Università degli Studi di Padova

Corso di Laurea Specialistica in Ingegneria Informatica

REALIZZAZIONE DI UN TERMINALE AUDIOCITOFONICO SIP SU PIATTAFORMA ANDROID

RELATORE: Chiar.mo Prof. Lorenzo Vangelista

LAUREANDO: Giuseppe Soldo

Anno Accademico
2010-2011

Indice

Introduzione **5**

1 Android **9**

1.1 Introduzione 9
1.2 Architettura 10
1.2.1 Kernel 10
1.2.2 Librerie Native 11
1.2.3 Dalvik Virtual Machine 13
1.2.4 Core Library 14
1.2.5 Application Framework 14
1.3 Applicazioni 15
1.3.1 Ciclo di vita dei componenti 17
1.3.2 Processi 21
1.3.3 Android User Interface 22
1.3.4 Il file manifest 23
1.4 Android SDK 26
1.4.1 Emulatore 27

2 Installazione di Android sulla I.MX51 BBG board **31**

2.1 I.MX51 BBG board 31
2.1.1 Caratteristiche 31
2.1.2 Hardware e periferiche 32
2.2 Preparazione dell'ambiente di sviluppo 35
2.2.1 Scaricare i sorgenti 36
2.2.2 Installare e configurare il server NFS 36
2.2.3 Configurare il server TFTP 37
2.2.4 Configurare il minicom 38
2.3 Compilazione dei sorgenti 38
2.3.1 Compilare l'U-boot 39
2.3.2 Compilare il kernel 40
2.3.3 Compilare Android 40

2.4 Linux arm boot process 42
2.5 Programmare il boot loader 45
2.6 Booting Android 48
2.6.1 TFTP/NFS 50
2.6.2 SD 53
2.6.3 Modalità video touch-screen 56
2.6.4 Problemi e soluzioni 58

3 Protocollo SIP 61
3.1 Introduzione al VOIP 61
3.2 Protocollo SIP 62
3.2.1 Struttura del protocollo 63
3.2.2 Protocolli di supporto 63
3.2.3 Messaggi 65
3.2.4 Componenti 69
3.3 MJSip 74
3.3.1 Architettura 75
3.3.2 Livello Transport 77
3.3.3 Livello Transaction 78
3.3.4 Livello Dialog 80
3.3.5 Livello Call 82

4 SipDroid 87
4.1 Introduzione 87
4.2 Struttura 89
4.2.1 UserAgent 90
4.2.2 SipdroidEngine 94
4.2.3 Receiver 95
4.2.4 Sipdroid 98
4.3 Compilazione e installazione di Sipdroid 103
4.4 Asterisk 104
4.4.1 File di configurazione 105
4.4.2 Installazione e configurazione del server Asterisk 109

5 Sipbell 115
5.1 Introduzione 115
5.2 Interfaccia principale 115
5.2.1 Interfaccia “CALL” 118
5.2.2 Menu opzioni 120
5.3 Sicurezza 123
5.3.1 Autenticazione 123

5.3.2 ALERT CALL 126
5.3.3 Inserimento di una nuova password 129
5.4 Installazione 132
5.5 Ulteriori caratteristiche 135

6 Conclusioni **139**

Bibliografia **141**

Introduzione

Grazie alla rapida diffusione delle connessioni a banda larga, la crescita del VoIP ha assunto dimensioni consistenti avvalendosi di applicazioni semplici da utilizzare che permettono di effettuare chiamate in tutto il mondo gratuitamente o affrontando una spesa estremamente contenuta. Le ragioni per le quali il VoIP è estremamente conveniente sono da ricercarsi proprio nell'infrastruttura della rete IP che è basata sulla commutazione di pacchetto che, rispetto alla commutazione di circuito, ovvero la normale linea PSTN, consente notevoli risparmi economici.

La convenienza dell'utilizzo della tecnologia VoIP rispetto alla tradizionale linea PSTN può essere riassunta nelle seguenti caratteristiche:

- a parità di qualità audio è richiesta una minore banda in quando il VoIP non necessita di una preallocazione di risorse come lo stabilire in fase di inizializzazione un circuito dedicato tra due end-point;

- riduzione dei costi di chiamata, specialmente per le chiamate a lunga distanza poichè gli unici costi sono quelli relativi al collegamento verso il proprio provider Internet;

- portabilità del numero a prefisso geografico: il numero infatti non è più legato fisicamente a uno specifico apparecchio. In poche parole è possibile, anche quando si è in vacanza o si è fuori sede, essere raggiungibili sul numero del telefono di casa.

In principio il VoIP veniva utilizzato come strumento di comunicazione vocale tra normali PC connessi alla rete Internet. Successivamente, grazie alla sua diffusione e ad una maggiore domanda sono stati immessi sul mercato specifici apparecchi telefonici (che hanno notevoli analogie con i comuni cordless) che possono essere collegati direttamente al computer (generalmente attraverso la porta USB) e consentono di utilizzare il servizio telefonico tramite una modalità molto simile a quella tradizionale. Per ovviare al problema della necessità di disporre di un computer acceso e collegato alla rete internet per iniziare una conversazione telefonica le moderne tecnologie hanno permesso di

scindere i due dispositivi permettendo l'utilizzo del VoIP anche in assenza di un computer. Per accedere a questa configurazione è sufficiente disporre di una connessione internet a banda larga e di un telefono collegato direttamente al modem/router ADSL. Stiamo parlando del telefono VoIP, un apparecchio sostanzialmente analogo ad un telefono normale con la sola differenza che si collega ad un router anzichè alla linea telefonica tradizionale.

Con la possibilità di poter disporre di un collegamento verso la rete internet anche mediante i moderni dispositivi mobili si è reso possibile l'utilizzo della tecnologia VoIP anche sui telefoni cellulari o smartphone eliminando quella che poteva essere considerara una delle maggiori pecche del voice over ip, ovvero la mobilità. Il dispositivo, dotato di connettività UMTS, 3G o più semplicemente WI-FI, con l'utilizzo di un client VoIP, può essere utilizzato per effettuare telefonate con una mobilità pari a quella di un cellulare.

Inoltre, grazie all'introduzione da parte dei provider Internet di gateway dedicati, si è potuto creare un ponte di collegamento tra la rete a commutazione di pacchetto e quella a commutazione di circuito permettendo a client VoIP di comunicare con i dispotitivi connessi alla rete PSTN e GSM inserendo il loro utilizzo in un contesto più ampio.

Nell'ambito di questo progetto di tesi si è cercato di sfruttare la tecnologia VoIP per un utilizzo diverso da quello tradizionale, ovvero l'audiocitofonia remota. L'idea base è quella di creare un terminale audiocitofonico in grado di interfacciarsi alla rete internet e di sfruttare le moderne tecnologie per permettere all'utente che utilizza il citofono di rintracciare il proprietario/inquilino dell'abitazione ovunque esso sia con costi irrisori. Diversamente dalla telefonia tradizionale non è richiesto al chiamante alcuna conoscenza sul numero di telefono. Infatti, come in un normale citofono, l'inizializzazione della chiamata avviene semplicemente attraverso la pressione di un tasto. Questo permette a chiunque di mettersi in contatto con l'interessato anche se quest'ultimo non è presente in casa o se non si è a conoscenza del suo numero di telefono.

Diversi sono i protocolli che permettono, attraverso le loro implementazioni, il concretizzarsi di una telefonata che sfrutti la tecnologia IP. I più importanti, nonchè i più diffusi, sono certamente il protocollo SIP e SKYPE. Il primo definisce uno standard aperto che sfrutta molteplici protocolli per l'instaurazione di una sessione e il seguente scambio di contenuti multimediali. Il secondo rappresenta un protocollo chiuso, che sfrutta una rete peer-to-peer costituita da molteplici client che condividono risorse computazionali per il rintracciamento dell'utente da contattare con il quale iniziare lo scambio di flussi multimediali. La scelta relativa al protocollo da utilizzare è caduta sul Session Initiation Protocol. Le motivazioni che hanno determinato questa scelta sono da ricercarsi nella semplicità implementativa e nello sfruttamento di tecnologie open, ovve-

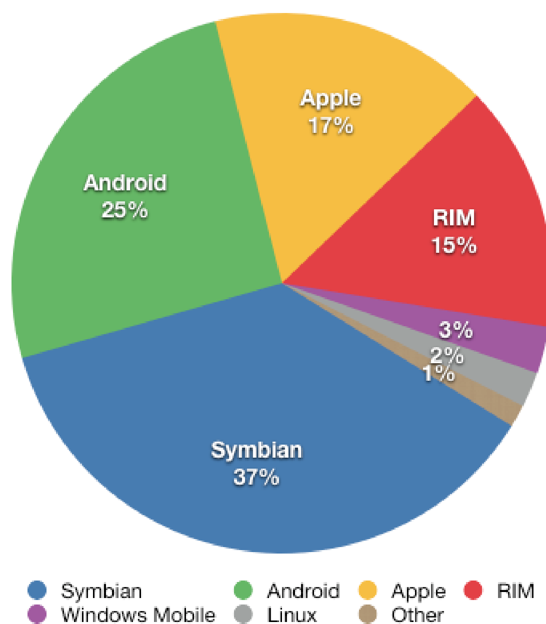


Figura 1: Vendite degli smartphone nel terzo quadrimestre del 2010. Dati forniti da gartner.com

ro tecnologie in continuo sviluppo che permettono di ampliare i servizi offerti con nuove funzionalità.

Open è il client SIP utilizzato così come la piattaforma sulla quale è stato installato. Per quest'ultima si è scelto di utilizzare il sistema operativo Google Android, uno stack comprensivo di strumenti utili che permettono ad uno sviluppatore di progettare ed eseguire applicazioni ricche di funzionalità. Rilasciato dalla Open Handset Alliance (OHA) verso la fine del 2008 con una versione integrata in un dispositivo per permettere agli sviluppatori di testare le proprie applicazioni, ben presto Android ha allargato i propri orizzonti affacciandosi al grande pubblico con il rilascio della versione 1.5 avvenuta nell'aprile 2009. Nonostante sia un sistema giovane, nel giro di pochi mesi Android ha scalato le classifiche di vendita posizionandosi al secondo posto dietro a un colosso come Symbian ma soprattutto avanti al sistema operativo dell'azienda di Cupertino, l'IOS per gli Apple IPhone. Allo stato dell'arte si è giunti alla versione 2.2 denominata **Froyo** ma è imminente l'uscita della versione 2.3 (**Gingerbread**) che a detta degli sviluppatori porterà importanti innovazioni come per esempio i supporto per la tecnologia **Near Field Communication**, un sistema di trasmissione wireless a cortissimo raggio (fino a 4 cm), che potrà sostituire in futuro le normali carte di credito.

Il punto di forza del sistema Android si fonda sull'utilizzo di un kernel Linux nella versione 2.6 che rende l'apparato software stabile e soprattutto sicuro, elemento da non sottovalutare per un sistema creato per dispositivi mobili in grado di interfacciarsi alla rete internet per la normale navigazione web e per il download e l'installazione di nuove applicazioni. Inoltre viene fornito un nuovo sistema per la gestione del risparmio energetico volto a fovorire l'allungamento della carica della batteria di un cellulare.

L'ambito di utilizzo di Android non si ferma allo smartphone ma trova una giusta collocazione anche all'interno di tablet o più in generale in sistemi embedded. La sua diffusione per questi dispositivi la si deve soprattutto al rilascio del codice con licenza open source che permette ad un qualsiasi vendor di portare il sistema sulla propria piattaforma hardware. Tra i vendor di dispositivi ricordiamo la società **Freescale** che ha portato Android sulla board con application processor I.MX51 per permettere allo sviluppatore di progettare e testare applicazioni su un hardware avanzato per soluzioni embedded.

Nel primo capitolo della tesi si è cercato di dare una descrizione più approfondita dello stack Android con particolare riferimento alla sua struttura, alle modalità che permettono ad uno sviluppatore di creare un'applicazione e agli strumenti messi a disposizione dalla OHA per il testing e il debugging dell'applicativo.

Nel secondo capitolo è stato introdotto l'application processor I.MX51 e sono stati descritti tutti i passi necessari alla corretta installazione del sistema operativo Android sulla board di sviluppo.

Nel terzo capitolo l'attenzione è stata rivolta al protocollo SIP e allo stack MjSip, una libreria sviluppata dal professore Luca Veltri dell'Università di Parma che implementa l'intero apparato SIP così come è stato descritto nella RFC 3261. Scritta in java permette ad un programmatore di sviluppare applicazioni operanti con il protocollo SIP interfacciandosi su di esso con API di alto livello facilmente estandibili e personalizzabili. MjSip è il core dell'applicazione Sipdroid, client SIP open source sviluppato esclusivamente per la piattaforma Android. Sipdroid, così come le modalità di installazione e la realizzazione di un semplice test realizzato con il centralino Asterisk, è stato descritto nel quarto capitolo.

Nel quinto capitolo sono state illustrate le scelte implementative utilizzate nella realizzazione del software Sipbell, versione modificata di Sipdroid per rendere possibile un'audiocitofonia che sfrutti il protocollo SIP.

Capitolo 1

Android

1.1 Introduzione

Android è un vero e proprio stack di strumenti e librerie per la realizzazione di applicazioni mobili. Esso ha come obiettivo quello di fornire tutto ciò di cui un operatore, un vendor di dispositivi o uno sviluppatore hanno bisogno per raggiungere i propri obiettivi. Android ha la fondamentale caratteristica di essere open, dove il termine assume diversi significati.

- Android è open in quanto utilizza tecnologie open come il kernel di Linux nella versione 2.6.

- Android è open in quanto le librerie e le API utilizzate per la sua realizzazione sono le stesse che verranno utilizzate per lo sviluppo di nuove applicazioni. Questo permette allo sviluppatore di rimpiazzare la quasi totalità dei componenti di Android con i propri rendendo la personalizzazione del sistema quasi totale.

- Android è open in quanto il suo codice è open source, consultabile da chiunque possa contribuire a migliorarlo, lo voglia documentare o semplicemente voglia scoprirne il funzionamento. La licenza scelta dalla Open Handset Alliance è la Open Source Apache License 2.0, che permette ai diversi vendor di costruire su Android le proprie estensioni anche proprietarie senza legami che ne potrebbero limitare l'utilizzo. Ciò significa che non bisogna pagare alcuna royalty per l'adozione di Android sui propri dispositivi.

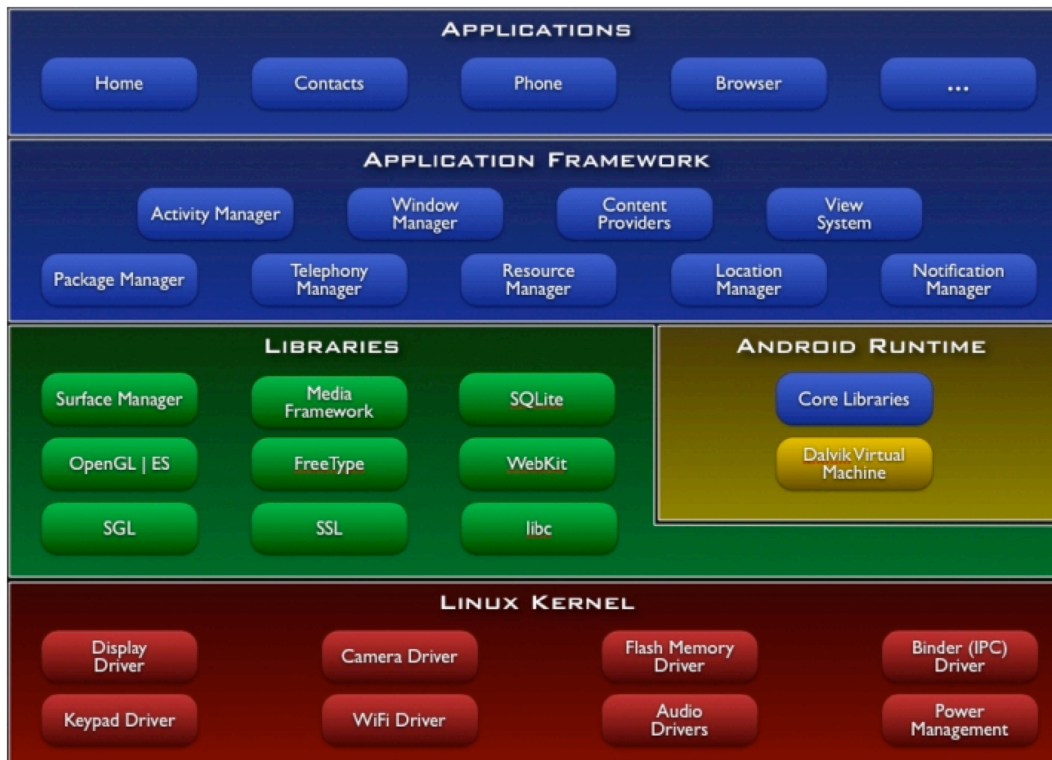


Figura 1.1: Architettura software di Android

1.2 Architettura

L'architettura di Android si presenta suddivisa in diversi livelli come mostrato in figura1.1.

1.2.1 Kernel

Alla base dello stack Android troviamo un kernel Linux nella versione 2.6. La scelta di una simile configurazione è nata dalla necessità di disporre di un vero e proprio sistema operativo che fornisse gli strumenti di basso livello per la virtualizzazione dell'hardware sottostante attraverso l'utilizzo di diversi driver. A differenza di un kernel Linux standard per Android sono stati aggiunti ulteriori moduli come:

- **Binder IPC Driver**, un driver dedicato che permette a processi di fornire servizi ad altri processi attraverso un insieme di API di più alto livello rispetto a quelle presenti su un kernel Linux standard, ciò permette la comunicazione tra processi con un costo computazionale minore.

- **Low memory killer**, sistema che si preoccupa di "uccidere" i processi liberando così spazio nella memoria centrale per soddisfare le richieste di un altro processo. Ad ogni processo viene assegnato un punteggio e il processo che detiene il punteggio più alto verrà scelto come candidato per un'eventuale eliminazione. Questi punteggi vengono assegnati in funzione dell'importanza di un processo, per esempio il processo che controlla la *user interface* (UI) di un'applicazione che per il momento non è visibile sullo schermo ha un punteggio superiore rispetto al processo che gestisce la UI che in questo momento è in primo piano. Il processo init non può essere ucciso.

- **Ashmen**, sistema di memoria condiviso anomimo (Anonymous Shared Memory) che definisce interfacce che consentono ai processi di condividere zone di memoria attraverso un nome. Per esempio il sistema potrebbe utilizzare Ashmen per memorizzare delle icone che possono essere utilizzate da più processi al momento dell'elaborazione della loro interfaccia utente. Il vantaggio di Ashmem rispetto ai sistemi Linux che fanno uso della tradizionale memoria condivisa è che fornisce un mezzo al kernel di recuperare questi blocchi di memoria se non sono attualmente in uso. Se un processo tenta di accedere a un blocco di memoria condivisa che il kernel ha liberato, verrà generato un errore, e sarà quindi necessario riallocare il blocco e ricaricare i dati.

- **RAM console e log devices**, per agevolare il debug, Android fornisce la capacità di memorizzare i messaggi di log generati dal kernel in un buffer RAM. Inoltre, Android fornisce un modulo separato che può essere utilizzato dai processi utente per leggere e scrivere messaggi di log.

- **Android Debug Bridge**, protocollo che permette di gestire in maniera versatile un'istanza dell'emulatore o un dispositivo reale.

- **Power Management**, progettato per permettere alla CPU di non consumare energia se nessuna applicazione o servizio ne fa richiesta.

1.2.2 Librerie Native

Sopra il layer costituito dal kernel di Linux 2.6 abbiamo un livello che contiene un insieme di librerie native realizzate in C e C++, che rappresentano il core vero e proprio di Android.

- **Surface Manager**, modulo che ha la responsabilità di gestire le view, ovvero le componenti fondamentali di un'interfaccia grafica. Ogni appli-

cazione attiva viene eseguita da un processo diverso e quindi più applicazioni attive disegnano la propria interfaccia in tempi diversi. Il compito del SM è di prendere le diverse finestre e di disegnarle sul buffer da visualizzare per evitare l'accavallamento tra le stesse.

