E' opportuno quindi spendere qualche parola per mostrare altre funzionalità che l'os di Google mette a disposizione. In questa sezione, prima di tentare operazioni più complesse, verrà esposta un'implementazione di un client SMS. Alla seconda Activity appena creata si attribuirà il compito di leggere i contatti telefonici della Sim Card, dando la possibilità all'utente di selezionarne uno a cui inviare un SMS. Si torni alla prima figura vista nel capitolo e si noti che di fatto il package android.telephony offre API le quali, oltre alla gestione degli SMS, si limita a monitorare stati, leggere informazioni standard e manipolare numeri telefonici. Non vi è traccia di un'utility per le chiamate o di un modo per accedere a informazioni della SIM quali la rubrica o gli sms salvati. Per il primo proposito, è possibile

[5] Attenzione a InvocationTargetException: esso viene sollevato per ogni eccezione (per così dire "remota") che si verifica nel metodo invocato. L'eccezione "remota" può essere conosciuta proprio tramite "getCause()".

gestire le chiamate tramite Intent[6], per il secondo si può spulciare il package android.provider o in alternativa si può controllare il package, direttamente dal sorgente, "com.android.internal.telephony". Qui si può notare una classe chiamata "IccProvider": essa estende l'interfaccia ContentProvider ed esibisce ben 3 URI.

```
public class IccProvider extends ContentProvider {
    private static final String TAG = "IccProvider";
    private static final boolean DBG = false;


    private static final String[] ADDRESS_BOOK_COLUMN_NAMES = new String[] {
        "name",
        "number",
        "emails"
    };

    private static final int ADN = 1;
    private static final int FDN = 2;
    private static final int SDN = 3;

    private static final String STR_TAG = "tag";
    private static final String STR_NUMBER = "number";
    private static final String STR_EMAILS = "emails";
    private static final String STR_PIN2 = "pin2";

    private static final UriMatcher URL_MATCHER =
                            new UriMatcher(UriMatcher.NO_MATCH);

    static {
        URL_MATCHER.addURI("icc", "adn", ADN);
        URL_MATCHER.addURI("icc", "fdn", FDN);
        URL_MATCHER.addURI("icc", "sdn", SDN);
    }
                                                                [...]
```

Questi sono mostrati in figura dagli ultimi tre metodi: si sappia che il primo parametro in ingresso è l'*authority*, il secondo la *path*, e il terzo un numero (positivo) che identifica la risorsa (in questo caso semplicemente 1,2,3). ADN sta per "Abbreviated Dialing Numbers" e coincide con i contatti salvati normalmente nella SIM, FDN (Fixed Dialing Numbers) include alcuni numeri telefonici predisposti nella SIM a cui è possibile limitare le chiamate effettuate e infine SDN include altri numeri telefonici, predisposti nella SIM, inerenti a servizi dell'operatore. Non tutte le SIM Card dispongono degli FDN (e del relativo

[6] l'attuale SDK offre possibilità limitate in questo versante. Esso ad esempio infatti non permette di implementare la propria "in call" Activity: la schermata che appare quando si riceve o si effettua una chiamata.

servizio), tuttavia in loro presenza è necessario disporre del Pin2 (differente dal comune codice Pin) per modificarli.

Si posseggono ora tutti gli strumenti per accedere al ContentProvider, così come si è visto precedentemente, con l'accortezza di dichiarare nel manifest l'utilizzo della "READ_CONTACTS" permission. Per l'esempio verranno scelti gli ADN. Una volta effettuata la chiamata di inizializzazione "startManagingCursor(c)", si può sfruttare il "puntatore" per copiare i nomi in una View di tipo Spinner (una lista di selezione).

```
Uri simContactsUri = Uri.parse("content://icc/adn");
final Cursor curs = getContentResolver().query(simContactsUri,
        null, null, null, null);
startManagingCursor(curs);

Spinner contactsSP = (Spinner) findViewById(R.id.SpinnerB);

String[] contactsName = new String[curs.getCount()];

int i = 0;
int index = curs.getColumnIndex("name");

while (curs.moveToNext())
{
    contactsName[i] = curs.getString(index);
    i++;
}

ArrayAdapter<String> adapter = new ArrayAdapter<String>(this,
        android.R.layout.simple_spinner_item, contactsName);

contactsSP.setAdapter(adapter);
```

Il codice suggerisce che l'insieme degli ADN appare organizzato come una tabella con 2 colonne ("name" e "number"). Il cursor punta ad una particolare riga (partendo da "-1"), il "moveToNext()" quindi lo fa spostare a quella sottostante[7]. Adesso è necessario definire anche le operazioni che lo Spinner dovrà compiere, una volta selezionato un certo elemento.