- **Open GL ES**, librerie utilizzate per la grafica 3D, permettono di accedere alle funzionalità di un eventuale acceleratore grafico hardware. Il target di dispositivi cui queste API sono rivolte è caratterizzato da quantità ridotte di memoria. Si tratta infatti di API con un basso consumo di memoria, che può andare da 1 a 64 MB.

- **SGL**, la Scalable Graphics Library (SGL) è una libreria in C++ che insieme alle OpenGL costituisce il motore grafico di Android. Mentre per la grafica 3D ci si appoggia all'Open GL, per quella 2D viene utilizzato un motore ottimizzato chiamato appunto SGL. Si tratta di una libreria utilizzata principalmente dal Window Manager e dal Surface Manager all'interno del processo di renderizzazione grafica.

- **Media Framework**, componente in grado di gestire i diversi CODEC per i vari formati di acquisizione e riproduzione audio e video. É basato sulla libreria open source OpenCore di PacketVideo. I codec gestiti dal Media Framework permettono la gestione dei formati più importanti tra cui MPEG4, H.264, MP3, AAC, AMR oltre a quelli per la gestione delle immagini come JPG e PNG.

- **FreeType**, motore di rendering per la gestione dei font, molto utile nella definizione di interfaccia. É di piccole dimensioni, molto efficiente, altamente customizzabile e soprattutto portabile. Permette la visualizzazione di immagini ad alta qualità. Di default, i principali tipi di font utilizzati sono il TrueType, Type1, X11 PCF e OpenType insieme ad altri più recenti.

- **SQLite**, una libreria in-process che implementa un DBMS relazionale caratterizzato dal fatto di essere molto compatto, diretto, di non necessitare di alcuna configurazione e soprattutto essere transazionale. SQLite è compatto in quanto realizzato completamente in C in modo da utilizzare solo poche delle funzioni ANSI per la gestione della memoria. È diretto in quanto non utilizza alcun processo separato per operare ma "vive" nello stesso processo dell'applicazione che lo usa, da cui il termine in-process.

- **WebKit**, framework utilizzato dal browser preinstallato sulla piattaforma. Si tratta di un browser engine open source basato sulle tecnologie

HTML, CSS, JavaScript e DOM che può essere integrato su diversi tipi di applicazioni.

- **SSL**, libreria per la gestione dei Secure Socket Layer. Permette una comunicazione sicura e una integrità dei dati su reti TCP/IP come, ad esempio, internet cifrando la comunicazione dalla sorgente alla destinazione sul livello di trasporto.

- **Libc**, implementazione della libreria standard C ottimizzata per i dispositivi basati su Linux embedded.

1.2.3 Dalvik Virtual Machine

Nata dall'esigenza di eseguire del software su macchine con risorse limitate la Dalvik virtual machine è un'ottimizzazione della macchina virtuale della Sun Microsystem che permette di sfruttare al massimo le caratteristiche del sistema operativo ospitante. La Dalvik Virtual Machine è in grado di eseguire codice contenuto all'interno di file con estensione .dex ottenuti a partire dal bytecode Java. I file con estensione .dex hanno un tipo di compressione che permette di dimezzare la spazio adibito alla loro memorizzazione rispetto ai file jar non compressi. La riduzione dello spazio utilizzato deriva dal fatto che stringhe e costanti presenti in più classi vengono incluse solo una volta nel singolo file .dex.

Similmente alla JVM la Dalvik implementa un garbage collector liberando lo sviluppatore dal compito di gestire in prima persona la memoria, ma non implementa alcun Just In Time Compiler, ovvero quel meccanismo attraverso il quale la JVM riconosce determinati pattern di codice Java e li traduce in altrettanti frammenti di codice nativo (C e C++) per una loro esecuzione più efficiente. Questo avviene perchè molte delle funzionalità di Android vengono già implementate in modo nativo dalle librerie di sistema in quanto le diverse API Java del core library sono implementate come oggetti che, mediante il Java Native Interface, utilizzano metodi scritti in codice nativo.

Attraverso l'utilizzo del meccanismo register based per la generazione del codice c'è una riduzione del 30% del numero di operazioni da eseguire rispetto allo stesso codice Java a discapito di una maggior elaborazione in fase di compilazione. Ultima importantissima caratteristica della DVM è quella di permettere una efficace esecuzione di più processi contemporaneamente. Infatti, ciascuna applicazione sarà in esecuzione all'interno del proprio processo Linux; ciò comporta dei vantaggi dal punto di vista delle performance e allo stesso tempo alcune implicazioni dal punto di vista della sicurezza.

1.2.4 Core Library

Per eseguire un'applicazione in Java servono tutte le classi relative all'ambiente nella quale la stessa viene eseguita. Per quello che riguarda la J2SE stiamo parlando delle classi contenute nel file rt.jar, ovvero quelle relative ai package java e javax. Per le applicazioni Android vale lo stesso discorso, con la differenza che in fase di compilazione avremo bisogno del jar (di nome android.jar) per la creazione del bytecode Java, mentre in esecuzione il device metterà a disposizione la versione dex del runtime che costituisce appunto la core library. Nel dispositivo non c'e' infatti codice Java, in quanto non potrebbe essere interpretato da una JVM, ma solamente codice dex eseguito dalla DVM.

1.2.5 Application Framework

L'application framework è costituito da un'insieme di API che sfruttano le librerie sottostanti nello stack Android. Permettono allo sviluppatore di realizzare applicazioni estremamente ricche e innovative.

- **Activity Manager**, modulo che si occupa della gestione delle activity, ovvero quelle entità che sono associate a una schermata e che quindi permettono all'utente finale di interagire con l'applicazione. Il suo compito è quello di gestire il ciclo di vita delle activity e di organizzare le loro schermate in uno stack secondo l'ordine di visualizzazione sullo schermo.

- **Package Manager**, gestisce il processo di installazione e rimozione di un'applicazione.

- **Window Manager**, astrazione attraverso le API Java dei servizi nativi del Surface Manager, si occupa della gestione delle finestre di applicazioni differenti eseguite da processi differenti.

- **Telephony Manager**, permettono l'interazione con le caratteristiche prorie di un telefono come iniziare una chiamata o controllare lo stato della stessa.

- **Content Provider**, gestisce la condivisione di informazioni tra i vari processi. Il suo utilizzo è simile a quello di un repository dove i vari processi hanno la possibilita di leggere e scrivere informazioni.

- **Resource Manager**, gestisce le informazioni relative a un'applicazione quali file di configuazione, file di definizione di layout, immagini utilizzate per la personalizzazione dell'interfaccia grafica e così via.

- **View System**, gestisce l'insieme delle viste utilizzate nella costruzione dell'interfaccia grafica di un'applicazione come bottoni, griglie, text boxes, etc.

- **Location Manager**, mette a disposizione delle API utilizzabili da quelle applicazioni che tra le altre cose si occupano della localizzazione.

- **Notification Manager**, rappresentano un insieme di strumenti utilizzabili dalle applicazioni per notificare eventi al dispositivo che reagirà in conseguenza della notifica ricevuta.

1.3 Applicazioni

All'ultimo livello dello stack architetturale di Android troviamo le applicazioni. Scritte in linguaggio Java, si presentano sotto forma di pacchetti con estensione .apk che incapsulano, oltre al codice java compilato, dati e risorse necessarie all'applicazione stessa.

Ogni applicazione viene eseguita all'interno di un processo Linux che viene avviato nel momento in cui il codice dell'applicazione deve essere eseguito. Ogni processo, a sua volta, viene eseguito all'interno di una propria virtual machine, isolando in questo modo il codice di due applicazioni in esecuzione nello stesso momento. Ad ogni applicazione viene assegnato un Linux user ID, questo permette di definire i permessi in modo tale che il codice, i dati e le risorse appartenenti ad un'applicazione siano visibili soltanto all'applicazione stessa a meno che non venga esplicitato il contrario.

Infatti è possibile fare in modo che due o più applicazioni condividano lo stesso user ID e che quindi detengano gli stessi permessi sui loro dati. Applicazioni che condividono lo user ID possono essere eseguite dallo stesso processo e quindi nella stessa VM.

Grazie ai permessi un'applicazione può far uso di una componente di un'altra applicazione. Tutto ciò è molto utile perchè permette allo sviluppatore di riutilizzare codice di altre applicazioni senza doverlo implementare nella propria.

Esistono quattro tipologie di componenti:

- **Activities**: un activity è un'interfaccia grafica attraverso la quale un utente può interagine con l'applicazione. A ciascuna activity viene data una finestra di default nella quale può essere disegnata. I contenuti visivi di ciascuna finestra vengono realizzati attraverso una gerarchia di viste (views), ciascuna delle quali controlla un particolare spazio rettangolare all'interno della finestra. Questa gerarchia può essere vista come un

albero dove viste genitori contengono e controllano la disposizione delle viste figlie. Le viste foglie invece rispondono direttamente alle azioni dell'utente. Una gerarchia di viste è collocata all'interno di una finestra activity attraverso il metodo **Activity.setContentView()**. Il content view è l'oggetto vista radice della gerarchia.

- **Services**: sono dei task che vengono eseguiti in background per un indefinito periodo di tempo. Un service può essere ad esempio della musica che viene suonata in background mentre l'utente non sta interagendo con un media player ma stà facendo dell'altro. Android mette a disposizione delle interfacce attravero le quali un'applicazione può connettersi a un determinato servizio e usufruirne in base alla definizione dell'interfaccia stessa. Nell'esempio della musica l'interfaccia può mettere a disposizione comandi come play, stop e pause.

- **Broadcast receivers**: è un componente che ha il compito di eseguire delle azioni solo in seguito alla ricezione di un determinato evento. Un evento può essere ad esempio la notifica del basso livello di energia della batteria, il cambio di fuso orario, etc. Un'applicazione può avere più di un broadcast receiver per rispondere agli annunci che si reputano importanti. Ogni ricevitore estende la classe **BroadcastReceiver**.

- **Content providers**: permette la condivisione di dati tra le applicazioni. Un content provider estende la classe **ContentProvider** al fine di implementare un insieme stendard di metodi che permettono ad altre applicazione di recuperare e memorizzare i dati che esso gestisce.

Diversamente dal Content Provider, che viene attivato in seguito a una richiesta da parte del Content Resolver, le altre tre componenti vengono attivati da messaggi asincroni chiamati *intent*.

Ci sono più modi per attivare ciascun componente:

- un'activity può essere attivata passando un oggetto di tipo Intent ai metodi **Context.startActivity()** oppure **Activity.startActivityForResult()**. L'activity può ottenere informazioni sull'intent che la ha attivata invocando il metodo **getIntent()**.

- un service viene attivato passando un oggetto di tipo intent al metodo **Context.startService()**. Un intent può anche essere passato al metodo **Context.bindService()** per stabilire una connessione tra la componente chiamante e il servizio che in questo caso riceve un oggetto intent passato al metodo **onBind()**.

- un'applicazione può iniziare un broadcast passando un oggetto intent ai metodi **Context.sendBroadcast()**, **Context.sendOrderedBroadcast()**, e **Context.sendStickyBroadcast()**. Il sistema consegnerà gli intent a tutti i ricevitori interessati invocando il loro metodo **onReceive()** ereditato dalla classe BroadcastReceiver.

Ciascuna componente deve poter essere disattivata nel momento in cui diventa non necessaria. Per i content provider, che vengono attivati solo in risposta ad una richiesta del content resolver, e per i broadcast receiver, attivati in risposta ad un messaggio broadcast, non è necessario esplicitarne la disattivazione. Un discorso diverso va fatto per le activity che possono essere disattivate invocando il proprio metodo **finish()** oppure possono essere disattivata da un'altra activity che in questo caso chiama il metodo **finishActivity()**. Un service, invece può essere invece bloccato chiamando il suo metodo **stopSelf()** oppure **Context.stopService()**.

1.3.1 Ciclo di vita dei componenti

Le componenti di un'applicazione hanno un ciclo di vita che ha inizio nel momento in cui Android le istanzia con lo scopo di rispondere ad un intent e termina quando le loro istanze vengono distrutte. Di seguito verrà descritto il ciclo di vita delle activity, service e broadcast receiver.

Activity

Durante il proprio ciclo di vita un'activity può trovarsi in uno degli stati mostrati nella figura 1.2.

- **Running**: l'activity si trova in cima allo stack e quindi è visibile sullo schermo per l'interazione con l'utente.

- **Paused**: in questo stato l'activity è ancora visibile all'utente che però non può interagire con essa in quando un'altra activity ha preso il suo posto in cima allo stack. L'activity rimane visibibe poichè la nuova activity può consistere in una schermata trasparente o perchè non copre per intero lo schermo. L'activity in pausa, pur mantenedo informazioni sul proprio stato può essere uccisa dal sistema nel caso in cui abbia bisogno di liberare spazio in memoria centrale.

- **Stopped**: l'activity entra in questo stato quando viene completamente coperta da un'altra activity. Mantiene informazioni sul suo stato e può essere uccisa dal sistema in caso di memoria insufficiente.

La transizione da uno satato ad un altro avviene attraverso la chiamata dei seguenti metodi:

onCreate(Bundle) Invocato quando l'activity viene attivata per la prima volta all'interno del task. Questo metodo prende in ingresso un oggetto di tipo Bundle contenente informazioni sullo stato precedente dell'activity se questo è stato memorizzato grazie alla chiamata del metodo **onSaveInstanceSatate()**.

onStart() Metodo chiamato prima che l'activity diventi visibile sullo schermo.

onResume() Chiamato subito prima che l'activity inizi ad interagire con l'utente. A questo punto l'attività si trova in cima allo stack della activity.

onPause() Chiamato quando l'activity si appresta ad essere oscurata da un'altra activity, viene utilizzato per memorizzare eventuali cambiamenti in modo persistente, fermare le animazioni e qualsiasi alta cosa faccia uso della CPU, etc. Viene seguita dal metodo **onResume()** se l'activity torna in cima allo stack o dal metodo **onStop()** se diventa completamente invisibile all'utente.

onStop() Metodo chiamato quando l'activity non è più visibile all'utente perchè viene distrutta o perche un'altra activity che ha preso il suo posto in cima allo stack la copre completamente. Viene seguito da metodo **onRestart()** se l'activity torna ad essere disponibile per l'interazione con l'utente, o dal metodo **onDestroy()** se non deve essere più utilizzata. Nel caso di limitate risorse di memoria il metodo **onStop()** può non essere invocato e il sistema semplicemente termina il processo che detiene l'activity.

onRestart() Quando viene invocato questo metodo l'activity passa dallo stato di stop allo stato running diventanto di nuovo visibile sullo schermo. A questa chiamata segue sempre **onStart()**.

onDestroy() Viene chiamato prima che l'activity sia distrutta. Ha il compito di rilasciare le risorse create in **onCreate()** compreso terminare i thread accessori ancora in esecuzione.

onSaveInstanceState(Bundle) Utilizzato per catturare lo stato di un'activity prima che questa venga distrutta. Al metodo viene passato un oggetto di tipo Bundle che contiene lo stato dinamico dell'activity. Quando

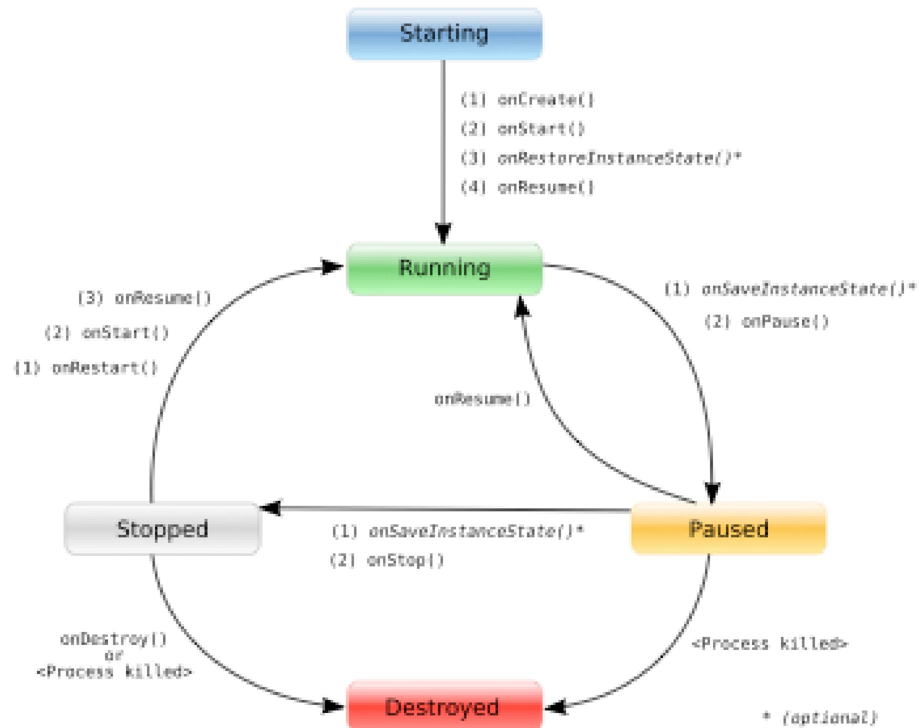


Figura 1.2: Ciclo di vita di un'activity

l'activity viene nuovamente fatta partire, l'oggetto Bundle viene passato sia al metodo **onCreate()** che al metodo **onRestoreIstanceState()** in modo tale da poter ricreare lo stato catturato.

onRestoreIstanceStates(Bundle) Invocato dopo il metodo **onStart()** serve a reinizializzare l'activity ad uno stato precedentemente salvato.

Service

Un servizio può essere utilizzato in due modi:

- può essere lanciato da un client attraverso la chiamata al metodo Context.startService(Intent) e fermato invocando il metodo Context.stopService(). Il servizio si può anche fermare in meniera autonoma attraverso la chiamata al metodo **Service.stopSelf()** oppure **Service.stopSelfResult()**. Per fermare un servizio occorre chiamare **stopService()** una volta sola mentre il metodo **startService()** può essere invocato più volte.

- se il client ha bisogno di interagire con le funzionalità che il servizio mette a disposizione attraverso la pubblicazione della sua interfaccia, deve stabilire una connessione con il servizio stesso utilizzando il metodo **Context.bindService()**. Questa connessione può essere chiusa invocando il metodo **Context.unbindService()**. Più client possono registrarsi allo stesso servizio il quale, se non attivo, può essere lanciato con il metodo **bindService()**.

I due metodi non sono completamente indipendenti, infatti è possibile collegarsi a un servizio attivato con il metodo startService(). Se più processi sono collegati allo stesso servizio la chiamata stopService() non avrà l'effetto di interrompere il servizio finchè non verranno chiuse tutte le connessioni verso di esso.

Come nel caso delle attività, il cambiamento da uno stato ad un altro all'interno del ciclo di vita di un servizio può essere monitorato attraverso la chiamata a dei metodi che in questo caso sono pubblici e non protetti.

L'intero ciclo di vita di un servizio si colloca all'intero delle chiamate ai metodi **onCreate()** e **onDestroy()**. Con la chiamata a onCreate() il servizio viene inizializzato mentre con la chiamata a onDestroy() vengono rilasciate le risorse di sistema in uso al servizio. All'interno di questi due metodi si colloca **onStart()**, che definisce il momento in cui il servizio diventa effettivamente attivo.

I metodi onCreate() e onDestroy() vengono utilizzati per tutti i servizi, sia per quelli iniziati da Context.startService() che per quelli iniziati Context.bindService() mentre in metodo onStart() viene chiamato solo per quei servizi iniziati da startService().

Come detto in precedenza un sevizio può mettere a disposizione ad uno o più client la possibilità di connettersi al servizio stesso per poterne manipolare i contenuti. Al tal proposito vengono resi disponibili ulteriori metodi per la gestione della connessione quali **IBinder onBind(Intent intent)**, **boolean onUnbind(Intent intent)** e **void onRebind(Intent intent)**.

Broadcast receiver

Quando un messaggio broadcast giunge al ricevitore, Android chiama il suo metodo onReceive() passandogli l'oggetto Intent contenente in messaggio. Il broadcast receive viene considerato attivo solo durante l'esecuzione del metodo onReceive(). Infatti quando questo metodo termina la sua esecuzione il ricevitore torna ad essere inattivo.

Un processo che detiene un broadcast receiver attivo non può essere ucciso. Dato che l'esecuzione della callback deve essere il più possibile veloce (altrimenti il main thread del processo nel quale risiede ne viene bloccato) e che

thread creati da questo componente non sono considerati attività protette da Android, elaborazioni di risposta ad eventi che richiedono tempo dovrebbero essere eseguite da un servizio accessorio.

1.3.2 Processi

Android cerca di mantenere vivo un processo il più a lungo possibile ma in caso di situazioni critiche, dovute soprattutto alla riduzione dello spazio in memoria centrale, deve decidere quale processo uccidere. Ad ogni processo, in base alle componenti in esecuzione al suo interno e al loro stato, viene assegnata una priorità che verrà poi utilizzata nel decidere se un processo può essere ucciso o meno.

Android prevede cinque livelli di priorità presentati qui di seguito in ordine di importanza:

1. **Processo in primo piano**: un processo appartiene a questa categoria se:

   (a) al suo interno è in esecuzione un'attività con la quale l'utente sta interagendo (ovvero è stato chiamato il metodo onResume())

   (b) ospita un servizio legato all'attività con la quale l'utente sta interagendo

   (c) ha al suo interno un oggetto di tipo Service che sta eseguendo una chiamata a un metodo appartenente al suo ciclo di vita (onCreate(), onStart(), or onDestroy())

   (d) è presente un broadcast receiver che sta eseguendo il suo metodo onReceive()

   Solo pochi processi posso appartenere, in un determinato istante, a questa categoria. Possono essere uccisi solo se impossibilitati a continurare la loro esecuzione nonostante le azioni intraprese dal sistema per evitare che ciò accada.

2. **Processo visibile**: sono quei processi che non hanno componenti in primo piano, ma che possono avere effetti su ciò che l'utente sta guardando sullo schermo. Un processo appartiene a questa categoria se:

   (a) possiede un'attività che pur non essendo in primo piano è ancora visibile sullo schermo.

   (b) possiede un servizio collegato ad un'attività visibile

Un processo appartenente a questa categoria viene considerato estremamente importante e può essere ucciso solo per salvaguardare la normale esecuzione dei processi in primo piano.

3. **Processo di servizio**: sono processi che hanno in esecuzione un servizio avviato con la chiamata al metodo startService(). Anche se i servizi non sono direttamente legati a ciò che è visibile sullo schermo, generalmente fanno cose molto importanti per l'utente come la riproduzione in background di un mp3. Vengono uccisi solo quando la loro esecuzione toglie risorse precludendo così l'esecuzione dei processi appartenenti alle prime due categorie.

4. **Processo in secondo piano**: sono processi che non possiedono attività attualmente visibili sullo schermo e che quindi non hanno un impatto diretto con l'attività svolta dall'utente. Finchè il processo in primo piano è in esecuzione, questi processi vengono inseriti in una lista LRU (least recently used) e terminati a partire dal più vecchio per essere ricaricati nel caso l'utente ne richieda la visualizzazione. Se le callback delle attività sono implementate correttamente questo meccanismo avviene in modo trasparente all'utente.

5. **Processo vuoto**: sono processi che non possiedono componenti attive. Questi processi vengono mantenuti in vita solo per ottimizzarne la loro esecuzione nel momento in cui un componente diventa attivo e necessita di essere eseguito.

1.3.3 Android User Interface

Un'interfaccia utente può essere costruita in Android in due modi: con l'utilizzo di codice XML o direttamente in java. Definire la GUI in XML risulta essere il modo preferibile perchè è così possibile avere una netta separazione tra l'interfaccia grafica e il core dell'applicazione stessa. Inoltre il file XML può essere convertito in un binario compresso dai tools presenti nell'SDK in modo da limitarne lo spazio occupato e permette di definire viste in modo molto intuitivo.

L'oggetto principale nella costruzione di un'interfaccia è definito dalla classe *android.app.Activity*. Un'attività, per interagire con l'utente, deve poter essere visualizzata sullo schermo, cosa che non è in grado di fare senza l'utilizzo dei view e viewgroup, che rappresentano l'unità base nella descrizione di un'interfaccia utente.

View Una vista è un oggetto che estende la classe *android.view.View*. É una struttura dati con determinate proprietà che racchiudono la definizione

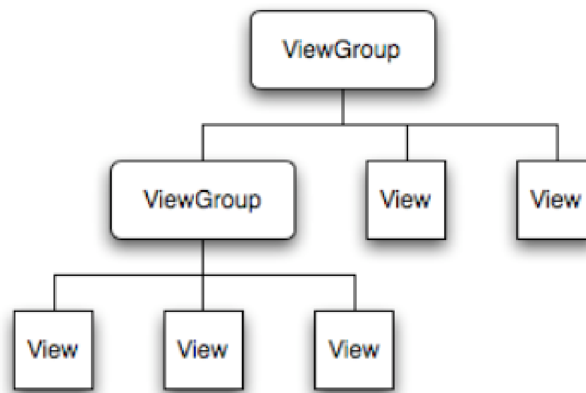


Figura 1.3: Struttura di una UI

di layout e dei contenuti per una specifica area di forma rettangolare dello schermo.

GroupView Un viewgroup è un oggetto dell classe *android.view.Viewgroup*. Il suo compito è quello di contenere e gestire un sotto-insieme di viste o altri gruppi di viste permettendo allo sviluppatore di aggiungere ulteriori strutture alla propria interfaccia la quale risulterà però essere indirizzabile come un'unica entità.

Sulla piattaforma Andorid l'interfaccia utente di un'attività può essere definita attraverso una struttura ad albero dove viste e gruppi di viste rappresentano i nodi come mostrato in figura 1.3

Per rendere effettiva la visualizzazione della struttura sullo schermo l'oggetto Activity deve chiamare il suo metodo *setContentView* e passargli un riferimento della radice dell'albero. Una volta ottenuto il riferimento al nodo radice, il sistema lavora su di esso per effettuare alcuni controlli e infine per stampare l'intera struttura a schermo. Ogni nodo padre è responsabile per la stampa a video di ogni nodo figlio.

1.3.4 Il file manifest

Ogni applicazione sviluppata per la piattaforma Android deve contenere all'interno della root directory il file AndroidManifest.xml. Questo file contiene informazioni riguardanti l'applicazione necessarie alla sua corretta esecuzione. L'Android Manifest ha il compito di:

- definire il nome del pacchetto java dell'applicazione

- descrivere le componenti dell'applicazione. Definire il nome delle classi in cui queste componenti vengono implementate e pubblicare le loro caratteristiche
- determinare quali processi ospiteranno le componenti dell'applicazione
- definire i permessi che un'applicazione deve avere per poter utilizzare alcune API Java protette e per poter interagire con altre applicazioni
- definire un livello minimo di API Java necessarie all'applicazione
- definire una lista di librerie aggiuntive (oltre al core library) necessarie all'applicazione.

Il diagramma seguente mostra la struttura generale del file manifest e ogni elemento che può contenere.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest>
    <uses-permission/>
    <permission/>
    <permission-tree/>
    <permission-group/>
    <instrumentation/>
    <uses-sdk/>
    <uses-configuration/>
    <uses-feature/>
    <supports-screens/>

    <application>
        <activity>
            <intent-filter>
                <action/>
                <category/>
                <data/>
            </intent-filter>
            <meta-data/>
        </activity>

        <activity-alias>
            <intent-filter> . . . </intent-filter>
            <meta-data/>
        </activity-alias>

        <service>
```

```
            <intent-filter> . . . </intent-filter>
            <meta-data/>
        </service>

        <receiver>
            <intent-filter> . . . </intent-filter>
            <meta-data/>
        </receiver>

        <provider>
            <grant-uri-permission/>
            <meta-data/>
        </provider>

        <uses-library/>

    </application>

</manifest>
```

<manifest> Elemento radice del file manifest deve contenere un elemento <application> e specificare gli attributi *xlmns:android* e *package*

<uses-permission> Richiede al sistema un permesso che deve essere concesso all'applicazione per garantirne il suo corretto funzionamento (per esempio il permesso di inviare un sms o utilizzare l'elenco dei contatti)

<permission> Definisce un permesso di sicurezza che può essere utilizzato per limitare l'accesso a specifiche componenti e funzioni di questa o altre applicazioni

<instrumentation> Definisce una classe Instrumentation che permette di monitorare l'interazione dall'applicazione con il sistema. L'oggetto Instrumentation viene instanziato prima di ogni componente dell'applicazione.

<application> Contiene la dichiarazione delle componenti dell'applicazione e i loro attributi.

<activity> Definisce un'attività che implementa una parte o tutta l'interfaccia grafica dell'applicazione. Ogni attività, per essere vista dal sistema e quindi per essere utilizzata, deve essere dichiarata all'interno del manifest

<intent-filter> Specifica i tipi di intent supportati dalle componenti dell'applicazione. Definisce le capacità del componente al quale fa riferimento come cosa un'activity o un service possono fare e quali tipi di messaggi un receiver supporta. Fa passare gli intent del tipo dichiarato e blocca tutti gli altri.

<receiver> Dichiara un broadcast receiver come uno dei componenti dell'applicazione. Abilita l'applicazione a ricevere intent inviati dal sistema o da altre applicazioni anche quando le altre componenti non sono attualmente in esecuzione.

<service> Dichiara un servizio che può essere utilizzato da altre applicazioni.

<provider> Dichiara un content provider per la gestione e l'accesso alle strutture dati utilizzate dall'applicazione per la memorizzazione dei propri dati.

1.4 Android SDK

Per agevolare lo sviluppatore nel compito di realizzare applicazioni per la piattaforma Android, la Open Handset Alliance ha rilasciato, nel novembre 2007, il software development kit (SDK), ovvero un'insieme di utili strumenti utilizzabili per il debugging, il packaging, l'installazione delle applicazioni.

Lo strumento più importante messo a dispozione dal SDK è senza dubbio l'emulatore, un software che consente di emulare un dispositivo reale sul quale è possibile debaggare, testare e progettare la propria applicazione. Scaricabile all'indirizzo *http://developer.android.com/sdk/index.html* è utilizzabile per le tre diverse piattaforme Windows, Apple's Mac OS X e Linux.

All'interno della cartella */tools* è possibile trovare una serie di strumenti utili per lo sviluppo di applicazioni. Questi strumenti vengono messi a disposizione anche sull'IDE Eclipse previa l'installazione del plugin ADT che permette, tra le altre cose, di creare un progetto Android, di compilarlo ed eseguirlo sull'emulatore che verrà avviato in maniera automatica, di debuggare l'applicazione con l'ausilio di una comoda interfaccia grafica. Di seguito verranno descritti alcuni dei tool più utilizzati per lo sviluppo dell'applicazione.

- **aapt - Android Asset Packaging Tool**: permette allo sviluppatore di visualizzare, creare e aggiornare archivi compatibili allo standard .zip (zip, jar, apk);
- **adb - Android Debug Bridge**: strumento molto potente e utile che consente di stabilire un canale di comunicazione verso un dispositivo sia

esso reale o virtuale. L'adb, con l'aggiunta di ulteriori opzioni, permette di aprire una shell, di installare o disinstallare applicazioni, eseguire il debug, ect.

- **aidl - Android Interface Definition Language**: viene utilizzato per generare codice che permette a due processi di comunicare attraverso l'uso dell'IPC.

- **ddms - Dalvik Debug Monitor Service**: rende disponibile un servizio di port-forwarding, permette di catturare la schermata del dispositivo, fornisce informazioni sui thread in escuzione, sull'utilizzazione del processore e della memoria, dà la possibilità attraverso il comando logcat di stampare a video i log generati dal sistema, etc..

- **dx**: a partire dai file .class genera del bytecode che verrà in seguito convertito nel formto .dex che puo essere eseguito dalla Dalvik VM.

- **traceview**: serve per visualizzare i tracefile che possono essere creati sul dispositivo.

- **emulator**: permette di avviare l'avd passatogli come parametro.

Inoltre l'SDK contiene:

- Il Core Library android.jar, ovvero quelle librerie di supporto indispensabili per la compilazione dell'applicazione per la piattaforma Android.

- Documentazione, Esempi e Tutorial: Tutte le informazioni necessarie per iniziare a sviluppare applicazioni per Android.

- Il kernel Qemu e il root file system indispensabili per avviare android in ambiente virtuale.

1.4.1 Emulatore

L'emulatore riproduce tutte le funzionalità hardware e software di un tipico dispositivo mobile a parte la ricezione o l'esecuzione di telefonate.

La creazione di una macchina virtuale può avvenire in semplici passaggi grazie all'utilizzo dell'Android Virtual Device manager, un software che attraverso un'interfaccia grafica permette anche di gestire l'aggiornamento delle versioni di Android disponibili per l'emulatore e della relativa documentazione.

Come detto in precedenza l'emulatore sfrutta il motore Qemu per la virtualizzazione del dispositivo. Questo ci permette di sfruttare le caratteristiche implicite del software Qemu per permettere a due istanze dell'emulatore o

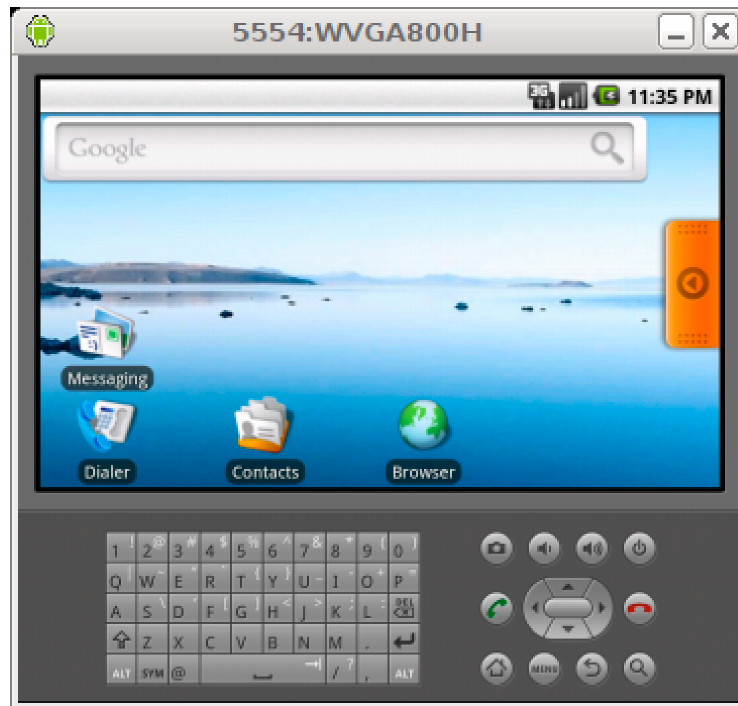


Figura 1.4: L'emulatore

a un'istanza dell'emulatore e una Qemu di poter comunicare sfruttando la configurazione della rete virtualizzata da ciascun dispositivo.

Ciascuna istanza Qemu viene eseguita dietro un router/firewall virtuale che impedisce all'interfaccia di rete della macchina host di poter comunicare con essa. Viene comunque permesso al dispositivo virtuale di interagire con la rete per consentire, ad esempio, la normale navigazione sul web attraverso un browser. Lo spazio degli indirizzi gestiti dal router è 10.0.2/24, quindi ogni dispositivo virtuale generato all'avvio dell'istanza Qemu avrà un indirizzo ip nella forma 10.0.2.xx. Questi indirizzi ip vengono allacati in maniera statica, per ogni istanza Qemu, come mostrato nella tabella 1.1

Come accennato in precedenza le comunicazioni tra la macchina host e l'emulatore sono bloccate dal firewall. É possibile però aprire le porte del firewall utilizzando l'opzione redirect, permettendo la comunicazione tra la macchina host e l'emulatore o tra due istanze Qemu. Dopo aver avviato l'emulatore è possibile interagire con esso attraveso il protocollo telnet digitando su una shell il seguente comando:

```
$ telnet localhost 5554
```

dove 5554 specifica il numero della console associata all'emulatore. Una volta connessi, per aggiungere una redirezione utilizzare il comando

Indirizzo IP	Descrizione
10.0.2.1	Indirizzo IP der router/gateway
10.0.2.2	Indirizzo IP dell'interfaccia loopback della macchina host (127.0.0.1)
10.0.2.3	Indirizzo IP del server DNS primario
10.0.2.4 10.0.2.5 10.0.2.6	Indirizzi IP del secondo, terzo e quarto server DNS
10.0.2.15	Indirizzo IP dell'interfaccia di rete del dispositivo virtuale
127.0.0.1	Indirizzo di loopback del dispositivo virtuale

Tabella 1.1: Assegnazione statica degi indirizzi in un'istanza Qemu

```
redir add <protocol>:<host-port>:<guest-port>
```

dove <protocol> indica il protocollo UDP o TCP mentre host-port e guest-port sono le porte della macchina host e di quella guest che andremo a collegare. Ogni connessione TCP o UDP con porta sorgente host-port verrà redirezionata verso la porta guest-port.

Supponendo di avere un server web attivo sull'emulatore possiamo connetterci ad esso impostando la seguente redirezione

```
redir add tpc:8080:80
```

Per attivare la connessione basta digitale sulla barra degli indirizzi di un browser la seguente stringa:

```
http://127.0.0.1:8080
```

Nel caso in cui il browser risieda su un altro emulatore occorre digitare sulla barra degli indirizzi