[7] Si noti il passaggio intermedio ad un ArrayAdapter. In ingresso il "this" è il Context, mentre il secondo parametro è un layout standard che può essere chiamato in qualsiasi Activity.

```
final TextView numberView = (TextView) findViewById(R.id.Number);


contactsSP.setOnItemSelectedListener(new OnItemSelectedListener(){

    @Override
    public void onItemSelected(AdapterView<?> arg0, View arg1,
            int arg2, long arg3) {

        name = arg0.getItemAtPosition(arg2).toString();

        int position = (int) arg0.getItemIdAtPosition(arg2);

        curs.moveToPosition(position);

        number = curs.getString(curs.getColumnIndex("number"));

        numberView.setText("Numero Selezionato: " + number);

    }

    @Override
    public void onNothingSelected(AdapterView<?> arg0) {}
    });
```

Impostati i dati (il messaggio testuale può essere prelevato in input tramite una View di tipo EditText), si può procedere all'invio dell'SMS. Innanzitutto è possibile sfruttare la modularità di Android mediante gli Intent, senza rimpiazzare il client SMS ma utilizzando quello nativo già disponibile.

```
//---------- Ottengo il riferimento alla EditText --------------------

final EditText editMsg = (EditText) findViewById(R.id.EditTextB);

//---------------- SMS tramite Intent -------------------------------

final Button smsIntentBtn = (Button) findViewById(R.id.SmsIntent);

smsIntentBtn.setOnClickListener(new OnClickListener(){

    @Override
    public void onClick(View v) {

    if (number == null)
        Toast
        .makeText(
                ActivityTwo.this, "ERRORE: Numero non selezionato",
                Toast.LENGTH_LONG).show();

    else {
    message = editMsg.getText().toString();

    Intent smsIntent = new Intent(Intent.ACTION_SENDTO,
            Uri.parse("sms:" + number));
    smsIntent.putExtra("sms_body", message);

    startActivity(smsIntent); }

    }});
```

All'esecuzione di "startActivity", l'SMS non viene inviato ma semplicemente viene attivata la schermata standard di "invio SMS" con i campi "numero" e "testo" (quest'ultimo passato come informazione extra) già compilati. E' interessante notare come, per effettuare questa operazione, non sia necessaria la dichiarazione di alcuna permission in quanto non è questa Activity che si occuperà di inviare realmente il messaggio, ma un'altra la cui applicazione già possiede tutti i permessi richiesti.

Si può procedere infine all'invio manuale del messaggio, cosa che richiede il permesso "SEND_SMS". Verrà utilizzata per l'occasione la classe SmsManager.

```
//-------------------------------------------------------------------
//----------- SMS manuale - richiede la permission SEND_SMS -----------

   final SmsManager smsManager = SmsManager.getDefault();

   Button smsManualBtn = (Button) findViewById(R.id.SmsNormal);

   smsManualBtn.setOnClickListener(new OnClickListener() {

       @Override
       public void onClick(View arg0) {

           if (number == null)
               Toast
               .makeText
               (ActivityTwo.this, "ERRORE: Numero non selezionato",
                Toast.LENGTH_LONG);

           else {

               message = editMsg.getText().toString();

               smsManager.sendTextMessage
               (number, null, message, null, null);

           }
       }});
```

Il secondo parametro del metodo “sendTextMessage”, di tipo String, è il “Service Center address”. Se lasciato null, viene usato quello predefinito. Gli ultimi due parametri invece sono Intent (“sentIntent” e “deliveryIntent”) e possono essere usati per monitorare l’invio del SMS. Se questi Intent non vengono creati e gestiti mediante BroadCast Receivers (da creare dentro la nostra app), alla pressione del pulsante, il messaggio viene inviato senza alcuna segnalazione all’utente.

3.2 Operazioni Avanzate

In questa sezione verrà portato avanti il tentativo di sfruttare il sistema operativo per compiere azioni più complesse e interessanti, contrariamente a quelle descritte in precedenza e già implementate di default nelle attuali distribuzioni di Android. In particolare si cercherà di accedere ai dati, eventualmente protetti

dal Pin, della SIM che non vengono esposti direttamente nè tramite le normali applicazioni preesistenti nè tramite le classi esposte pubblicamente nell'ADK.