```
http://10.0.2.2:8080
```

Capitolo 2

Installazione di Android sulla I.MX51 BBG board

2.1 I.MX51 BBG board

L'application processor I.MX51 rappresenta l'ultimo arrivato di una crescente famiglia di prodotti focalizzati sul multimedia che offrono alte prestazioni in contesti ottimizzati per un basso consumo energetico. La principale caratteristica di questa famiglia di application processor è l'integrazione di un core costituito da un ARM Cortex A8™ che opera ad una frequenza massima pari a 800 MHz. Inoltre supporta una memoria RAM DDR2 con frequenza massima di 200 MHz. Il dispositivo è stato costruito per utilizzi quali:

- Netbook (web tablet)
- Nettops (dispositivi internet desktop)
- Dispositivi internet mobili (MID)
- Lettori multimediali portatili (PMP)
- Navigatori portatili (PND)
- Console di gioco
- Navigatori satellitari e intrattenimento.

2.1.1 Caratteristiche

Tra le caratteristiche più importanti ricordiamo:

Smart Speed Technology Il cuore dei processori I.MX51 è costituito da un livello di gestione energetico che abbraccia tutte le funzionalità del dispositivo e mette a disposizione una ricca suite di caratteristiche multimediali e periferiche che richiedono un minimo consumo energetico sia che essi siano in modalità attiva che in modalità low-power. Permette inoltre allo sviluppatore di progettare prodotti ricchi di caratteristiche che richiedono un dispendio energetico di gran lunga inferiore rispetto alle aspettative tipiche del settore.

Applications Processor I processori I.MX51 accrescono le capacità delle applicazioni portabili provvedendo ai bisogni sempre crescenti di incrementare il numero di istruzioni processate per secondo (MIPS) necessarie ad un sistema operativo o gioco. Il Dynamic Voltage and Frequency Scaling (DVFS) di Freescale permette al dispositivo di funzionare con un basso voltaggio garantendo un MIPS sufficiente per attività come la decodifica audio con una conseguente riduzione significativa dell'energia utilizzata.

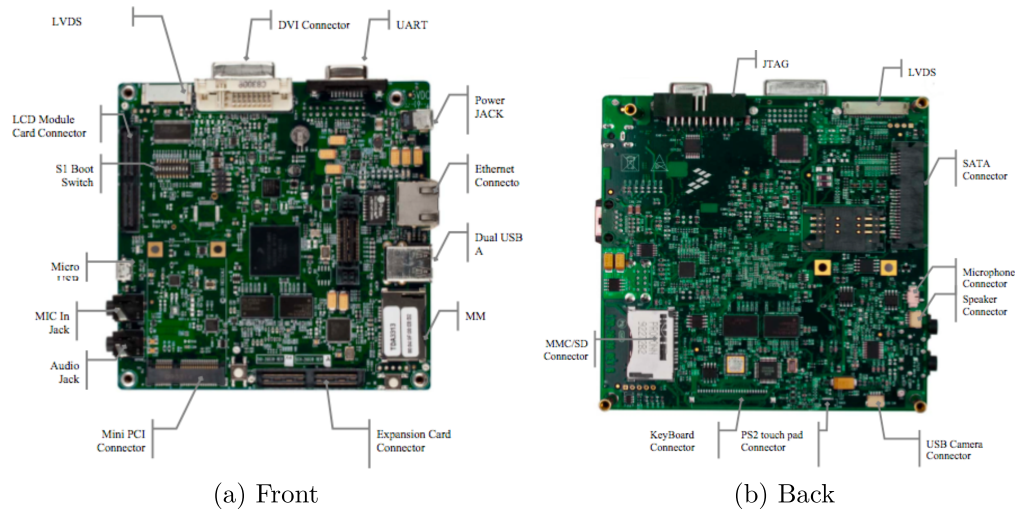
Multimedia Powerhouse Le prestazioni multimediali sono ulteriormente rafforzate dalla presenza di una cache multilivello e da un hardware che supporta diversi codec video standard. É anche presente un'autonoma unità per l'elaborazione delle immagini, un codificatore video HD a 720p, un'unità Neon e un controler DMA (SDMA).

Powerful Graphics Acceleration La grafica risulta essere fondamentale per i giochi, la navigazione web e per altre applicazioni. I processori I.MX51 forniscono due indipendenti e integrate unità per l'elaborazione video: l'acceleratore 3D OpenGL ES 2.0 e l'acceleratore grafico 2D OpenVG1,1.

Increased Security Considerando il continuo aumento del bisogno di sicurezza per i dispositivi mobili, i processori I.MX51 offrono funzioni di sicurezza integrate a livello hardware che consentono una sicura e-commerce, una gestione dei diritti digitali (DRM), crittografia dei dati e il download di software sicuro.

2.1.2 Hardware e periferiche

Di seguito verranno descritte le principali caratteristiche hardware presenti nella board I.MX51:



(a) Front (b) Back

Figura 2.1: I.MX51

RAM La board possiede 512 MB SDRAM DDRII suddivisi in 4 moduli da 128 MB ciascuno dei quali è costituito da 8 banchi di 16 M parole da 16 bit. La memoria lavora ad una frequenza di 200 MHz con bus da 32 bit.

SPI NOR Flash Memoria non volatile integrata nella board di dimensione pari a 4MB. Può essere utilizzata in fase di boot del sistema configurando in maniera opportuna gli switch SW1 (figura 2.1 (a)). La sua dimensione gli permette di contenere sia il bootloader che l'immagine compressa del kernel linux.

USB OTG e Host La i.MX51 monta un chip USB PHY per lo standard USB2.0 ad alta velocità di trasferimento dati che può supportare l'USB OTG. Inoltre la i.MX51 ha un'interfaccia USB parallela connessa all'USB transceiver per supportare la funzione USB host. Grazie all'utilizzo del chip USB Hub connesso all'interfaccia USB Host, è possibile espandere tale porta per essere utilizzata da più dispositivi come un mouse, tastiera, hard disk, etc.

CMOS Sensor L'image processing unit (IPUv3EX) nell'i.mx51 consente la connettività a display e image sensors, ovvero quei dispositivi che convertono il segnale ottico in segnale elettrico. Supporta due porte per il collegamento di un display e due porte per il collegamento di una foto camera. Nella board non è presente di default una foto camera ma questa può essere aggiunta collegando all'ingresso expansion connector (figura 2.1 (a)) la scheda di espansione.

Touch screen controller La board supporta un 4 wire touch screen controller. Di default la i.mx51 non monta uno schermo LCD ma è possibile utilizzare l'interfaccia LCD Module Card Connector, alla quale è connessa il 4 wire touch screen controller, per montare uno schermo che dispone della funzione touch.

Uscite DVI e VGA La board dispone di un'uscita DVI e di un'uscita VGA che permette il collegamento con un dispositivo video esterno.

LCD Connectors La i.MX51 supporta un pannello LVDS (Low Voltage Differential Signaling) connesso all'interfaccia display1. Supporta VGA, SVGA, XGA, SXGA (dual pixel), SXGA+ (dual pixel), and UXGA (dual pixel).

Interfacce MMC/SD Sulla board sono presenti due connettori (P1 e P2) per il trasferimento dati da e verso schede MMC/SD. Il connettore P1 (figura2.1 (b)) supporta schede MMC (Multi Media Card) con interfaccia seriale a 8 bit mentre il connettore P2 (figura 2.1 (a)) supporta schede SD (Secure Digital) con interfaccia seriale da 4 bit. L'interfaccia SD supporta sia lo standard SD memory card che lo standard SDIO card (es. schede Wi-Fi SDIO). I due slot P1 e P2 vengono utilizzati per scopi diversi: P1 può essere selezionata, settando in modo appropriato gli switch SW1, per il boot del sistema nel caso non si voglia utilizzare la SPI-NOR, mentre P2 può essere utilizzata per l'archiviazione di dati o per schede supportanti lo standard SDIO.

Fast Ethernet Connect (FEC) Interfaccia progettata per supportare gli standard 10 e 100 Mbps Ethernet /IEEE 802.3.

Application processor L'application processor i.MX51 è basato sulla piattaforma ARM Cortex A8™ che presenta le seguenti caratteristiche:

- Processore ARM Cortex A8™
- 32 Kbyte L1 Instruction Cache
- 32 Kbyte L1 Data Cache
- 256 Kbyte L2 cache
- Coprocessore Neon
  - SIMD Media Processing Architecture
  - NEON register file with 32 $\times$64-bit general-purpose registers

  - NEON Integer execute pipeline (ALU, Shift, MAC)
  - NEON dual, single-precision floating point execute pipeline (FADD, FMUL)
  - NEON load/store and permute pipeline
  - Non-pipelined Vector Floating Point (VFP) co-processor (VFP)

- Frequenza massima del clock pari a 800 MHz

La board I.MX51 è stata utilizzata per l'esecuzione del sistema operativo Android nella versione 2.1 (Eclair) e per lo sviluppo e il testing dell'applicativo. Nel prosieguo del capitolo verrà illustrata passo dopo passo la procedura che ha portato il sistema Android ad essere installato ed eseguito sulla board I.MX51.

2.2 Preparazione dell'ambiente di sviluppo

Il PC utilizzato per la fase di sviluppo monta un processore Pentium 4 con 1 Gigabyte di RAM con sistema operativo Ubuntu 9.10. Per prima cosa è utile definire quella che sarà la directory che conterrà il nostro progetto. Questa deve essere situata in una partizione abbastanza capiente da poter contenere oltre ai file sorgenti di Android e il codice di supporto fornito da Freescale anche tutti quei file che verranno generati dal processo di compilazione. Uno spazio di memoria non inferiore ai 10 Gigabyte dovrebbe bastare.

La directory `opt` sarà il nostro root project path

```
$ mkdir /home/<username>/opt
```

Scarichiamo il codice di supporto fornito da Freescale, salviamolo all'interno della cartella `opt` e scompattiamolo.

```
$ tar xzvf imx-android-r8.tar.gz
$ cd imx-android-r8/code
$ tar xzvf R8.tar.gz
$ cd imx-android-r8/tool
$ tar xzvf gcc-4.1.2-glibc-2.5-nptl-3.tar -C /opt
```

La directory `code/` contiene le patch che verranno utilizzate per adattare il codice a nostra disposizione alla macchina target e prepareranno il sistema per essere compilato per quest'ultima.

La directory `gcc-4.1.2-glibc-2.5-nptl-3/` contiene un toolchain utilizzato per la cross-compilazione, ovvero quella procedura che porta un software ad essere compilato per una macchina con architettura diversa rispetto a quella in cui è stata effettuaua la compilazione. Questo tipo di procedura risulta essere utile per i sistemi embedded dove le risorse sono generalmente molto limitate e quindi non sufficienti a supportare la normale compilazione.

2.2.1 Scaricare i sorgenti

Scarichiamo ora i sorgenti di Android. All'interno della directory `/home/<username>/opt` creiamo la cartella myandroid che conterrà, oltre ai sorgenti di Android, i sorgenti del boot loader e del kernel UNIX.

```
$ mkdir myandroid
$ cd myandroid
```

Scarichiamo lo script repo che ci permetterà di automatizzare le operazioni di download e inizailiziamolo

```
$ curl http://android.git.kernel.org/repo > repo
$ chmod a+x ~/bin/repo
$ ./repo init -u git://android.git.kernel.org/platform/
    manifest.git -b eclair
$ cp ../imx-android-r8/code/r8/default.xml .repo/manifests/
    default.xml
```

sincroniziamo il repository remoto con quello locale con il comando

```
$ ./repo sync
```

a questo punto il processo di download verrà avviato.

Una volta teminato il download dei sorgenti di Android scarichiamo il kernel linux

```
$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/
    stable/linux-2.6.31.y.git kernel_imx
```

e il boot loader

```
$ cd bootable/bootloader
$ git clone git://git.denx.de/u-boot.git uboot-imx
```

2.2.2 Installare e configurare il server NFS

Il Network File System è un protocollo che consente a dei calcolatori di utilizzare la rete per accedere ai dischi remoti come se fossero dischi locali.

Per installere un server NFS digitare su una shell

```
$ sudo apt-get install nfs-kernel-server
```

Per configurarlo è necessario editare il file `exports` contenuto nella directory `/etc/`

```
$ sudo nano /etc/exports
```

All'interno del file `exports` digitiamo la seguente stringa

```
/nsfroot 192.168.0.2/255.255.255.0(rw,no_all_squash,
    no_root_squash)
```

dove:

- `/nsfroot` è la directory che conterrà il nostro file system;
- `192.168.0.2/255.255.255.0` sono rispettivamente l'indirizzo ip e la sottomaschera di rete dell'unico host al quale è permesso da remoto di accedere a `/nfsroot`;
- `rw` assegna in lettura e scrittura i permessi di accesso;
- `no_root_squash` consente all'utente con privilegi di root sulla macchina client di mantenere gli stessi privilegi di root sulla macchina server. Se non abilitata l'utente root verrà tracciato come utente nobody sulla macchina server;
- `no_all_squash` simile all'opzione no_root_squash, viene utilizzata per gli utenti che non hanno privilegi di root.

Per attivare il demone che gestisce il server eseguire

```
$ sudo /etc/init.d/nfs-kernel-server start
```

Per verificare l'effettiva esistenza del server digitare

```
$ netstat -al | grep nfs
tcp      0      0 *:nfs  *:*      LISTEN
udp      0      0 *:nfs  *:*      LISTEN
```

2.2.3 Configurare il server TFTP

Il Trivial File Transfer Protocol permette il trasferimento di file tra una macchina server e una client senza richiedere credenziali per l'autenticazione. Risulta utile per caricare in una macchina target l'immagine del kernel.

Per installare un server tftp eseguire da shell

```
$ sudo apt-get install xinetd tftpd tftp
```

Creare il file `/etc/xinetd.d/tftp` se non presente ed editarlo in questo modo:

```
service tftp {
                protocol        = udp
                port            = 69
                socket_type     = dgram
```

```
        wait            = yes
        user            = nobody
        server          = /usr/sbin/in.tftpd
        server_args     = /tftpboot
        disable         = no
}
```

Creare la directory di accesso ad un client tftp

```
$ sudo mkdir /tftpboot
$ sudo chmod -R 777 /tftpboot
$ sudo chown -R nobody /tftpboot
```

Infine avviare il daemon che gestisce il server

```
$ sudo /etc/init.d/xinetd start
```

Come nel caso del server nfs è utile verificarne la sua effettiva presenza con il comando netstat

```
$ netstat -al | grep tftp
udp      0       0 *:tftp        *:*
```

2.2.4 Configurare il minicom

Il minicom è un programma che permette alla macchina host di comunicare con la macchina target attraverso la porta seriale. Per installarlo digitare su una shell

```
$ sudo apt-get install minicom
```

Una volta installato occorre configurarlo secondo i parametri forniti dal produttore della board mostrati nella tabella seguente:

Baud Rate	115200
Data Bits	8
Parity	None
Stop Bits	1
Flow Control	None

2.3 Compilazione dei sorgenti

Una volta scaricati i sorgenti è possibile passare alla fase di compilazione. Questa non è strettamente necessaria in quanto Freescale mette a disposizione per gli sviluppatori delle immagini precompilate pronte all'uso. La compilazione

risulta utile nel caso in cui lo sviluppatore decida di personalizzare la propria piattaforma.

Il primo passo è quello di applicare le patch che si trovano nella directory `/home/<username>/opt/imx-android-r8/code/r8` al codice sorgente. Per far questo occore preparare l'ambiente eseguendo lo script *and_patch.sh*.

```
$ cd /home/<username>/opt
$ . imx-android-r8/code/R8/and_patch.sh
```

L'esecuzione di questo script rende disponibile allo sviluppatore la funzione "c_patch" visualizzabile eseguendo

```
$ help
```

La funzione "c_patch" verrà utilizzata per l'applicazione delle patch nel seguente modo

```
$ c_patch imx-android-r8/code/R8 imx_r8
```

dove "imx_r8" è il branch che verrà automaticamente creato per contenere tutte le patch. É possibile scegliere qualsiasi nome al posto di quello utilizzato per definire il branch.

Se l'applicazione delle pacth è andata a buon fine sullo schermo verrà stampato il seguente avviso

```
**********************************************************
Success: Now you can build android code for FSL i.MX platform
**********************************************************
```

2.3.1 Compilare l'U-boot

Per prima cosa occorre esportare le variabili "ARCH" e "CROSS_COMPILE" in modo da indicare al "make" il tipo di architettura per la quale si desidera compilare i sorgenti e la directory contenente i cross-compilatori.

```
$ export ARCH=arm
$ export CROSS_COMPILE=/home/opt/gcc-4.1.2-glibc-2.5-nptl-3/
  arm-none-linux-gnueabi/bin/arm-none-linux-gnueabi-
```

Per compilare l'U-boot per la board i.MX51 BBG eseguire i comandi

```
$ cd /home/<username>/opt/myandroid/bootable/bootloader/
  uboot-imx
$ make mx51_bbg_android_config
$ make
```

Al termine della compilazione, se non ci saranno errori, verrà generato il file "u-boot.bin". Questo file ha come header un padding di 1024 byte, per cui la

prima istruzione eseguibile appartenente all'U-boot avrà un offset di 1 Kbyte. É possibile comunque, a partire da questo file, ottenere un'immagine che non presenta un padding come header utilizzando il comando dd come segue

```
$ dd if=./u-boot.bin of=./u-boot-no-padding.bin bs=1024 skip
    =1
```

2.3.2 Compilare il kernel

Prima di procedere alla compilazione del kernel assicurarsi che le variabili d'abiente siano state esportate in modo corretto come nel caso della compilazione dell'U-boot.

I comandi

```
$ echo $ARCH
$ echo $CROSS_COMPILE
```

Dovrebbero restituire rispettivamente i valori "arm" e "/home/<username>/opt/gcc-4.1.2-glibc-2.5-nptl-3/arm-none-linux-gnueabi/bin/arm-none-linux-gnueabi-".

Spostiamoci all'interno della directory contenente i sorgenti del kernel

```
$ cd /home/<username>/opt/myandroid/kernel_imx
```

e digitiamo il seguente comando che serve a generare il file ".config" che raccoglie le informazioni riguardante la configurazione del sistema secondo le informazioni riportate nella directory `arch/arm/configs`.

```
$ make imx51_android_defconfig
```

Infine compiliamo il kernel

```
$ make uImage
```

In caso di successo, all'interno della directory `kernel_imx/arch/arm/boot/` verrà generata l'immagine "uImage".

2.3.3 Compilare Android

Per prima cosa è necessario esportare la variabile d'ambiente PATH in modo da includere la posizione assoluta della directory contenente i compilatori

```
$ export PATH=home/<username>/opt/gcc-4.1.2-glibc-2.5-nptl
    -3/arm-none-linux-gnueabi/bin:$PATH
```

É ora possibile generare le immagini di Android eseguendo i seguenti comandi:

```
$ cd /home/<username>/opt/myandroid
$ make PRODUCT=imx51_BBG-eng | tee build_imx51_BBG_android.
    log
```

dove imx51_BBG è il nome del prodotto per il quale si desidera avere le immagini e build_imx51_BBG_android.log è il file nel quale verranno salvati i log generati durante la fase di compilazione. Il nome imx_BBG è riconducibile alla definizione di prodotto situata all'interno della directory `myandroid/out/target/product/imx51_BBG`. *eng* invece è un modificatore di prodotto che viene utilizzato per impostare le seguenti variabili di default:

- ro.secure=0: le applicazioni non certificate possono eseguire nello stack.
- ro.debuggable=1: viene generato il server adb che viene attivato di default.

Gli altri modificatori di prodotto che possono essere impostati sono definiti dai tag:

user permette di generare la versione "finale" dello stack. Vengono installati i soli moduli taggati user e quelli non taggati ma inclusi nelle specifiche di prodotto scelto. Vengono impostate le seguenti variabili di default:

- ro.secure=1: le applicazioni non certificate non possono eseguire nello stack.
- ro.debuggable=0 : il server adb viene generato ma viene disabilitato di default.

userdebug permette di generare una versione simile a user contenente inoltre i moduli taggati debug. Vengono impostate le seguenti variabili di default:

- ro.secure=1 : le applicazioni non certificate non possono eseguire nello stack.
- ro.debuggable=1 : è generato il server adb e viene attivato di default.

All'interno della cartella `myandroid/out/target/product/imx51_BBG` verranno generate le immagini di Android, ovvero:

ramdisk.img: Contiene quei file strettamente necessari per avviare il processo di inizializzazione dell'ambiente Android:

- Lo scheletro del root-filesystem Android.
- Il file binario /init di inizializzazione dello stack.
- Lo script che determina le operazioni eseguite dal processo init : /init.rc.

system.img: Contiene tutti i file binari e di configurazione dello stack Android.

userdata.img: Contiene una versione iniziale della partizione dedicata ai dati utente.

recovery.img: Contiene tutto il necessario per avviare il sistema in modalità recovery.

2.4 Linux arm boot process

In questa sezione verrà descritto per sommi capi il processo che porta all'avvio del kernel Linux su architettura arm. Questo processo coinvolge inizialmente il boot loader, nel nostro caso l'U-boot che deve:

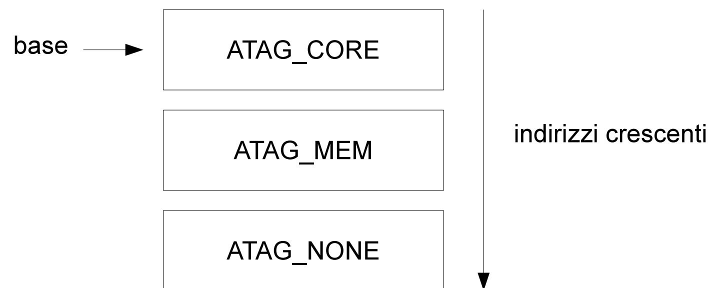
Configurare e inizializzare la RAM Per poter essere utilizzata al meglio dal kernel la RAM deve essere inizializzata e configurata. Infatti il kernel non dispone di alcuna informazione relativa alla RAM di sistema, informazioni che gli possono essere fornite solamente dal boot loader, essendo quest'ultimo il primo programma che viene eseguito in fase di start-up. Questa operazione, dipendente dall'architettura utilizzata, può essere compiuta utilizzando algoritmi interni che localizzano e valutano la dimensione dell'intera RAM, o possono essere utilizzate informazioni fornite dalla RAM stessa. Una volta determinato il layout della memoria fisica del sistema, viene utilizzato il parametro ATAG_MEM per passare quest'informazione al kernel.

Inizializzazione di una porta seriale Inizializza e abilita una porta seriale per permettere al kernel serial driver di localizzare in modo automatico quale porta seriale deve essere utilizzata come console (generalmente utilizzata per debaggare il sistema o per comunicare con esso). Questa operazione può essere gestita in maniera diretta dallo sviluppatore che può decidere quale porta seriale utilizzare indicando al boot loader di passare al kernel questa informazione mediante il parametro “console=”.

Individuazione del tipo di macchina Una volta individuato il tipo di macchina deve fornire al kernel il valore MACH_TYPE_xxx (mx51_BABBAGE arch/arm/tools/mach-types #2125).

Configurazione della tagged list di sistema Una tagged list non è altro che una lista di parametri e informazioni utili riguardanti la memoria di sistema compilata dal boot loader per comunicare con il kernel. Affinchè possa essere considerata valida una tagged list deve rispettare i seguenti vincoli:

1. la lista deve essere salvata in RAM e posta in una regione di memoria che non può essere sovrascritta nè dal kernel in fase di decompressione nè dalla ram disk
2. l'indirizzo fisico della tagged list deve essere posto nel registro R2
3. la lista deve iniziare con ATAG_CORE e deve terminare con ATAG_NONE
4. deve contenere almeno un ATAG_MEM. Sotto le condizioni 3 e 4 la tagged list di lungezza minima che può essere considerata valida è delle forma



Ciascuna tag nella lista è composta da un header contenente due valori espressi da una parola di 32 bit. Il primo valore esprime la dimensione della tag mentre il secondo identifica la tag stessa.

```
struct atag_header {
        u32 size; /* legth of tag in words including
                                this header */
        u32 tag;  /* tag value */
};
```

Ciascun header è a sua volta seguito da una serie di dati associati alla tag. Fanno eccezione ATAG_NONE che non contiene dati e ATAG_CORE che può o non può contenere dei dati.

Nella tabella 2.1 sono riportati i possibili tag che possono essere utilizzati e le rispettive descrizioni.

Tag	Descrizione
ATAG_NONE	Delimita la fine di una lista
ATAG_CORE	Delimita l'inizio di una lista
ATAG_MEM	Descrive l'area fisica di memoria
ATAG_VIDEOTEXT	Utilizzata per descrivere
ATAG_RAMDISK	Descrive come la ramdisk deve essere configurata dal kernel
ATAG_INITRD2	Descrive la locazione fisica della ramdisk
ATAG_SERIAL	Numero seriale della board
ATAG_REVISION	Tag per la revisione della board
ATAG_VIDEOLFB	Descrive i parametri per il tipo di framebuffer del display
ATAG_CMDLINE	Utilizzata per passare comandi al kernel

Tabella 2.1: Lista di tag utilizzabili

Caricare l'immagine del kernel Il caricamento dell'immagine del kernel comporta alcune operazioni preliminari illustrate di seguito:

1. disabilitare i trasferimenti

2. cofigurare nel modo seguente i registri della CPU:

   r0 = 0

   r1 = numero di macchina

   r2 = indirizzo fisico della tagged list in RAM

3. modalità CPU: disabilitare ogni tipo di interrupts. La CPU deve trovarsi in modalità SVC (supervisore)

4. caches, MMUs:

   MMu deve essere disabilitata

   La cache istruzioni può essere attiva

   La cache dati deve essere disabilitata e vuota

5. il boot loader deve saltare alla prima istruzione dell'immagine del kernel per poterne avviare l'esecuzione.

2.5 Programmare il boot loader

L'U-Boot può essere configurato dall'utente che può agire sulla modifica delle variabili d'ambiente che possono essere salvate in modo persistente nella memoria flash.

Variabili d'ambiente Le variabili d'ambiente possono essere:

- inizializzate e modificate con l'utilizzo del comando "setenv";

  ```
  BBG U-Boot > setenv ipaddr 192.168.0.10
  BBG U-Boot > setenv ethaddr 00:04:9f:01:07:4c
  ```

  l'uso del comando setenv seguito solo dal nome della variabile ne cancella il contenuto

- stampate a video con il comando "printenv"

  ```
  BBG U-Boot > printenv ipaddr
  ipaddr=192.168.0.10
  BBG U-Boot > printenv
  ipaddr=192.168.0.10
  ethaddr 00:04:9f:01:07:4c
  ```

- salvate sulla memoria flash con il comando "saveenv".

Ecco una lista delle variabili d'ambiente più utilizzate:

- bootcmd: definisce una stringa di comandi che viene automaticamente eseguita all'avvio del sistema se non viene premuto alcun tasto sulla tastiera;
- bootargs: viene utilizzata per passare argomenti al comando bootm;
- bootfile: generalmente utilizzato per indicare il nome dell'immagine da caricare attraverso il comando tftp;
- ipaddr: indirizzo ip della macchina target;
- loadaddr: indirizzo fisico della memoria RAM utilizzato di default dal comando bootp per l'esecuzione di un'immagine;
- serverip: indirizzo ip del server tftp o NFS;
- netmask: sottomaschera di rete.

Comandi di rete L'U-Boot supporta il protocollo TFTP (Trivial File Transfer Protocol), una versione meno evoluta del FTP che non richiede l'autenticazione dell'utente che può essere utilizzata per caricare immagini nella memoria RAM della board. Il comando tftp necessita di due informazioni, il nome del file da caricare e l'indirizzo di memoria a partire dal quale il file verrà salvato. Per esempio il comando

```
BBG U-Boot > tftp 0x90800000 uImage-babbage
```

trasferisce il file uImage-babbage dalla macchina in cui è attivo il server tftp nella locazione di memoria con indirizzo 0x90800000 della macchina target.

Questo comando è molto utile perchè non solo può essere utilizzato per caricare in memoria l'immagine del kernel ma anche per caricare le immagini contenenti piccoli script che possono essere utilizzati per settare in maniera più efficiente le variabili di sistema.

L'U-Boot supporta anche i protocolli NFS e PING.

Eseguire uno script Affinchè un file sia letto e utilizzato dall'U-Boot è necessario che esso sia in un formato specifico. Per costruire un tale file si utilizza **mkimage**, un programma contenuto nella directory `/home/<username>/opt/myandroid/kernel_imx/bootable/bootloader/tools`, che viene generato al termine della compilazione del boot loader. Questo programma aggiunge al file che riceve in ingresso come perametro un header di 64 byte contenente utili informazioni quali l'architettura target, il sistema operativo, il tipo di immagine, il metodo di compressione, l'entry points, il CRC32 checksum, etc. Per esempio costruiamo un semplice script "network" che stampa a video alcune informazioni utili riguardanti lo stato della rete della macchina target come segue:

```
echo
echo Network Configuration:
echo ----------------------
echo Target:  printenv ipaddr hostname
echo
echo Server:  printenv serverip rootpath
echo
```

dove ipaddr, hostname, serverip e rootpath sono delle veriabili locali all'U-Boot precedentemente settate con il comando printenv e salvate con il comando saveenv.

Con il comando

```
$ ./home/<username>/opt/myandroid/kernel_imx/bootable/
    bootloader/uboot-imx/tools/mkimage -A arm -O linux -T
```

```
script -C none -a 0 -e 0 -n network configuration -d
network network.img
```

verrà prodotta, a partire dallo script “network”, l’immagine “network.img”. Significato delle opzioni:

- -A definisce l’architettura (alpha, arm, x86, ia64, m68k, microblaze, mips, mips64, nios, nios2, powerpc, ppc, s390, sh, sparc, sparc64, blackfin, avr32)
- -O definisce il sistema operativo (linux, lynxos, netbsd, rtems, u-boot, qnx, vxworks, integrity, 4_4bsd, dell, esix, freebsd, irix, ncr, openbsd, psos, sco, solaris, svr4)
- -T definisce il tipo di immagine (filesystem, firmware, kernel, multi, ramdisk, script, standalone, flat_dt)
- -C definisce il tipo di compessione (none, bzip2, gzip, lzma)
- -a definisce il load address in esadecimale (indirizzo a partire dal quale verrà caricata l’immagine)
- -e definisce l’entry point in esadecimale (indirizzo a partire dal quale l’immagine verrà eseguita)
- -n definisce il nome dell’immagine
- -d definisce il payload dell’immagine (nell’esempio il file network verrà utilizzato come payload dell’immagine finale network.img)

Una volta copiata l’immagine “network.img” all’interno della directory `/tftpboot` sarà possibile caricarla nella memoria della macchina target utilizzando il seguente comando:

```
BBG U-Boot > tftp 0x90800000 network.img
```

Lo script viene eseguito con il seguent comando:

```
BBG U-Boot > autoscr 0x90800000
```

Nel nostro esempio lo script è stato caricato in memoria a partire dall’indirizzo 0x90800000, è stato spostato dall’U-Boot all’indirizzo 0x0 come definito dall’opzione -a ed eseguito a partire dall’indirizzo 0x0 come definito dall’opzione -e.

Questa procedura può risultare molto utile perchè permette, ad esempio, di configurare il boot loader attraverso l’esecuzione di uno script e di switchare da una configurazione ad un altra in modo semplice e veloce.

Comando bootm Viene utilizzato per avviare le immagini del sistema operativo. Può essere utilizzato in due modi secondo le seguenti sintassi:

```
BBG U-Boot > bootm $(kernel_addr)
```

```
BBG U-Boot > bootm $(kernel_addr) $(ramdisk_addr)
```

Nel primo caso l'U-Boot legge l'immagine contenuta a partire dall'indirizzo definito dalla variabile *kernel_addr*, estrae dal suo header utili informazioni quali il tipo di sistema operativo, il tipo di compressione utilizzato, il load address e l'entry point. L'immagine viene quindi caricata nell'indirizzo di memoria definito da *load_address*, viene decompressa nel modo adeguato e in base al sistema operativo gli vengono passati o no ulteriori argomenti. Infine ne viene avviata l'esecuzione a partire dall'entry point.

Nel secondo caso gli argomenti passati al comando bootm sono due, entrambi indirizzi di memoria, uno localizzante la posizione del kernel e l'altro localizzante la posizione della ramdisk. In questo caso l'U-Boot decomprime e copia in RAM il kernel, carica in RAM anche la ramdisk, e prima di eseguire il kernel a partire dall'indirizzo definito dall'entry point gli passa informazioni riguardanti la posizione in memoria della ramdisk.

Comando run Viene utilizzato per eseguire comandi contenuti in una variabile d'ambiente. Infatti una variabile può essere utilizzata sia per contenere dei parametri che dei comandi.

Se una variabile contiene più di un comando (separati da ;) e uno di questi non va a buon fine, i restanti verranno comunque eseguiti. Invece se vengono eseguite più variabili con un'unica chiamata run, ogni errore nell'esecuzione di un comando causerà la terminazione della run implicando la non esecuzione delle restanti variabili.

Comando mmc read Definito dalla seguente sintassi

```
BBG U-Boot > mmc read <dev> <addr> <block> <count>
```

Riceve come parametri il dispositivo che deve essere letto, l'indirizzo di memoria RAM dove caricare le informazioni lette, il blocco di memoria a partire dal quale iniziare la lettura e il numero di blocchi da leggere. *block* e *count* sono espressi in esadecimale e ogni blocco ha una dimensione pari a 512 byte.

2.6 Booting Android

É possibile avviare il sistema operativo Android utilizzando due modalità:

1. via TFTP/NFS: consiste nel caricare il kernel mediante il protocollo tftp e montare il filesystem localizzato su server remoto mediante il protocollo NFS;

2. via MMC/SD: il bootloader, il kernel, la ramdisk e l'intero filesystem sono localizzari all'interno della MMC

Inoltre è possibile scegliere se avviare l'U-Boot da SD o da SPIN-OR.

Nel primo caso è necessario copiare l'immagine dell'U-Boot in una SD correttamente formattata. Questo può essere fatto su ambienti Unix utilizzando il comando dd.

Dopo aver identificato il device node assegnato alla MMC/SD card con il comando

```
$ cat /proc/partitions
major minor  #blocks     name
  8     0     78125000   sda
  8     1     75095811   sda1
  8     2            1   sda2
  8     5      3028221   sda5
  8    32    488386584   sdc
  8    33    488386552   sdc1
  8    16      3921920   sdb
  8    18      3905535   sdb1
```

Eseguire l'istruzione:

```
$ sudo dd if=/path-to-imx51bgg-image/u-boot-no-padding.bin
    of=/dev/sdc bs=1K seek=1
```

Con questa operazione l'U-Boot viene copiato sulla SD con un offset di un Kilobyte (bs * seek) per non sovrascrivere il MBR che è situato nei primi 512 byte della SD.

Una volta copiato l'U-Boot sulla SD inseriamo la stessa nello slot presente sul retro della board e prima di accendere il sistema assicurarsi che lo switch boot mode sia configurato in modo opportuno come mostrato in tabella 2.2. All'avvio viene eseguito l'U-Boot che mostra un countdown scaduto il quale il sistema viene avviato con le configurazioni di default. Per evitare questo va premuto un tasto sulla tastiera, il countdown viene bloccato rendendo così possibile all'utente l'interazione con il bootloader.

Nel secondo caso occorre copiare l'U-boot all'interno della memoria flash. Questo può essere fatto nel modo seguente:

1. avviare la macchina target con boot via MMC/SD

2. caricare il file u-boot.bin nella directory */tftpboot*

Boot mode	D1	D2	D3	D4	D5	D6	D7	D8	D9	D0
SPI-NOR	0	0	1	1	1	0	1	1	0	0
MMC-1	0	0	0	0	0	0	1	1	0	0

Tabella 2.2: Switch boot mode

3. caricare l'immagine dell'U-boot nella memoria RAM della macchina target

```
BBG U-Boot > setenv serverip 192.168.0.2
BBG U-Boot > setenv ethaddr  00:04:9f:01:07:4c
BBG U-Boot > setenv ipaddr   192.168.0.10
BBG U-Boot > setenv loadaddr 0x90800000
BBG U-Boot > tftp ${loadaddr} u-boot.bin
```

4. copiare il file u-boot-no-padding.bin nella memoria flash a partire dall'indirizzo 0x0

   - inizializzare la memoria

     ```
     BBG U-Boot > sf probe 1
     ```

   - cancellare il contenuto della memoria

     ```
     BBG U-Boot > sf erase 0 0x40000
     ```

   - copiare l'immagine nella memoria

     ```
     BBG U-Boot> sf write ${loadaddr} 0 0x40000
     ```

5. configurare lo switch boot mode come mostrato in tabella 2.2.

2.6.1 TFTP/NFS

Le variabili utilizzate sono:

Variabili	Valore	Descrizione
ethaddr	00:04:9f:01:07:4c	Indirizzo MAC dell'interfaccia ethernet della macchina target
serverip	192.168.0.2	Indirizzo ip del server contenente il filesystem e il kernel

ipaddr	192.168.0.10	Indirizzo ip della macchina target
netmask	255.255.255.0	Netmask
loadaddr	0x90800000	Indirizzo di memoria RAM nel quale verra caricato il kernel
bootfile	uImage	Identifica il nome del kernel
nfsroot	/nsfroot/android_fs	Path che localizza sulla macchina host la posizione del filesystem di Android

Al kernel vengono passati i seguenti parametri:

Parametro	Valore	Descrizione
console	tty0	Console utilizzata dal kernel per stampare informazioni di log
console	ttymxc0,115200	Console utilizzata da Android per stampare informazioni di log
loglevel	8	Tutti i messaggi con un loglevel minore di 8 verranno stampati a video (utile in fase di debug)
ip	$(ipaddr)	Indirizzo ip della macchina target
root	/dev/nfs	Necessario per abilitare uno pseudo dispositivo NFS. Serve ad indicare al kernel che non verrà utilizzato un dispositivo reale per montare il file system.

nfsroot	nfsroot=${serverip}: ${nfsroot}, v3,tcp	Localizza, all'interno della macchina con ip ${serverip} la directory da montare come root file system. v3 e tcp indicano rispettivamente la versione NFS e il protocollo di trasporto utilizzati.
mem	400M	Individua la memoria fisica che può essere utilizzata dal kernel
init	/init	Indica al kernel dove è contenuto il processo init. Deve essere specificato poichè di default il kernel cerca l'init in /sbin
video	mxcfb:1024x768M-16@60	Driver per l'uscita video con indicati la risoluzione e la profondità di colore

Programmare l'U-Boot Inizializziamo le variabili con il comando setenv:

```
BBG U-Boot > setenv ethaddr 00:04:9f:01:07:4c
BBG U-Boot > setenv serverip 192.168.0.2
BBG U-Boot > setenv ipaddr 192.168.0.10
BBG U-Boot > setenv netmask 255.255.255.0
BBG U-Boot > setenv loadaddr 0x90800000
BBG U-Boot > setenv bootfile uImage
BBG U-Boot > setenv nfsroot /nfsroot/android_fs
```

Definiamo il bootargs contenente i parametri da passare al kernel:

```
BBG U-Boot > setenv bootargs_base 'setenv bootargs console=
    tty0 console=ttymxc0,115200 loglevel=8'
BBG U-Boot > setenv bootargs_nfs 'setenv bootargs ${bootargs
    } ip=${ipaddr} root=/dev/nfs nfsroot=${serverip}:${
    nfsroot},v3,tcp'
BBG U-Boot > setenv bootargs_android 'setenv bootargs ${
    bootargs} mem=400M init=/init'
BBG U-Boot > setenv bootargs_video 'setenv bootargs ${
    bootargs} video=mxcfb:1024x768M-16@60'
```

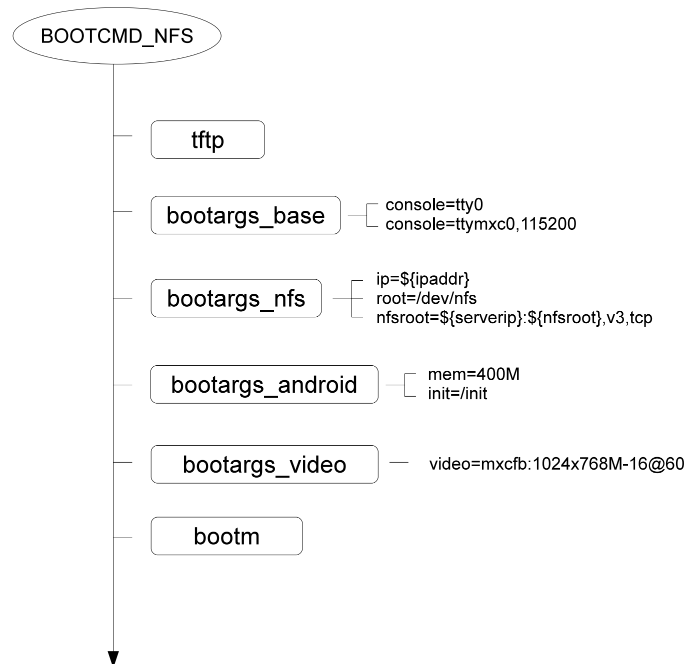


Figura 2.2: NFS

Definiamo il comando che carica il kernel in memoria e inizializza il bootargs:

```
BBG U–Boot >  setenv bootcmd_NFS 'tftp ${loadaddr} ${
    bootfile}; run bootargs_base bootargs_nfs
    bootargs_android'
```

Carichiamo il kernel in memoria e avviamo il sistema:

```
BBG U–Boot > setenv bootcmd 'run bootcmd_NFS bootargs_video;
     bootm ${loadaddr}'
```

A questo punto salviamo la nostra configurazione e avviamo il systema:

```
BBG U–Boot > saveenv
BBG U–Boot > bootd
```

La sequenza temporale delle azioni eseguite dopo il comando bootd è illustrata in figura 2.2

2.6.2 SD

Per prima cosa copiamo le immagini del kernel e della ramdisk all'interno della SD. Come visto nel caso dell'U-Boot questa operazione può essere compiuta con il comando dd.

Eseguiamo:

```
$ sudo dd if=/path-to-imx51bbg-image/uImage of=/dev/sdc bs=
    1M seek=1
$ sudo dd if=/path-to-imx51bbg-image/uramdisk.img of=/dev/
    sdc bs=4M seek=1
```

L'immagine del kernel viene copiata con un offset di 1M mentre la ramdisk viene copiata con un offset di 4M. In questo modo si evita di sovrascrivere le immagini precedentemente copiate.

Ora non resta che copiare all'interno della SD il filesystem di Android. Utilizziamo il comanfo fdisk per creare quattro partizioni, tre primarie e due logiche.

La tabella delle partizioni creata deve essere configurata nel modo seguente:

Tipo di partizione / indice	Dispositivo	File System	Contenuto / dimensione
Primaria 1	/dev/sdc1	VFAT montata come /sdcard	File multimediali dell'utente
Primaria 2	/dev/sdc2	EXT3 montata come /system	Binari e librerie di Android / almeno 100MB
Estesa 3	/dev/sdc3		
Logica in estesa	/dev/sdc5	EXT3 montata come /data	Dati di Android (es. applicazioni)
Logica in estesa	/dev/sdc6	EXT3 montata come /cache	Cache di Android
Primaria 4	/dev/sdc4	EXT3 montata come /recovery	Root file System per Android recovery mode / almeno 10MB

Dopo aver partizionato la SD creiamo per ciascuna partizione il file system appropriato.