Si torni alla classe IccProvider vista prima, e si controlli l'implementazione del metodo "query", utilizzato per inizializzare l'oggetto cursor nella seconda Activity del progetto. Il metodo chiama a sua volta "loadFromEf" passando come parametro una costante Icc che identifica i file EF della rubrica SIM da caricare. Questo a sua volta chiama un servizio di nome "simphonebook" ed invoca su di esso il metodo "getAdnRecordsInEf", attraverso la classe "IIccPhoneBook" (sempre dentro il package "com.android.internal.telephony"). Nel dettaglio...

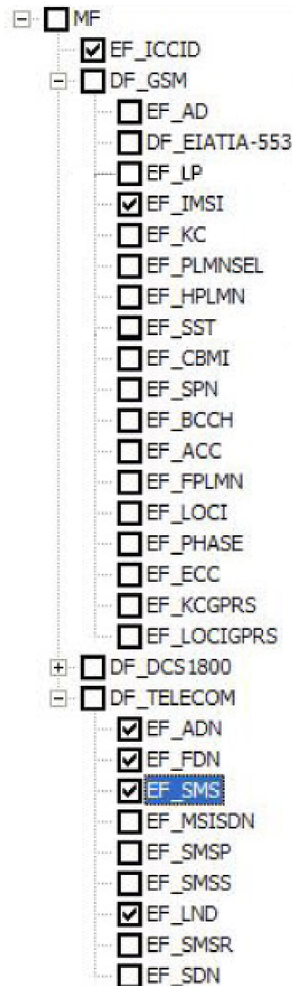
```
private ArrayList<ArrayList>  loadFromEf(int efType) {
    ArrayList<ArrayList> results = new ArrayList<ArrayList>();
    List<AdnRecord> adnRecords = null;

    if (DBG) log("loadFromEf: efType=" + efType);

    try {
        IIccPhoneBook iccIpb = IIccPhoneBook.Stub.asInterface(
                ServiceManager.getService("simphonebook"));
        if (iccIpb != null) {
            adnRecords = iccIpb.getAdnRecordsInEf(efType);
        }
    } catch (RemoteException ex) {
        // ignore it
    } catch (SecurityException ex) {
        if (DBG) log(ex.toString());
    }
  [...]
```

IIccPhoneBook non è una normale classe java, tant'è che i suoi metodi, se consultati tramite il sito web "grepcode"[8], sono solo dichiarati, bensì un file di tipo "aidl". IDL sta per "Interface description language", ed è un particolare linguaggio che permette la comunicazione, tramite chiamate remote, tra sistemi operativi o linguaggi di programmazione differenti, ad esempio C++ (metodi nativi in Android) e Java. In questo caso i file "AIDL" (Android Interface description language) permettono la comunicazione tra due processi Android (Linux) differenti usando l'IPC (interprocess communication). Senza approfondire l'argomento, si può già intuire che il metodo *loadFromEf* si interfaccia tramite l'aidl IIccPhoneBook con il servizio (che corrisponde ad un altro processo) di nome "simphonebook", chiamando *getAdnRecordsInEf*, dichiarato in IIccPhoneBook e implementato dal servizio. La ricerca di questo servizio porta prima di tutto alla classe astratta *IccPhoneBookInterfaceManager*, che appunto estende *IIccPhoneBook.Stub*. Quest'ultima utilizza un oggetto particolarmente interes-

sante: PhoneBase. Esso, che tra l'altro è una classe astratta, ha un'importanza fondamentale in tutto il package telephony, infatti qualsiasi classe che gestisca i file EF della Sim Card (eccetto RIL.java), per essere inizializzata richiede come parametro del costruttore un oggetto di questo tipo. Di seguito si cercherà quindi di creare e di utilizzare un oggetto che implementi PhoneBase, prima tuttavia è conveniente fornire una breve infarinatura sulla SIM Card e su i relativi file EF (cui si cercherà di accedere). In seguito ci si basa sull'articolo [9].

Android e la SIM Card

La Subscriber Identity Module (SIM) è una particolare ICC (Integrated Chip Card), che può essere vista a tutti gli effetti non solo come un dispositivo di memoria, ma come un vero microcomputer a sè stante. Esso infatti possiede un quantitativo di memoria RAM per elaborare i comandi richiesti dal dispositivo che monta il chip, di EEPROM per salvare alcuni file utente, e di ROM, ove risiede il sistema operativo. Come tutti i sistemi operativi possiede un rudimentale file system comprensivo di directory ad albero con la seguente struttura: il master file (MF), il dedicated file (DF), e il file elementare (EF). Ogni file, di qualsiasi tipo, viene identificato con 2 byte (ad esempio il MF è sempre identificato con "3F00"). Il Master File è unico e coincide con la directory principale, o meglio "root". Il MF solitamente contiene solo dedicated file più un particolare EF, l'EFMF1, meglio conosciuto come l'EF_ICCID, il quale contiene il numero seriale della SIM che la identifica univocamente. I dedicated file possono essere considerati come contenitori dei EF. Di particolare interesse sono il DF_Telecom, che contiene tutti i dati (EF) più usati della SIM come rubrica (ADN, FDN e SDN) o Sms e il DF_GSM, contenente applicazioni esclusive per le reti GSM. Gli EF infine costituiscono i dati veri e propri, anch'essi sono identificati con 2 byte, ma spesso per accedere ad essi è necessario specificare anche i byte (cioè l'indirizzo) dei rispettivi DF che li contengono e del MF. Il seguente grafico dovrebbe chiarificare la struttura del file system.



Si noti che, sempre all'interno del parckage "com.android.internal.telephony", è presente l'interfaccia "IccConstants" che espone, come costanti, tutti i principali byte identificativi dei file SIM.

File elementari possono avere forme diverse, in particolare si distinguono 4 tipologie di EF:

Trasparente: è un unico record di byte. Per le operazioni di lettura/scrittura è necessario specificare in byte la posizione iniziale (indirizzo di offset) e il numero di byte a partire da essa a cui viene effettuato l'accesso (il primo byte ha l'indirizzo relativo di "0000"). La struttura di un EF traspa-

rente consiste in un header, contenente la lunghezza del file, e da un body contenente i dati.

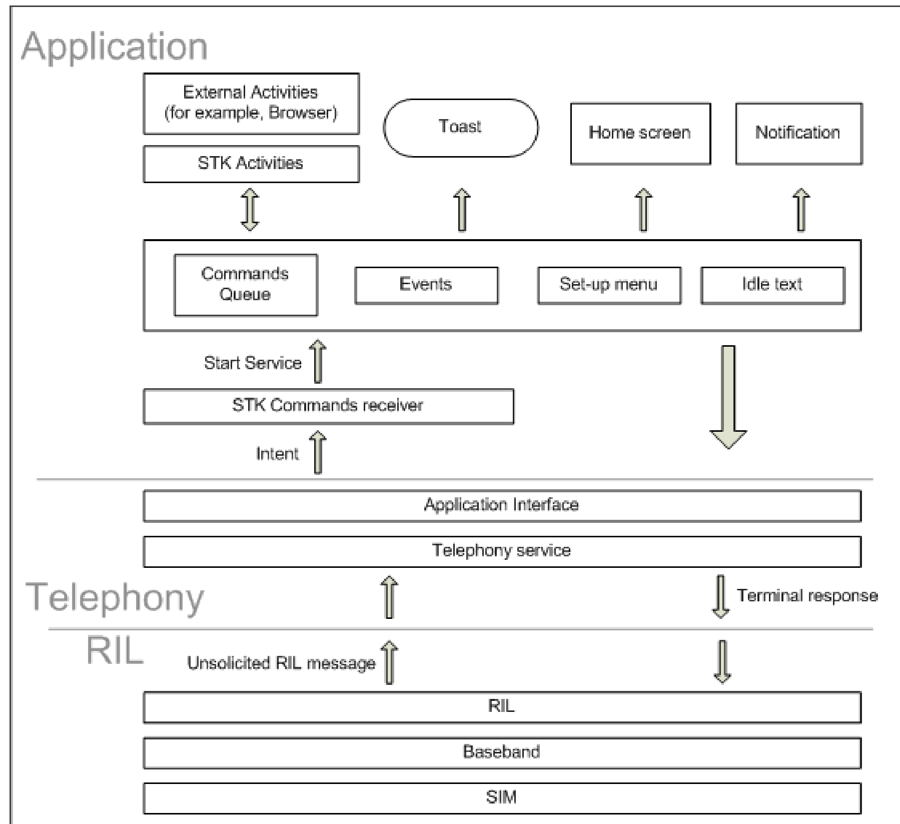
Lineare, con record a lunghezza fissa: come suggerisce il nome, è una sequenza di record di byte a lunghezza fissa. La sua struttura è ancora costituita da un header, contenente la lunghezza di un singolo record e la lunghezza complessiva, e dal corpo costituito dai record, cui è possibile accedere specificandone il numero.

Lineare, con record a lunghezza variabile: come prima, solo che i record sono a lunghezza variabile e questo comporta una maggiore complessità nelle operazioni di lettura/scrittura.