```
$ mkfs.vfat /dev/sdc1
$ mkfs.ext3 /dev/sdc2
$ mkfs.ext3 /dev/sdc4
$ mkfs.ext3 /dev/sdc5
$ mkfs.ext3 /dev/sdc6
```

Ora popoliamo la SD copiandovi le immagini system.img e recovery.img

```
$ sudo dd if=/path-to-imx51bbg-image/system.img of=/dev/sdc2
```

```
$ sudo dd if=/path−to−imx51bbg−image/recovery.img of=/dev/
    sdc4
```

Le variabili utilizzate nel boot da SD sono:

Variabili	Valore	Descrizione
loadaddr	0x90800000	Indirizzo di memoria RAM a partire dal quale verrà caricato il kernel
rd_loadaddr	0x90B00000	Indirizzo di memoria RAM a partire dal quale verrà caricata la ramdisk

Al kernel vengono passati i seguenti parametri:

Parametro	Valore	Descrizione
console	tty0	Console utilizzata dal kernel per stampare informazioni di log
console	ttymxc0,115200	Console utilizzata da Android per stampare informazioni di log
loglevel	8	Tutti i messaggi con un loglevel più piccolo di 8 verranno stampati a video (utile in fase di debug)
mem	400M	Individua la memoria fisica che può essere utilizzata dal kernel
init	/init	Indica al kernel dove è contenuto il processo init. Di default il kernel cerca l'init in /sbin/init
video	mxcfb:1024x768M-16@60	Driver per l'uscita video con indicati la risoluzione e la profondità di colore

Programmare l'U-Boot Inizializziamio le variabili:

```
BBG U−Boot > setenv loadaddr 0x90800000
BBG U−Boot > setenv rd_loadaddr 0x90B00000
```

Configuriamo il bootargs per il passaggio dei parametri al kernel

```
BBG U-Boot > setenv bootargs_base 'setenv bootargs console=
    tty0 console=ttymxc0,115200 loglevel=8'
BBG U-Boot > setenv bootargs_android 'setenv bootargs ${
    bootargs} mem=400M init=/init'
BBG U-Boot > setenv bootargs_video 'setenv bootargs ${
    bootargs} video=mxcfb:1024x768M-16@60'
```

Definiamo il comando che inizializza il bootargs

```
BBG U-Boot > setenv bootcmd_SD1 'run bootargs_base
    bootargs_android bootargs_video'
```

Definiamo il comando che carica il kernel e la ramdisk in memoria e li esegue

```
BBG U-Boot > setenv bootcmd_SD2 'mmc read 0 ${loadaddr} 0
    x800 0x1280; mmc read 0 ${rd_loadaddr} 0x2000 0x258;
    bootm ${loadaddr} ${rd_loadaddr}'
```

Definiamo i comandi che permettono l'avvio del sistema

```
BBG U-Boot > setenv bootcmd_SDCard 'run bootcmd_SD1
    bootcmd_SD2'
BBG U-Boot > setenv bootcmd 'run bootcmd_SDCard'
```

Salviamo la nostra configurazione e facciamo il reboot del sistema

```
BBG U-Boot > saveenv
BBG U-Boot > bootd
```

La sequenza temporale delle azioni eseguite dopo il comando bootd è illustrata in figura 2.3

2.6.3 Modalità video touch-screen

In entrambe le modalità di boot è stata selezionata come uscita video l'interfaccia DVI poichè risulta più comoda in fase di debug. Volendo è possibile scegliere di visualizzare le informazioni di log e l'avvio del sistema sullo schermo led touch screen, inserendo nel bootargs, tra i parametri da passare al kernel, l'uscita WVGA.

Modifichiamo per entrambe le configurazioni la variabile bootargs_video come segue:

```
BBG U-Boot > setenv bootargs_video 'setenv bootargs ${
    bootargs} wvga calibration'
```

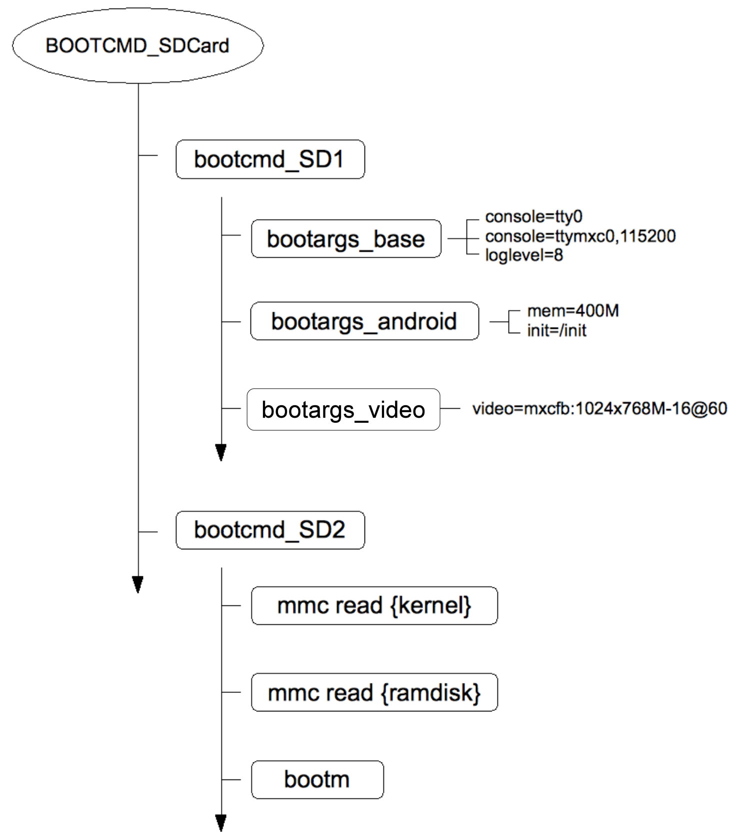


Figura 2.3: SD

La stringa calibration serve per la calibrazione del touch screen fatta un'unica volta al primo avvio del sistema con tale configurazione. Sullo schermo, in fase di calibrazione, verranno visualizzate in sequenza delle crocette che devono essere premute per completare la calibrazione dello schermo.

2.6.4 Problemi e soluzioni

La procedura di boot da SD viene completata con successo ma il sistema non è in grado di rilevare un'interfaccia di rete rendendo impossibile la connessione ad un network. Poichè l'interfaccia ethernet risulta essere abilitata durante il boot via NFS, si è pensato che un utilizzo di qualche comando di rete durante la fase di boot la potesse attivare rendendola utilizzabile anche dopo il completamento dell'avvio di sistema.

Il boot loader è stato quindi programmato con l'inserimento del comando **ping** come prima azione eseguita dall'U-boot. Per poter funzionare, il comando ping deve essere accompagnato da una minima configurazione dell'interfaccia di rete, ovvero dall'impostazione di un indirizzo ethernet e di un indirizzo ip.

- ethaddr: 00:04:9f:01:07:4
- ipaddr: 192.168.0.10

Nella procedura di boot è stato modificato il **bootcmd_SDCard** in questo modo

```
BBG U–Boot > setenv bootcmd_SDCard 'ping ${ipaddr}; run
    bootcmd_SD1 bootcmd_SD2'
```

Nel momento in cui si fa il reboot il boot loader prima di eseguire il bootcmd_SD1 e il bootcmd_SD2 fa un ping verso l'ip *ipaddr*, in questo modo, al completamento dell'avvio di sistema, questa risulta essere attiva e quindi configurabile. L'indirizzo ip assegnato all'interfaccia al livello di boot loader non viene passato al kernel come invece accade per il boot via NFS e anche in presenza di un server DHCP attivo l'interfaccia eth0 risulta avere sempre l'indirizzo ip pari a 0.0.0.0. Per ovviare a questo problema si è pensato di agire sul file *init.rc*, ovvero il file di configurazione di sistema che viene letto ed eseguito dal processo */init*, che è il primo processo in user space eseguito dal kernel.

Il file *init.rc* è contenuto nella cartella `/home/<username>/opt/myandroid/vendor/fsl/imx51_BBG` e in fase di compilazione viene compresso all'interno dell'immagine *ramdisk.img*. Le azioni più importanti definite all'interno dell'init.rc riguardano il settaggio delle variabili di sistema mediante il comando **export**, la costruzione dell'intero file system con il montaggio delle partizioni

e il settaggio dei permessi di accesso, l'avvio di servizi come il daemon adb o il processo zygote. Un servizio, ovvero un programma che può essere lanciato dal processo init, viene definito con la seguente sintassi:

```
service <name> <pathname> [ <argument> ]*
    <option>
    <option>
```

Alla fine del nostro init.rc è stato definito il seguente servizio:

```
service net-setup /net.sh
    oneshot
```

che ha il compito di lanciare lo script *net.sh* una sola volta all'avvio del sistema.

Lo script *net.sh*

```
#!/system/bin/sh
netcfg eth0 up
netcfg eth0 dhcp
```

attiva l'interfaccia ethernet della board e le assegna un indirizzo ip via dhcp.

Ora non rimane che modificare l'init.rc e inserire all'interno del nodo radice `/`, che corrisponde al punto di montaggio della ramdisk, lo script net.sh. Per poter modificare la ramdisk bisogna seguire i seguenti passi ricordando che si deve operare sulla ramdisk.img e non sulla uramdisk.img:

```
$ mkdir tmp
$ cp ramdisk.img tmp/ramdisk.cpio.gz
$ cd tmp
$ sudo gzip -d ramdisk.cpio.gz
$ sudo cpio -i -F ramdisk.cpio
$ rm ramdisk.cpio
```

In questo modo all'interno della cartella `tmp` sarà possibile trovate il contenuto della ramdisk. Dopo aver modificato il file init.rc occorre copiare all'interno della cartella `tmp` lo script net.sh. Una volta copiato bisogna ricostruire l'immagine. All'interno della cartella `tmp` digitare quanto segue:

```
$ sudo su
# find . | cpio --quiet -H newc -o | gzip -9 -n > ../ramdisk
    .img
```

Infine bisogna aggiungere all'immagine creata l'appropriato hearder per renderla utilizzabile dall'U-boot

```
$ . /opt/myandroid/bootable/bootloader/uboot-imx/tools/
    mkimage -A arm -O linux -T ramdisk -C none -a 0x90308000
    -n "Android Root Filesystem" -d ./ramdisk.img ./uramdisk.
    img
```

Non resta ora che fleshare l'immagine all'interno della SD come mostrato nella sottosezione 2.6.2.

Per poter usufruire del servizio DNS, qualora disponibile, aprire una shell sul dispositivo via adb

```
$ adb shell
```

e impostare il server DNS

```
# setprop net.dns1 <IP_OF_YOUR_PRIMARY_DNS_SERVER>
```

Capitolo 3

Protocollo SIP

3.1 Introduzione al VOIP

Voice over IP (Voip) è una tecnologia che sfrutta reti utilizzanti il protocollo IP per trasportare un particolare tipo di traffico: la digitalizzazione del parlato di una conversazione telefonica. Più specificamente con VoIP si intende l'insieme dei protocolli di comunicazione di strato applicativo che rendono possibile questo tipo di comunicazione.

Grazie a numerosi provider VoIP e a gateway dedicati è possibile effettuare telefonate anche verso la rete telefonica tradizionale (PSTN) e qualla mobile (GSM). Diversamente dalla rete telefonica tradizionale, che per abilitare la comunicazione tra due endpoint deve necessariamente stabilire un cammino che li colleghi (commutazione di circuito) il Voip, come del resto tutti i protocolli che si appoggiano su IP, non stabilisce alcun percorso ad uso esclusivo prevedendo un'allocazione dinamica delle risorse e un routing end to end basato sulla commutazione di pacchetto. Dal punto di vista dello sfruttamento delle risorse, un sistema basato sulla commutazione di pacchetto risulta più efficiente di un sistema basato sulla commutazione di circuito ma difetta di problemi dovuti alla congestione e al conseguente ritardo nella consegna dei pacchetti.

Fra i vantaggi rispetto alla telefonia tradizionale si annoverano:

- minore costo per chiamata, specialmente su lunghe distanze, poichè quello che si paga al proprio provider è l'accesso alla rete. Una volta connessi alla rete si potranno effettuare telefonate illimitate verso altri utenti Internet nel mondo. Se si necessita di chiamare un utente sulla rete telefonica tradizionale il provider che mette a disposizione un servizio di tipo "Dial-out Gateway" cercherà di distribuire getaway sul territorio in modo da essere il più vicino possibile alla maggior parte delle destinazioni. Nel caso di una telefonata internazionale l'utente dovrà pagare

solamente il servizio telefonico corrispondente a una chiamata effettuata dal punto in cui è situato il gateway alla destinazione finale.

- nuove funzionalità avanzate come per esempio la videochiamata, l'utilizzo dell'instant messaging o una conferenza audio/video;

- l'implementazione di future opzioni prevedono modifiche software e non hardware.

Gli standard principali per la gestione delle comunicazioni Voip sono: H.323 (definito dall'ITU) e il più diffuso SIP (definito invece dall'IETF). Esistono poi tutta una serie di protocolli proprietari che utilizzano una rete di tipo peer-to-peer. L'esempio più noto è dato da Skype, il software più utilizzato per la telefonia internet, che presenta delle prestazioni e una qualità audio eccellente ma non consente l'interoperabilità con dispositivi VoIP basati sui due standard citati sopra se non mediante l'utilizzo di gateway dedicati e non può essere utilizzato su reti locali ma necessita dell'interfacciamento alla rete pubblica.

3.2 Protocollo SIP

Il protocollo SIP (Session Initiation Protocol) è un protocollo di controllo a livello applicativo per creare, modificare e terminare sessioni multimediali tra due o più partecipanti. È stato accettato come standard dalla IETF nel marzo del 1999 quando è stata rilasciata la prima stesura delle specifiche (RFC 2543). La seconda, nonché attuale stesura, risale a maggio 2002 (rfc 3261). Le applicazioni più comuni di tale protocollo sono la telefonia su IP, lo streaming audio-video, le conferenze e la messaggistica istantanea.

SIP è un protocollo orientato allo scambio di messaggi e al Web con una struttura simile ad HTTP/1.1 che gli permette di utilizzare paradigmi di tipo "Client-Server"con l'instaurazione di comunicazioni di tipo Punto-Punto.

Tra i principali punti di forza del protocollo SIP ricordiamo la sua scalabilità (intrinseca nella sua architettura) e l'interoperabilità con sistemi preesistenti e sistemi "SIP-based". La scalabilità del protocollo deriva dal suo sistema di instaurazione delle sessioni di comunicazione, che prevede l'utilizzo di un server per la gestione delle connessioni solo durante le fasi di instaurazione, modifica e rilascio della connessione stessa. Durante la conversazione il flusso di dati avviene direttamente tra gli elementi terminali, non aumentando il carico della rete e permettendo così al server di gestire numerose connessioni. L'interoperabilità invece scaturisce dalla struttura modulare del protocollo che

consiste di un limitato set di funzionalità sulle quali è possibile basare delle estensioni che ne consentono utilizzi più complessi.

La descrizione del protocollo contenuta nella rfc 3261 è da considerare un insieme di direttive non imperative, mentre l'implementazione delle funzionalità è lasciata libera. Grazie all'assenza di limitazioni pertanto, nel rispetto dello standard "de facto", la progettazione e lo sviluppo risultano facilitati permettendo una più veloce diffusione del protocollo.

3.2.1 Struttura del protocollo

Il protocollo SIP è strutturato a strati, consentendo una descrizione dei comportamenti dei singoli strati in termini di un insieme di processi semplici ed indipendenti, limitando lo studio sull'interazione tra livelli adiacenti. Non è necessario che gli elementi specificati dal protocollo contengano tutti gli strati definiti. Inoltre spesso tali strati sono soltanto logici e non fisici.

TU
TRANSACTION
TRANSPORT

Il livello più basso è quello di trasporto che definisce le modalità in cui vengono inviate le richieste e ricevute le risposte all'interno della rete.

Il secondo strato è quello di transaction che si occupa della gestione delle transazioni, ovvero dello scambio di messaggi tra uno User Agent Client e uno User Agent Server che ha inizio con l'invio di un messaggio di request da parte dello UAC e si conclude con un messaggio di risposta inviato dallo UAS. Gestire una transazione vuol dire tenere traccia delle richieste inviate e delle relative risposte, e controllare, attraverso l'ausilio di un time-out per l'eventuale ritrasmissione, che ad ogni richiesta sia associata una risposta.

Sullo strato di transazione si stabilisce un terzo strato detto Transaction User (TU). Ogni entità SIP, ad esclusione del proxy stateless è un'entità di tipo TU. Quando viene effettuata una richiesta, il TU crea un'istanza di transazione a cui associa un indirizzo ip di destinazione, una porta ed un protocollo di trasporto. Il TU stesso può cancellare l'istanza creata bloccando tutto il processo e generando una risposta d'errore specifica per quella transazione.

3.2.2 Protocolli di supporto

Il protocollo SIP in quanto tale non definisce tutte le funzioni richieste per stabilire una sessione interattiva multimediale. Agenti e applicazioni SIP

richiedono la presenza di altri protocolli per svolgere le seguenti operazioni:

- Descrizione delle caratteristiche di una sessione. Definiscono una sessione audio o video, i codec impiegati, le sorgenti multimediali e gli indirizzi di destinazione.
- Gestione dei contenuti multimediali. Questi protocolli controllano e trasmettono pacchetti audio e video all'interno di una sessione.
- Supporto a servizi addizionali.

Le sessioni configurate all'interno di una rete SIP utilizzano in genere i protocolli IETF indicati di seguito:

DNS (*Domain Name Server*). La configurazione di una sessione SIP può richiedere l'utilizzo del protocollo DNS per risolvere i nomi di host o di dominio rispetto a indirizzi IP globali instradabili. Il protocollo DNS consente anche di distribuire e condividere il traffico tra più server su un cluster identificato da un nome host.

SDP (*Session Description Protocol*). Utilizzato nel corpo di un messaggio SIP per descrivere i parametri di una sessione multimediale. Le informazioni in questione includono il tipo di sessione, per esempio audio, video o entrambi, i parametri quali codec o porte necessarie per configurare uno stream multimediale. RFC 2327

RTP (*real-time Transport Protocol*). Definito nella RFC 1889 consente di trasmettere dati in tempo reale tra terminali endpoint coinvoliti in una sessione. Il protocollo RTCP (*Real-time Transport Control Protocol*), definito nella RFC 3550, specifica operazioni di monitoraggio e di reporting al mittente dei livelli di (*QoS*) relativi al corrispondente stream RTP.

RSVP (*Resource reSerVation Protocol*), serve per riservare determinate risorse di rete prima di stabilire la sessione multimediale.

TLS (*Transport Layer Security*). Definito dalle specifiche RFC 2246, permette di definire il meccanismo di privacy e di integrità delle informazioni di segnalazione SIP scambiate all'interno della rete. Stabilisce operazioni di mutua autenticazione tra applicazioni client e server, la negoziazione di algoritmi di crittografia dei dati e l'impostazione di chiavi di crittografia prima della trasmissione di informazioni di segnalazione all'interno della rete.

STUN (*Simple Traversal of UDP through NAT*). Permette agli User Agent Client di individuare la presenza e la tipologia di NAT disponibile tra gli User Agent e la rete Internet pubblica. Consente anche al client di individuare l'indirizzo IP pubblico che risulta allocato dal servizio NAT. Questa procedura non può essere utilizzata nei sistemi che fanno uso del NAT simmetrico ovvero quel sistema che mappa i messaggi request provenienti da un indirizzo IP o numero di porta TCP interni rivolti a un determinato indirizzo IP e porta destinazione ad uno stesso indirizzo IP e porta sorgente esterni.

I protocolli descritti non costituiscono un elenco esaustivo dei protocolli che possono essere impiegati da un sistema SIP il quale non richiede necessariamente l'utilizzo di tutti.

3.2.3 Messaggi

I messaggi si dividono principalmente in richieste (Request) da client a server e risposte (Response) da server a client. Sia le richieste, sia le risposte scambiate dal protocollo seguono il formato standard definito nella rfc 2822 anche se integrano sintassi e set di caratteri anche non previsti in questo standard. Entrambe le tipologie di messaggio sono costituite da una start-line, uno o più campi di intestazione (header) una linea vuota usata come separatore tra le intestazioni ed il corpo vero e proprio del messaggio (body), che è da considerarsi opzionale.