Ciclico: è di fatto un EF lineare con record a lunghezza fissa, con l'unica differenza che c'è un legame tra l'ultimo record e il primo. Questo permette, in operazioni di scrittura sequenziale, di sovrascrivere il primo (cioè quello più "vecchio" in ordine cronologico) record dopo l'ultimo.

Di fatto gli header non contengono solamente la lunghezza del file (o dei record) ma anche altre informazioni che identificano il file stesso come il suo ID, il tipo di file (trasparente, lineare o ciclico), e l'eventuale blocco PIN.

I tre layer rappresentati nella figura sottostante offrono uno schema introduttivo utile a comprendere come Android si interfaccia con la SIM. Fino ad ora ci si è mossi a cavallo tra il primo e il secondo layer, mentre non si è ancora preso in considerazione il terzo, cioè il Radio Interface Layer (RIL).



La RIL di fatto dispone di un'interfaccia che permette la comunicazione tra i servizi e metodi "telephony" di Android con parte dell'hardware dello smartphone, tra cui la Sim Card. Esso è costituito da due componenti principali, interagenti fra di loro: il RIL Daemon, più interno e il Vendor RIL, più esterno, implementato in maniera differente in base alla tecnologia del dispositivo.

Operazioni Avanzate - 2a parte

L'articolo [9] non si limita a fornire una descrizione generale del file system della SIM Card ma propone un test di tipo "Steganografico"[8] volto ad individuare alcuni file EF non utilizzati normalmente dalle applicazioni, definiti come "Nonstandard Parts" (NS) e sfruttarli , in quanto "nascosti", per memorizzare

[8] Per "Steganografia" si intende una particolare scienza volta a studiare metodi di comunicazione attraverso canali nascosti.

dati. L'esperimento, secondo l'articolo, è stato portato avanti mediante due tool: "SIMBrush" e una versione modificata dell'open-source software "GSM Phone Card Viewer". Per ripetere un tentativo simile in questa sede, cioè utilizzando solo Android, è necessario innanzitutto individuare classi e metodi (non pubblici, visto che quelli pubblici evidentemente non lo permettono) che offrano la possibilità di interrogare qualsiasi file EF. Restando all'interno del package "internal.telephony" si possono già dunque scartare classi come AdnRecordLoader in quanto limitati al file EF "ADN" (contenente la rubrica standard).

Invece ci si può focalizzare sugli oggetti PhoneBase, prima introdotti. PhoneBase è una classe astratta, estesa da CDMAPhone e da GSMPhone. Ci si concentri sul secondo. Osservando il codice si nota che la chiamata al costruttore implica la creazione di un oggetto di tipo IccFileHandler, il quale è poi prelevabile attraverso il metodo "getIccFileHandler"[9]. La creazione di un oggetto GSMPhone tramite "reflection" è complessa a causa dei parametri d'ingresso da passare al costruttore, ma è possibile appoggiarsi ai metodi statici della classe PhoneFactory. Il tentativo di accesso ai file della SIM si strutturerà dunque come segue: prima si creerà o preleverà un'istanza di tipo GSMPhone attraverso il metodo statico "getGsmPhone" o "getDefaultPhone", in seguito dalla classe GSMPhone così creata si effettuerà l'accesso al suo SimFileHandler (estensione di IccFileHandler, prelevabile con "getIccFileHandler"). Infine dovrebbe essere possibile (in teoria[10]) invocare da quest'ultimo oggetto metodi come "loadEFLinearFixed" o "updateEFLinearFixed", tutti corrispondenti all'intento esposto all'inizio. Verrà sfruttata la terza Activity dell'applicazione prima creata per l'occasione.

Per prima cosa si può eseguire ingenuamente il seguente codice.

```
phoneFactory = Class.forName
("com.android.internal.telephony.PhoneFactory");

getGsmPhone = phoneFactory.getDeclaredMethod("getGsmPhone", null);

gsmPhone = getGsmPhone.invoke(phoneFactory, null);

Log.i("PROGRESS", gsmPhone.getClass() + " creato");
```

L'esecuzione tuttavia termina in un errore sull'invocazione del metodo (InvocationTargetException) la cui causa è segnalata come NullPointerException. Controllando l'implementazione del metodo da invocare, si nota che la classe PhoneFactory crea manualmente un'istanza di tipo GsmPhone passando come parametri d'ingresso al costruttore oggetti che vengono inizializzati solo attraverso il metodo "makeDefaultPhone". Essendo statico, è probabile in realtà

[9]Se si volesse creare manualmente un oggetto di questo tipo sarebbe comunque necessario passare in ingresso al costruttore un'istanza di tipo PhoneBase.