```
generic-message  =  start-line
                    *message-header
                    CRLF
                    [ message-body ]
start-line       =  Request-Line / Status-Line
```

Request

I messaggi Request hanno come start-line una request-line costruita come segue:

```
Request-Line  =  Method SP Request-URI SP SIP-Version CRLF
```

Contenente il metodo di richiesta, l'URI (Uniform Resurce Identifier) a cui è indirizzato il messaggio e la versione del protocollo utilizzata. Secondo le specifiche di SIPv2 sono previsti sei metodi:

1. REGISTER: utilizzato quando uno User Agent vuole registrare presso un Registrar Server il proprio punto di ancoraggio alla rete;

2. INVITE: serve ad invitare un utente a partecipare ad una sessione; una risposta affermativa al messaggio INVITE (risposta 200), include il tipo di informazioni del chiamante. Con questo metodo gli utenti possono riconoscere le funzionalità offerte dall'altro capo della comunicazione e aprire una sessione di comunicazione con uno scambio limitato di messaggi;

3. ACK: messaggio di riscontro, serve per concludere la transazione iniziata con il comando INVITE;

4. CANCEL: utilizzato per terminare un dialogo quando la sessione non ha ancora avuto inizio;

5. BYE: utilizzato per interrompere la comunicazione tra due utenti;

6. OPTIONS: serve per interrogare e raccogliere le funzionalità degli agenti utente e dei server di rete.

É comunque data la possibilità di estendere il protocollo con l'aggiunta di ulteriori metodi di richiesta come:

1. PUBLISH: usato per pubblicare informazioni legate a determinati eventi come per esempio l'evento presenza e l'evento registrazione. L'utilizzo di tale metodo richiede l'implementazione di un presence Agent sul lato client e server. Ulteriori informazioni sono reperibili nella rfc 3903;

2. SUBSCRIBE/NOTIFY: definiti nelle rfc 3265 permettono ad una entità nella rete di sottoscrivesi ad un determinato tipo di evento, per esempio un evento message-summary, inviando un messaggio di tipo SUBSCRIBE a un altra entità SIP detentrice della risorsa per la quale si desidera avere notizie su eventuali cambi di stato. Il metodo NOTIFY serve appunto per notificare ad un sottoscrittore l'evento per il quale esso si è registrato;

3. UPDATE: definito nella rfc 3311 viene utilizzato quando un'entità SIP coinvolta in un dialogo vuole aggiornare una sessione già istanziata. Un'entità che desidera aggiornare la sessione nella quale è coinvolta costruisce un messaggio di tipo UPDATE nel quale specifica l'offerta SDP alla quale si desidera ottenere l'aggiornamento. L'entità che riceve questa richiesta ingloba la risposta all'aggiornamento in un messaggio del tipo 2xx;

4. REFER: definito nella rfc 3515 può essere utilizzato ad esempio per abilitare qualche applicazione come un trasferimento di chiamata. Se un'entità SIP A, che ha stabilito un dialogo con un'altra entità B desidera che quest'ultima si metta in contatto con una terza entità C non deve

fare altro che costruire una richiesta di tipo REFER con all'interno informazioni sul contatto C e inviarlo a B. In seguito lo User Agent di B segnalerà ad A se è riuscito o meno a contattare C.

5. MESSAGE: descritto nella rfc 3428 viene utilizzato per inviare un messaggio (tipo SMS) ad una entità SIP.

6. INFO: descritto nella rfc 2976 permette lo scambio di informazioni opzionali lungo un percorso di segnalazione. In particolare può essere utilizzato per trasportare messaggi di segnalazione di tipo PSTN tra gateway PSTN, trasportare segnali DTFM generati durate una sessione SIP, trasportare informazioni relative alla qualità del segnale wireless in supporto al corretto funzionamento di applicazioni mobili, consegnare informazioni sul credito residuo di un particolare account.

Nella tabella 3.3 viene mostrato un esempio di messagio INVITE e per la descrizione dei principali campi header utilizzati in un qualsiasi messaggio SIP si rimanda alla tabella 3.5

Response

I messaggi Response hanno come start-line una status-line costruita come segue:

```
Status-Line  =  SIP-Version SP Status-Code SP Reason-Phrase CRLF
```

Costituita dalla versione del protocollo usata, un numero intero di tre cifre (status-code) ed una frase opzionale a commento della risposta. Lo status-code indica il tipo di risposta contenuta nel messaggio. In base al codice le tipologie di risposta si possono distinguere in sei gruppi:

1xx: Provisional: risposte provvisorie che informano il chiamante che la richiesta inoltrata è stata ricevuta ma non se ne conosce lo stato di avanzamento del suo processamento. Quando viene ricevuto un messaggio provisional il timer di ritrasmissione viene interrotto. Solitamente un proxy server invia una risposta con codice 100 quando inizia a processare una richiesta di tipo INVITE mentre uno User Agent invia una risposta con codice 180 per segnalare che il telefono del chiamato sta squillando.

2xx: Success: risposte che confermano il successo dell'operazione richiesta. Queste risposte informano il mittente della richiesta che quest'ultima è stata processata correttamente e servono anche per teminare una transazione. Ad una richiesta di tipo INVITE andata a buon fine un server risponde con un messaggio con codice 200.

INVITE sip:7170@iptel.org SIP/2.0	Status-Line
Via: SIP/2.0/UDP 195.37.77.100:5040;rport Max-Forwards: 10 **From**: "jiri" <sip:jiri@iptel.org>;tag=76ff7a07-c091-4192-84a0-d56e91fe104f **To**: <sip:jiri@bat.iptel.org> **Call-ID**: d10815e0-bf17-4afa-8412-d9130a793d96@213.20.128.35 **CSeq**: 2 INVITE **Contact**: <sip:213.20.128.35:9315> User-Agent: Windows RTC/1.0 Proxy-Authorization: Digest username="jiri", realm="iptel.org", algorithm="MD5", uri="sip:jiri@bat.iptel.org", nonce="3cef753900000001771328f5ae1b8b7f0d742", response="53fe98db10e1074 b03b3e06438bda70f" **Content-Type**: application/sdp **Content-Length**: 451	Header
	Riga vuota tra Header e Body
v=0 o=jku2 0 0 IN IP4 213.20.128.35 s=session c=IN IP4 213.20.128.35 b=CT:1000 t=0 0 m=audio 54742 RTP/AVP 97 111 112 6 0 8 4 5 3 101 a=rtpmap:97 red/8000 a=rtpmap:111 SIREN/16000 a=fmtp:111 bitrate=16000 a=rtpmap:112 G7221/16000 a=fmtp:112 bitrate=24000 a=rtpmap:6 DVI4/16000 a=rtpmap:0 PCMU/8000 a=rtpmap:4 G723/8000 a=rtpmap: 3 GSM/8000 a=rtpmap:101 telephone-event/8000 a=fmtp:101 0-16	Corpo SDP del messaggio

Tabella 3.3: Messaggio INVITE

3xx: Redirection: richieste di redirezione della richiesta. Una risposta di redirezione contiene le informazioni riguardanti la nuova posizione dell'utente chiamato o riguardanti servizi alternativi che il chiamante può utilizzare per soddisfare la chiamata. Questo tipo di risposte vengo solitamente inviate da un proxy server. Quando quest'ultimo riceve una richieste che per qualche motivo non vuole o non può processare invia una risposta redirection al chiamente includendo in questa l'indirizzo di un altro server al quale il client può inoltrare la richiesta.

4xx: Client Error: avvisi di errore da parte del client nella sintassi della richiesta.

5xx: Server Error: avvisi d'errore da parte del server che non può effettuare l'operazione richiesta

6xx: Global Failure: fallimenti critici dovuti a motivazioni diverse dalle precedenti

Campi header

L'header SIP è simile a quelli dell'HTTP in maniera conforme all'rfc 2234 ed assume la forma:

```
header = "header-name" HCOLON header-value *(COMMA header-value)
```

in cui i campi sono separati da virgole nell'ordine “nome-campo: valore” seguiti da una lista eventualmente vuota di parametri. In tabella 3.5 è presente una lista degli header sip più utilizzati con la rispettiva spiegazione.

Corpo del messaggio

Il corpo del messaggio (Body) dipende strettamente dal tipo di richiesta o risposta. La lunghezza e la tipologia dei contenuti deve essere definita nell'intestazione del messaggio. Può contenere anche dati binari secondo gli standard MIME. Quando non diversamente specificato nell'intestazione del messaggio il set di caratteri è “UTF-8”.

3.2.4 Componenti

Il protocollo SIP definisce le entità coinvolte durante l'instaurazione di una sessione di comunicazione.

Campi header	Descrizione
From	Identità di chi ha avviato una procedura di tipo request.
To	Destinatario del messaggio sip request.
Call-ID	Identifica una serie di messaggi SIP come appartenenti ad un'unica conversazione.
Cseq	Identifica, ordina e dispone in sequenza le richieste sip all'interno di una conversazione. Serve anche a distinguere i messaggi ritrasmessi da quelli nuovi.
Via	Indica il percorso intrapreso dal messaggio di richiesta e identifica la destinazione rispetto alla quale inviare i messaggi response.
Conctat	Identifica il SIP URI rispetto al quale un dispositivo UA vuole ricevere un nuovo messaggio di tipo sip request.
Allow	Riporta i metodi supportati dal dispositivo che sta generando il messaggio.
Supported	Riporta le estensioni SIP supportate dal dispositivo.
Require	Riporta l'estensione SIP relativa al dispositivo UA remoto considerata necessaria per processare la transazione.
Content-Type	Indica la tipologia del corpo del messaggio allegato al messaggio di SIP request o SIP response (es application/sdp per una sessione audio e text/plain per l'invio di un instant messaging).
Content-Length	Indica (in decimale) la dimensione del corpo del messaggio.

Tabella 3.5: Header di un messaggio SIP

User Agent

Lo User Agent è l'entità che attraverso un'interfaccia permette all'utente di stabilire una connessione con un altro terminale SIP. Inoltre rende trasparente l'utilizzo dello stack e si occupa di determinare il tipo del media da utilizzare.

Lo User Agent è costituito dall'unione di due entità logiche:

- UAc (User Agent Client): ovvero l'entità in grado di generare messaggi di richiesta per l'istaurazione di una connessione.
- UAs (User Agent Server): entità in grado di ricevere richieste ed inviare risposte.

Nella norma si intende per UA un'entità unica che svolge entrambe le funzioni sopraelencate.

Redirect Server

Un redirect server è un'entità che aiuta un UA a localizzare la posizione all'interno della rete dell'utente con il quale si desidera iniziare una sessione. A differenza del proxy non può accettare delle chiamate, ma può generare risposte che diano informazioni allo UAc sull'instradamento da effettuare per contattare altre entità. I server redirect permettono agli altri server di aggiungere informazioni sui percorsi nei messagi di risposta ai client e si tirano fuori dal ciclo per il resto della connessione. Infatti il client che ha composto la richiesta una volta ricevuto l'instradamento dal server redirect comporrà i successivi messaggi con l'URI ricevuto. Questo procedimento permette una buona scalabilità anche per reti molto grandi. Nella figura 3.2è mostrato il modo in cui operano due redirect server.

Proxy Server

Server intermedio che può rispondere direttamente alle richieste oppure "ruotarle" ad un client, ad un server o ad un ulteriore proxy. Un proxy server a seconda della configurazione può anche funzionare in modalità biforcazione (fork), ovvero può inoltrare una richiesta di connessione a due o più entità in parallelo consentendo un rintracciamento più veloce degli utenti. I proxy server possono essere raggruppati in tre categorie:

- Call-stateful: memorizza tutte le informazioni sugli stati da quando la sessione viene stabilita finché questa non viene terminata.
- Stateful: memorizza tutti gli stati dall'inizio alla fine di una transaction.

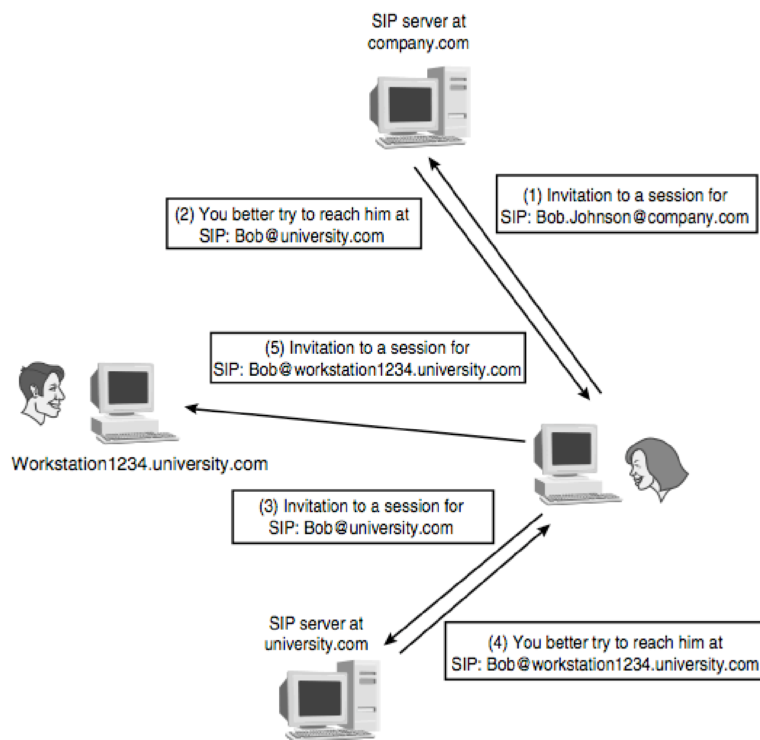


Figura 3.1: Scenario con due server Redirect

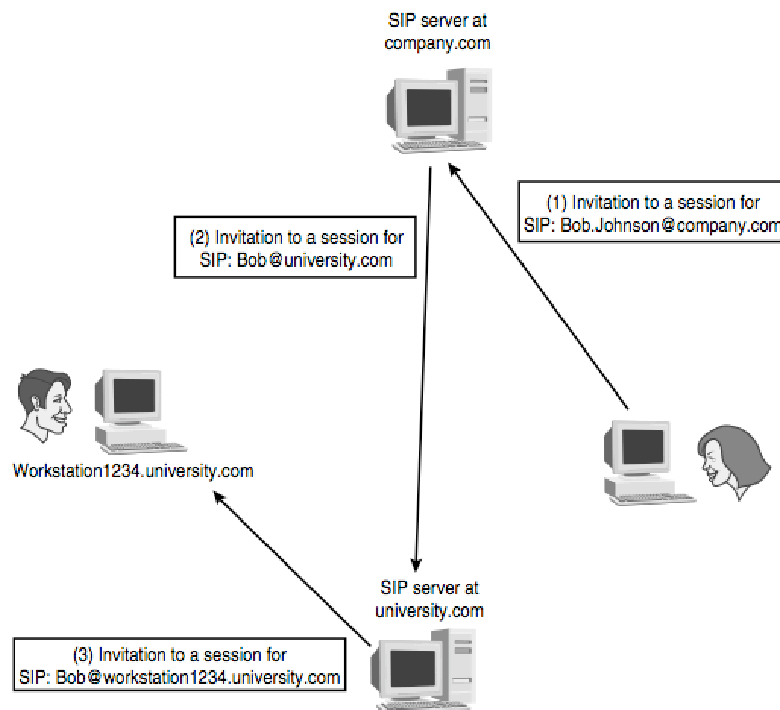


Figura 3.2: Scenario con due proxy server

- Stateless: non memorizza nessuna informazione sullo stato. Viene ricevuta ed inoltrata una richiesta al nodo successivo ed immediatamente vengono cancellate tutte le informazioni riguardanti quella richiesta.

Un proxy server analizza i parametri di instradamento dei messaggi e "nasconde" la reale posizione del destinatario del messaggio - essendo quest'ultimo indirizzabile con un nome convenzionale del dominio di appartenenza. Il vantaggio nel suo utilizzo sta nel fatto che al proxy server può essere delegata in toto la gestione della chiamata, ragion per cui i terminali utente non devono preoccuparsi di tutte le procedure di segnalazione nella loro interezza.

Location Server

Il location server viene richiamato dal redirect server o dal proxy server per ottenere informazioni circa le possibili locazioni dell'utente chiamato. Il funzionamento di questo servizio non viene specificato nella rfc 3261 che ne indica solo alcuni requisiti come la possibilità di accesso in lettura e scrittura da parte di un server Registrar e la formattazione dei dati in modo da essere letti dai

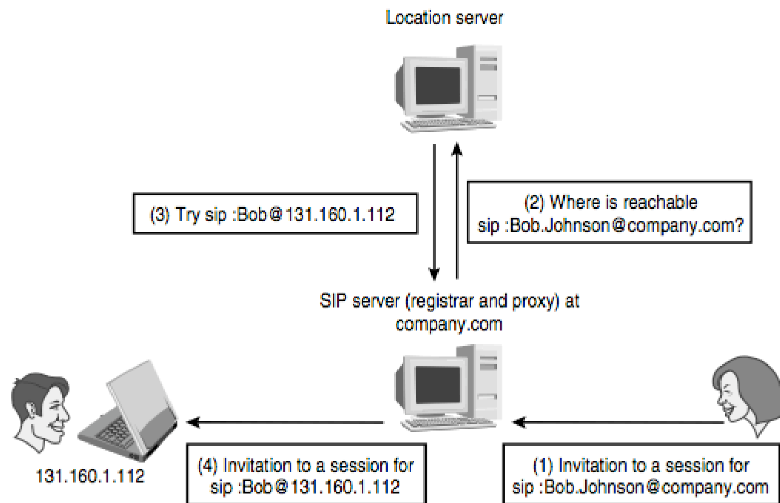


Figura 3.3: Location server

Proxy e dai Redirect server. Il locator server non utilizza protocollo sip per comunicare con il registrar ed il proxy.

In figura 3.3 è mostrato il funzionamento di un location server

Registar Server

Questo componente immagazzina tutte le informazioni necessarie al rintracciamento degli utenti che all'inizio di una connessione sono tenuti a registrare la loro presenza all'interno della rete.

3.3 MJSip

MjSip è una compatta e potente libreria scritta in linguaggio Java che permette di sviluppare facilmente applicazioni e servizi operanti con il Session Initiation Protocol. MjSip include tutte le classi e metodi necessari all'implementazione dello stack Sip descritto nella RFC 3261, in modo da renderlo completamente compatibile con questo standard. Le principali caratteristiche di MjSip sono:

- scritto in Java, può essere utilizzato su diverse piattaforme;
- non sono solo API ma include la completa implementazione dello stack SIP;

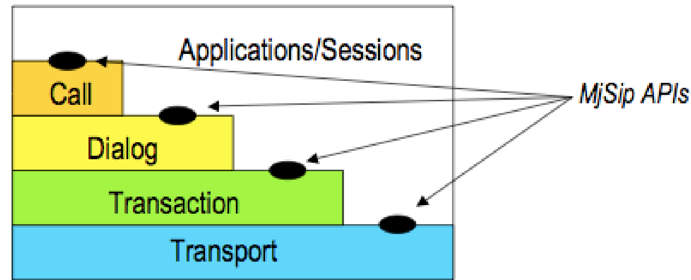


Figura 3.4: Stack di MjSip

- compatibile con lo standard definito nalla RFC 3261;
- semplice da usare e semplice da estendere;
- implementa la stessa interfaccia di basso livello utilizzata da JAIN Sip risultando così più familiare per chi ha già lavorato con quest'ultima;
- aggiunge a queste interfacce di basso livello nuove API per il controllo delle chiamate rendendo così più semplice il suo utilizzo.

3.3.1 Architettura

Secondo quanto definito nella RFC 3261 il core di MjSip è strutturato in tre livelli: Transport, Transaction e Dialog. In cima a questa struttura si colloca il livello Call, contenente le API per il controllo delle chiamate. In figura 3.4 è rappresentato lo stack MjSip nel quale vengono evidenziati i punti di accesso definiti dalle API.

Lo sviluppatore può scegliere da quale livello partire per sviluppare la propria applicazione interfacciandosi con le API messe a disposizione in quel livello e implementando i metodi dell'interfaccia di tipo <levelLayer>Listener per avere notizie su determinate notifiche.

Le classi che permettono l'interazione con i diversi livelli sono:

- **Call** contenuta nel package **org.zoolu.sip.call**;
- **InviteDialog** contenuta nel package **org.zoolu.sip.dialog**;
- **TransactionClient**, **TransactionServer**, **InviteTransactioClient**, **InviteTransactionServer** contenute nel package **org.zoolu.sip.transaction**;

- **sipProvider** contenuta nel package **org.zoolu.sip.provider**.

Le interfacce tra livelli adiacenti si basano su un modello Provider - Listener. Se una classe vuole interagire con un livello inferiore deve estendere la classe LayerListener appartenente a questo livello e aggiungere se stesso alla lista dei possibili listener di eventi generati dal provider del livello inferiore. Gli eventi vengono catturati dalla classe del livello superiore attraverso l'implementazione dello specifico metodo ereditato dalla specifica classe Listener.

Oltre ai pacchetti elencati in precedenza, che di fatto rappresentano il core di uno stack SIP, sono presenti altri pacchetti di supporto che implementano classi utili al corretto funzionamento dell'apparato applicativo. Questi pacchetti sono:

- **org.zoolu.sip.message**: contiene le classi che permettono la costruzione di massaggi SIP utilizzati all'interno di una transazione. La classe **BaseMessageFactory** contiene i metodi per la creazione di messaggi di richiesta standard, ovvero quei messaggi che hanno nel campo satusline un metodo appartenente all'insieme di medodi definiti nella rfc 3261 e nella classe **BaseSipMethods**.

  Se invece si vuole costruire un messaggio con metodo appartenente ad una estensione del protocollo (definiti nella classe **SipMethods**) si fa riferimento alla classe **MessageFactory**. Le funzioni implementati in queste due classi sono nella forma **create<METHOD>Request** dove METHOD rappresenta il metodo utilizzato per la richiesta.

  Per esempio, per un messaggio di tipo MESSAGE si può utilizzare il metodo **createMessageRequest(SipProvider sip_provider, NameAddress recipient, NameAddress from, String subject, String type, String body)** . Invece se si vuole costruire un messaggio di tipo INVITE si chiama il metodo **createInviteRequest(SipProvider sip_provider, SipURL request_uri, NameAddress to, NameAddress from, NameAddress contact, String body)**.

  Per i messaggi di risposta si utilizza il metodo **createResponse(Message req, int code, String reason, String local_tag, NameAddress contact, String content_type, String body)** .

- **org.zoolu.sip.header**: contiene le classi necessarie per la creazione di un header SIP.

- **org.zoolu.sip.authentication**: contiene la classe **DigestAuthentication** che implementa l'HTTP Digest Authentication definito nella rfc 2617 per l'autenticazione dell'applicativo client SIP con il registar server.

- **org.zoolu.sip.address**: contiene le classi **NameAddress** e **SipURL**. La prima viene utilizzata per la rappresentazione di un valido SIP Name Address definito nella forma *[display-name] address* dove address è un valido SIP URL definito dalla classe omonima.

- **org.zoolu.sip.sdp**: contiene tra le altre la classe **SessionDescriptor** che permette la costruzione del corpo di un messaggio secondo le direttive definite dal Session Description Protocol. L'invocazione del suo costruttore **SessionDescriptor(String origin, String session, String connection, String time)** permette di impostare i campi o, s, c, t ovvero i campi che specificano l'identificatore della sessione, il nome della sessione, informazioni sulla connessione, tempo in cui la sessione è attiva. Una volta creato l'SDP viene inizializzato con l'aggiunta del campo "v (protocol version) = 0", con la creazione di un vettore che riporta informazioni riguardanti i media utilizzati nella sessione (campo "m") e con la creazione di un vettore contenente altri possibili campi che possono essere aggiunti all'SDP.

3.3.2 Livello Transport

Il livello più basso è costitiuto dal livello di trasporto. La classe che si occupa di fornire le API necessarie per l'utilizzo di tale livello è la classe *SipProvider*. Quest'oggetto si occupa di rendere disponibile ai livelli superiori un servizio di trasporto, cioè l'invio e la ricezione di messaggi SIP attraverso differenti protocolli e demultiplexare i messaggi ricevuti verso l'appropriata entità di livello superiore. Ogni elemento SIP dovrebbe utilizzare le API messe a disposizione dal SipProvider per avere accesso ai servizi offerti dal livello di trasporto.

Il costruttore di questa classe