[10]in pratica alcuni EF sono prevedibilmente protetti a livello SIM (sono nascosti o non sono sovrascrivi). L'articolo [9] indica che buona parte dei file NS, non dispongono di protezione adeguata.

che tale metodo sia già stato chiamato dal sistema, creando tutto il necessario, solo che tali risorse non sono visibili da una normale app creata mediante l'S-DK (che ha appunto un diverso "Context"). Aggiungendo all'app l'invocazione del metodo "makeDefaultPhone" prima del "get", è necessario aggiungere alla nostra applicazione i permessi "WAKE_LOCK" e "WRITE_SETTINGS", ma anche così facendo ci si scontra con questa eccezione molto eloquente:

```
FATAL EXCEPTION: main
java.lang.SecurityException: Permission Denial: not allowed to send broadcast
android.provider.Telephony.SPN_STRINGS_UPDATED from pid=847, uid=10073
```

Andando nel dettaglio, l'istanza di GSMPhone viene creata (è possibile verificarlo tramite Debug o LogCat) e con essa un'istanza (presente come variabile interna) di GsmServiceStateTracker. Quest'ultima in esecuzione riceve una segnalazione dal sistema che lo stato del servizio RADIO del telefono è cambiato (sul source code si nota che il parametro message.what elaborato dal metodo "handleMessage" contiene la costante intera nominata come "EVENT_RADIO_-STATE_CHANGED"). Accertato in seguito che il nuovo stato viene identificato dalla costante intera "RADIO_UNAVAILABLE" (che è diverso da "RA-DIO_OFF"), lo StateTracker si dichiara "fuori servizio" (attraverso la chiamata "newSS.setStateOutOfService()") ed effettua la trasmissione di un particolare Intent.

```
Intent intent = new Intent(Intents.SPN_STRINGS_UPDATED_ACTION);
        intent.addFlags(Intent.FLAG_RECEIVER_REPLACE_PENDING);
        intent.putExtra(Intents.EXTRA_SHOW_SPN, showSpn);
        intent.putExtra(Intents.EXTRA_SPN, spn);
        intent.putExtra(Intents.EXTRA_SHOW_PLMN, showPlmn);
        intent.putExtra(Intents.EXTRA_PLMN, plmn);

        phone.getContext().sendStickyBroadcast(intent);
```

Il broadcast dell'intent non va però a buon fine e in risposta si ottiene l'errore che specifica l'impossibilità da parte della nostra applicazione[11] di inviare quel determinato Intent, il cui tentativo, si ricorda, viene effettuato automaticamente alla creazione di GSMPhone. Ci si trova di fronte quindi ad un oggetto inutilizzabile: anche se non ci fosse questo errore, come si è visto, l'IccFileHandler (che, come si vedrà in seguito, si appoggia ai comandi RIL) probabilmente non funzionerebbe visto che il sistema Radio non è disponibile in questo contesto.

Per convincere il lettore di questo, si può prendere in considerazione la classe che gestisce direttamente i comandi RIL, di nome "RIL.java". Si osservi il codice sorgente di IccFileHandler: l'implementazione di un metodo come "loadEFLinearFixedAll" si appoggia prevedibilmente al RIL.

[11]"pid" sta per Process Id e "uid" sta per User Id

```
public void  loadEFLinearFixedAll(int fileid, Message onLoaded)   {

    Message response = obtainMessage(EVENT_GET_RECORD_SIZE_DONE,
                    new LoadLinearFixedContext(fileid,onLoaded));

    phone.mCM.iccIO(COMMAND_GET_RESPONSE, fileid, getEFPath(fileid),
            0, 0, GET_RESPONSE_EF_SIZE_BYTES, null, null, response);

        }
```

"mCM" è un "CommandsInterface" e nella particolare implementazione di GSMPhone coincide con RIL. Osservando quindi la classe[12] si notano un grande numero di metodi che intuitivamente forniscono un'estesa interazione col layer in questione. "iccIO" è quello più generico, e verrà utilizzato in quest'ultimo test. Per ottenere un'istanza di RIL non è necessario passare per GSMPhone, ma è richiesto in ingresso al costruttore solo un oggetto di tipo Context. Nel seguente codice di esempio si è cercato di accedere ad un file EF classico, cioè ADN, invocando "iccIO" nello stesso modo in cui è invocato in "loadEFLinearFixedAll".

[12]per esteso, "**com.android.internal.telephony.RIL**"

```
package com.example.telephonyTestApp;

import java.lang.reflect.Constructor;


public class ActivityThree extends Activity {

    Context thisContext = this;

     static protected final int COMMAND_GET_RESPONSE = 0xc0;
     static protected final int GET_RESPONSE_EF_SIZE_BYTES = 15;
     static final int EF_ADN = 0x6F3A;
     static final String MF_SIM = "3F00";
     static final String DF_TELECOM = "7F10";

    Class RILCommands;
    Constructor RILCommandsCostructor;
    Method IccIO;
    Message messaggioRIL;
    Handler thisHandler = new Handler();


    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.act3);

        try {

            RILCommands = Class.forName
            ("com.android.internal.telephony.RIL");

            Method[] Allmethods = RILCommands.getDeclaredMethods();

            IccIO = Allmethods[46];

            messaggioRIL = thisHandler.obtainMessage();

            RILCommandsCostructor = RILCommands.getConstructor(Context.class);

            Object RILCommandsObject = RILCommandsCostructor.newInstance(thisContext);

            Log.i("PROGRESS", "Invocazione terminata con successo");
            Log.i("PROGRESS", IccIO.getName() + " trovato");


            Object[] param = new Object[9];

            param[0] = COMMAND_GET_RESPONSE;
            param[1] = EF_ADN;
            param[2] = MF_SIM + DF_TELECOM;
            param[3] = 0;
            param[4] = 0;
            param[5] = GET_RESPONSE_EF_SIZE_BYTES;
            param[6] = null;
            param[7] = null;
            param[8] = messaggioRIL;

            IccIO.invoke(RILCommandsObject, param);

            Log.i("PROGRESS", "IccIO invocato senza errori"); }

        [...]
```

Si commenta ora il codice. Siccome il tentativo di prelevare il metodo "iccIO" non ha avuto successo utilizzando la procedura standard ("per nome"), sono stati memorizzati (sempre tramite "reflection") tutti i metodi e poi dall'array risultante si è visto, grazie al tool di Debug di Eclipse, che l'elemento cercato era il numero 46. I comandi e gli Id dei file SIM utilizzati sono stati riportati (leggi "copia-incollati") dall'interfaccia IccConstants. Il terzo parametro in ingresso al metodo invocato alla fine, coincide con il "path" dell'EF, ottenibile con la concatenazione dell'id del MF e del DF che lo contiene. L'ultimo parametro coincide infine con un Message che va prelevato direttamente da un Handler[13]. Il metodo non ritorna niente ma, a invocazione terminata, il messaggio viene modificato: all'interno del suo campo obj viene memorizzata un'istanza di AsyncResult contenente a sua volta nel campo "result" un "ArrayList<byte[]>" corrispondente al file elementare da leggere. L'esecuzione del codice non genera errori, il metodo viene invocato correttamente. Tuttavia se si controlla con precisione, sempre usando il tool di Debug, il contenuto di messaggioRIL ci si rende conto che si è ripresentato un problema già affrontato prima.

[13]l'Handler è un particolare oggetto che gestisce code di messaggi (Message) associati al thread in esecuzione.

Name	Value
this	ActivityThree (id=829748224656)
Allmethods	Method[161] (id=829748060824)
RILCommandsObject	RIL (id=829748153408)
savedInstanceState	null
param	Object[9] (id=829748170560)
[0]	Integer (id=829748170704)
[1]	Integer (id=829748170720)
[2]	"3F007F10" (id=829748170736)
[3]	Integer (id=829644583168)
[4]	Integer (id=829644583168)
[5]	Integer (id=829644583408)
[6]	null
[7]	null
[8]	Message (id=829748076712)
arg1	0
arg2	0
arg3	0
callback	null
data	null
next	Message (id=829748074368)
obj	AsyncResult (id=829748175648)
cause	CommandException (id=829748172056)
detailMessage	"RADIO_NOT_AVAILABLE" (id=829748172392)
count	19
hashCode	498848441
offset	0
value	(id=829748172424)
e	CommandException$Error (id=829748172368)
stackState	(id=829748174440)
stackTrace	null
result	null
userObj	null
replyTo	null
target	Handler (id=829748225832)
what	0
when	4460937

Il campo "result" è null, in compenso il messaggio contiene l'errore "CommandException: RADIO_NOT_AVAILABLE". Controllando sempre l'implementazione della classe RIL, si potrebbe essere indotti ad utilizzare il metodo "setRadioStateFromRILInt", peccato che esso sia privato ed inaccessibile tramite

"reflection" (non è presente nell'array "Allmethods"[14] e cercando di prelevarlo "per nome" viene lanciata una NullPointerException).

[14]neanche utilizzando "getMethods" al posto di "getDeclaredMethods", che restituisce più metodi

Capitolo 4

Conclusioni

Il risultato non completamente positivo, esposto alla fine del capitolo 3, di avere più controllo sui file elementari della SIM (replicando magari il test esposto nell'articolo [9]) è dovuto alla protezione hardcoded del sistema operativo, o almeno delle sue versioni standard installate nei dispositivi commerciali. La nostra applicazione ha cercato di accedere a risorse (i comandi RIL) non disponibili all'SDK, utilizzabili solo dai processi di sistema, e neanche lo strumento della "reflection" è riuscito a scavalcare l'ostacolo. Non serve a niente (per i motivi esposti alla fine del capitolo 1) attribuire alla nostra app un "Uid" di tipo "system" perchè essa non sarebbe nemmeno installabile nel dispositivo[1]. Tutto questo d'altra parte è un punto a favore del sistema operativo Google, se infatti si fosse riusciti a leggere e scrivere in maniera quasi completa gli EF della SIM, ci si troverebbe di fronte ad una falla dell'OS sfruttabile anche in maniera dannosa.

Test simili sono comunque possibili ma esclusivamente utilizzando un developer phone[2], ossia telefoni da sviluppatori che mettono a disposizione il codice sorgente della piattaforma e su cui è possibile installare/aggiornare versioni opportunamente modificate dell'OS. Alcune di esse tra l'altro dispongono di "platform" key e certificate per signare le app. Queste, pur avendo accesso a più privilegi (l'esperimento, in questo caso, potrebbe dare un esito positivo), comunque funzionerebbero solamente sulle relative distribuzioni personalizzate.

Il contesto universitario su cui ci si è mossi non disponeva di dispositivi simili, dunque tutte le prove sopra presentate sono state effettuate su un telefono disponibile sul mercato di massa[3], con installata una versione standard di Android (2.2).

Per ulteriori ricerche nel campo introdotto dall'elaborato si suggerisce di acquisire un developer phone e di costituire un opportuno gruppo di lavoro. E'

[1] anche qui l'errore è eloquente "INSTALL_FAILED_SHARED_USER_INCOMPATIBLE"

[2] Questi sono acquistabili tramite un account Google. Oppure in alternativa è possibile rendere tale uno smartphone normale, tuttavia la procedura è complessa e potenzialmente molto dannosa.

[3] per la precisione un Acer Liquid E, "rootato" per l'occasione

bene ricordare che gli EF relativi ai contatti telefonici e agli SMS salvati sono comunque accessibili sia in lettura che in scrittura tramite Content Provider.

Bibliografia

[1] Buona parte della descrizione dell'OS, nonchè dell'analisi delle sue *componenti* (Activity, Intent, ecc...) è basata sul testo "Android - Guida per lo sviluppatore" di Massimo Carli (Apogeo) e su "Application Fundamentals" presente nella documentazione ufficiale (http://developer.android.com/guide/topics/fundamentals.html)

[2] http://www.watblog.com/2010/02/06/symbian-os-now-fully-open-source/

[3] http://www.tulsaworld.com/business/article.aspx?subjectid=52&articleid=20100813_52_E2_Google299586

[4] Per maggiori informazioni, si consulti il sito ufficiale del progetto "MeeGo", http://meego.com/

[5] http://www.bada.com/

[6] Il sito del progetto "Taintdroid" http://appanalysis.org/index.html, e nello specifico la pubblicazione dello studio "http://appanalysis.org/tdroid10.pdf"

[7] La procedura di creazione dei certificati e "key" e del conseguente "sign" (firma) è liberamente consultabile nel sito ufficiale di android al seguente indirizzo http://developer.android.com/guide/publishing/app-signing.html

[8] Il codice sorgente di Android è liberamente scaricabile all'indirizzo http://android.git.kernel.org/, mediante il tool GIT (solo per parti di codice) o lo script Repo, funzionante solo in ambito Unix (per scaricare l'intera piattaforma). In alternativa è possibile consultare il sito http://grepcode.com/, il quale dispone, oltre del source di java, di praticamente tutto il source code di Android (dalla versione 1.5 fino alla 2.3).

[9] "*Data Hiding in SIM/USIM Cards: A Steganographic Approach*" di Antonio Savoldi e Paolo Gubian.