```
public SipProvider(String via_addr, int port){
    init(via_addr, port, null, null);
    initlog();
    startTrasport();
}
```

riceve come parametri l'indirizzo ip del server e la porta TCP/UDP da utilizzare per comunicare con quest'ultimo e inizializza l'oggetto SipProviderListener. Quest'ultimo viene definito attraverso l'omonima interfaccia che mette a disposizione il metodo

```
public void onReceivedMessage(SipProvider sip_provider,
   Message message)
```

che viene implementato dalle classi InviteTransactionClient, InviteTransactionServer, TransactionClient, TransactionServer, InviteDialog per gestire l'evento "ricezione di un messaggio". Un SipProviderListener può essere aggiunto a un sipProvider attraverso il metodo **addSipProviderListener(Identifier id, SipProviderListener listener)** dove *id* identifica l'interfaccia SIP interessata alla ricezione dei messaggi mentre *listener* è un'istanza di *sipProviderListener* alla quale vengono passati i messaggi ricevuti. L'identificatore dell'interfaccia SIP specifica il tipo di messaggio che l'ascoltatore si aspetta di ricevere e insieme al SipProvider forma il completo SIP Service Access Point (SAP) utilizzato per il demultiplexing messaggi SIP. L'identificatore può essere di tipo transaction_id, dialog_id o method_id identificanti rispettivamente i messaggi all'interno di una specifica transazione, i messaggi all'interno di una finestra di dialogo e i messaggi relativi a uno determinato metodo SIP. É anche possibile utilizzare l'identificatore ANY per tutti quei messaggi che non appartengono ad una transazione, dialogo, o non contengono un tipo di metodo specificato.

I metodi più importanti che la classe mette a disposizione per le entità definite ai livelli superiori sono:

- **public ConnectionIdentifier sendMessage(Message msg)**: metodo utilizzato per l'invio di un messaggio incapsulato all'interno della struttura Message. Questo metodo ritorna un oggetto di tipo ConnectionIdentifier rappresentante l'identificativo di una connessione. Il suo valore è diverso da **null** nel caso in cui si stia utilizzando un protocollo affidabile come TCP, **null** in caso contrario.

- **public void halt()**: permette l'interruzione del SipProvider e quindi la disconnessione dal server SIP.

3.3.3 Livello Transaction

Al secondo livello è collocato il livello Transaction. Componente fondamentale dello stack SIP gestisce le transazioni, ovvero le richieste inviate da un client (transaction client) verso un transaction server e le risposte che quest'ultimo invia al transaction client. É possibile identificare due tipi di transazioni:

1. **two way transaction**: questo tipo di transazione, che non richiede da parte del client di inviare il riscontro del messaggio di fine transazione inviato da server, viene implementato dalle classi **TransactionClient** e **TransactionServer** che si occupano rispettivamente di creare una nuova transazione SIP partendo con un messaggio di richiesta inviato al SipProvider e di inviare la risposta relativa a una richiesta ricevuta.

2. **three-way transactions**: questo tipo di transazione prevede invece il riscontro della ricezione del messaggio inviato dal server transaction al client transaction. Questo tipo di transazione viene implementata dalle classi **InviteTransactionClient** e **InviteTransactioServer**.

Queste quattro classi ereditano i metodi definiti nella classe **Transaction** che a sua volta implementa l'interfaccia SipProviderListener ereditandone i metodi per la funzione di callback.

Per creare una InviteTransaction si utilizza il costruttore

```
/** Creates a new ClientTransaction */
public InviteTransactionClient(SipProvider sip_provider,
    Message req, TransactionClientListener listener){
     super(sip_provider);
     request = new Message(req);
     init(listener, request.getTransactionId());
}
```

che riceve in ingresso un'istanza di tipo SipProvider, il messaggio di richiesta e un'istanza di tipo TransactionClientListener utilizzata per la notifica al creatore della transazione che ne implementa i metodi, eventi riguardanti l'evoluzione della transazione stessa.

Il metodo

```
/** Initializes timeouts and listener. */
void init(TransactionClientListener listener,
    TransactionIdentifier transaction_id) {
     this.transaction_listener = listener;
     this.transaction_id = transaction_id;
     this.ack = null;
     retransmission_to = new Timer(SipStack.
        retransmission_timeout, "Retransmission", this);
     transaction_to = new Timer(SipStack.transaction_timeout,
         "Transaction", this);
     end_to = new Timer(SipStack.transaction_timeout, "End",
        this);
     [...]
}
```

inizializza il listener, l'identificatore della transazione, il timer per la ritrasmissione dei messaggi per i quali non si riceve un riscontro (impostato a 500 ms come suggerito dalla rfc 3261), il timer relativo alla durata massima di una transazione impostato a 32 s e infine il timer end_to, anch'esso impostato a 32 s, attivato quando la transazione si trova nello stato COM-

PLETED indica il tempo massimo concesso alla transazione per passare dallo stato COMPLETED allo stato TERMINATED.

La transazione viene attivata con il metodo

```
/** Starts the InviteTransactionClient and sends the invite
    request. */
public void request() {
    printLog("start", LogLevel.LOW);
    changeStatus(STATE_TRYING);
    retransmission_to.start();
    transaction_to.start();
    sip_provider.addSipProviderListener(transaction_id, this
        );
    connection_id = sip_provider.sendMessage(request);
}
```

che imposta lo stato della transazione a TRYING, attiva i time-out di ritrasmissione e transazione, aggiunge un ascoltatore per la ricezione dei messaggi e infine viene inviato il messaggio di request.

3.3.4 Livello Dialog

La classe principale di questo livello è la classe **InviteDialog** utilizzata per la gestione di differenti transazioni all'interno della stessa sessione. Un dialogo è una relazione peer-to-peer tra due user agent che persiste per qualche tempo. La classe **InviteDialog** può fungere sia da client che da server inviando o rispondendo a richieste di tipo INVITE. All'interno di questo livello è presente l'interfaccia **InviteDialogListener** che mette a disposizione metodi di callback per notificare eventi al livello superiore.

Il costruttore della classe **InviteDialog**

```
/** Creates a new InviteDialog. */
public InviteDialog(SipProvider sip_provider,
    InviteDialogListener listener) {
     super(sip_provider);
     init(listener);
}
```

riceve come parametri un'istanza di tipo SipProvider e InviteDialogListener. La prima, passata a sua volta alla superclasse **Dialog** serve a istanziare il livello di trasporto mentre la seconda viene utilizzata per l'interazione con la classe che ne implementa i metodi di callback.

Il metodo **init(InviteDialogListener listener)** viene utilizzato per il completamento della creazione di un dialogo.

Per invitare un utente a partecipare ad una sessione viene utilizzato il metodo

```
/* Starts a new InviteTransactionClient and initializes the
   dialog state */
public void invite(String callee, String caller, String
   contact, String session_descriptor) {
    [...]
    NameAddress to_url = new NameAddress(callee);
    NameAddress from_url = new NameAddress(caller);
    SipURL request_uri = to_url.getAddress();
    NameAddress contact_url = null;
    if (contact != null) {
        if (contact.indexOf("sip:") >= 0)
            contact_url = new NameAddress(contact);
        else
            contact_url = new NameAddress(new SipURL(contact
               , sip_provider.getViaAddress(), sip_provider.
               getPort()));
        }else
            contact_url = from_url;
        Message invite = MessageFactory.createInviteRequest(
           sip_provider, request_uri, to_url, from_url,
           contact_url, session_descriptor);
    // do invite
    invite(invite);
}
```

che inizializza le variabili to_url (NameAddress del chiamato), from_url (NameAddress del chiamante), request_uri (URI del chiamato), contact_url e le utilizza per la creazione di un messaggio di tipo INVITE.

Il metodo

```
public void invite(Message invite) {
    [...]
    invite_req = invite;
    InviteTransactionClient invite_tc = new
       InviteTransactionClient(sip_provider, invite_req,
       this);
    invite_tc.request();
}
```

prende in ingresso il messaggio creato in precedenza, inizializza un'istanza di tipo InviteTransactionClient e fa partire la transazione chiamando su di essa il metodo request().

Se invece ci si vuole mettere in ascolto in attesa della ricezione di un possibile messaggio INVITE inviato da un'entità SIP remota si utilizza il metodo

```
/** Starts a new InviteTransactionServer. */
public void listen() {
    if (!statusIs(D_INIT))
        return;
    changeStatus(D_WAITING);
    invite_ts = new InviteTransactionServer(sip_provider,
       this);
    invite_ts.listen();
}
```

che crea un'istanza della classe InviteTransactionServer e l'abilita chiamando su di essa il metodo listen().

3.3.5 Livello Call

All'ultimo livello dello stack MjSip si trova lo strato Call, descritto dal package Call, che mette a disposizione dello sviluppatore una serie di API per il controllo di una chiamata SIP. La classe più importante è la classe Call che implementa l'interfaccia InviteDialogListener ereditandone i metodi di callback. Tra i metodi più importanti che la classe mette a disposizione segnaliamo i seguenti:

- **call()** che permette di iniziare una nuova chiamata invitando un utente remoto;
- **listen()** che si preoccupa della ricezione di eventuali chiamate;
- **hangup()** che termina una chiamata;
- **accept()** per accettare una chiamata in ingresso.

All'interno del package Call si colloca l'interfaccia **CallListener** che mette a disposizione i seguenti metodi di callback per essere implementati dalla classe che definisce lo User Agent:

- **void onCallIncoming** chiamato quando viene ricevuto un messaggio di tipo INVITE;
- **void onCallModifying** chiamato quando viene ricevuto un messaggio di tipo RE-INVITE;

- **void onCallRinging** chiamato quando viene segnalata la ricezione di un messaggio 180 ringing;
- **void onCallAccepted** chiamato quando viene ricevuto un messaggio di tipo 2xx ovvero la segnalazione che la chiamata è stata accettata;
- **void onCallRefused** chiamato quando viene ricevuto un messaggio 4xx indicante che la chiamata è stata rifiutata;
- **void onCallRedirection** chiamato quando viene ricevuto un messaggio 3xx indicante la redirezione della chiamata;
- **void onCallConfirmed** chiamato quando viene ricevuto un ACK confermante una chiamata;
- **void onCallTimeout** chiamato scade il timeout per una richiesta di tipo INVITE;
- **void onCallReInviteAccepted** chiamato quando arriva un messaggio 2xx indicante l'accetazione di una chiamata re-invite o modify;
- **void onCallReInviteRefused** chiamato quando arriva un messaggio 4xx indicate il fallimento dei metodi re-invite/modify;
- **void onCallReInviteTimeout** chiamato quando scade il timeout per una richiesta re-invite;
- **void onCallCanceling** chiamato quando arriva una richiesta di tipo CANCEL;
- **void onCallClosing** chiamato quando arriva una richiesta di tipo BYE;
- **void onCallClosed** chiamato quando arriva una risposta ad una richiesta di tipo BYE. Chiude la connessione.

Un oggetto di tipo Call viene creato invocando il suo costruttore

```
public class Call implements InviteDialogListener {
[...]
/** Creates a new Call. */
    public Call(SipProvider sip_provider, String from_url,
       String contact_url, CallListener call_listener) {
        this.sip_provider = sip_provider;
        this.log = sip_provider.getLog();
        this.listener = call_listener;
        this.from_url = from_url;
```

```
        this.contact_url = contact_url;
        this.dialog = null;
        this.local_sdp = null;
        this.remote_sdp = null;
    }
[...]
}
```

al quale si passa un oggetto di tipo SipProvider per la gestione delle comunicazioni a livello di trasporto, una stringa *from_url* indicante l'URL dell'entità chiamante e una stringa *contact_url* indicante l'URL dell'entità che si desidera contattare. Infine gli si passa un oggetto di tipo CallListener utilizzato per la notifica di eventi all'istanza della classe che ne implementa i metodi.

Un utente remoto può essere inivitato a iniziare una chiamata attraverso il metodo

```
/** Starts a new call, inviting a remote user */
public void call(String callee, String from, String contact,
    String sdp) {
    printLog("calling " + callee, LogLevel.HIGH);
    if (from == null)
        from = from_url;
    if (contact == null)
        contact = contact_url;
    if (sdp != null)
        local_sdp = sdp;
    dialog = new InviteDialog(sip_provider, this);
    if (local_sdp != null)
        dialog.invite(callee, from, contact, local_sdp);
    else
        dialog.inviteWithoutOffer(callee, from, contact);
    }
```

Questo metodo inizializza le variabili **from**, **contact** e **local_sdp**, crea un'istanza dell'oggetto **InviteDialog** passandogli come parametri il *sip_provider* e un'istanza della classe Call. In funzione del contenuto della variabile *local_sdp* viene deciso se iniziare un dialogo con un'iniziale offerta di codec oppure no.

Il metodo

```
/** Waits for an incoming call */
public void listen() {
    dialog = new InviteDialog(sip_provider, this);
    dialog.listen();
}
```

permette all'utente di stare in uno stato di ascolto in attesa di ricevere una chiamata. Nel caso in cui un utente remoto stia cercando di invitarci a partecipare ad una conversazione possiamo decidere se accettare la chiamata invocando il metodo

```
/** Accepts the incoming call */
public void accept(String sdp) {
    local_sdp = sdp;
    if (dialog != null)
        dialog.accept(contact_url, local_sdp);
}
```

al quale si passa il tipo di codec che si vuole utilizzare nel corso della chiamata. L'accettazione della chiamata viene infine passata all'oggetto di tipo InviteDialog inizializzato nel metodo **listen()**.

Il metodo

```
/** Cancels the outgoing call */
public void cancel() {
        if (dialog != null)
            dialog.cancel();
    }
```

viene utilizzato per terminare una chiamata inizializzata dall'utente locale. In questo caso l'istanza della classe **InviteDialog** viene creata all'interno del metodo **call**.

Capitolo 4

SipDroid

4.1 Introduzione

Sipdroid è un client SIP open source fornito sotto licenza GPL. Disponibile solo per la piattaforma Android permette di effettuare chiamate su rete WIFI, 3G ed EDGE con un'ottima qualità audio e una bassa latenza. Allo stato attuale è giunto alla versione 1.6 ed è pienamente compatibile con la totalità dei dispositivi Android 1.6, 2.1, 2.2. Con l'uso della libreria jSTUN viene garantito il suo funzionamento attraverso le varie strutture NAT preesistenti. Dal punto di vista multimediale il client Sipdroid mette a disposizione diversi tipi di codec audio garantendo all'utente un buon rapporto di qualità/compressione ed una corretta connessione con la maggior parte dei server SIP.

I codec audio supportati sono:

- U-LAW e A-LAW: sono derivati rispettivamente dai codec T1 (utilizzato in Nord America e in Giappone) e E1 (utilizzato nel resto del mondo) di tipo G.711 PCM appartenenti allo standard ITU. T1 ed E1 vengono utilizzati nella telefonia tradizionale e permettono di codificare segnali audio con teorica banda massima pari a 4000 Hz. Utilizzati con la tecnologia voip permettono di codificare segnali vocali con un'ottima qualità poichè non viene utilizzato alcun tipo di compressione rendendo il suono prodotto molto simile a quello di un regolare telefono. La mancanza di compressione (quindi l'assenza di un audio processing iniziale in fase di trasmissione) permette ai pacchetti di essere consegnati con una bassa latenza. D'altro canto, richiede un bit rate mediamente superiore agli 84 Kbps.

- BV16: codec ottimizzati per la trasmissione di segnali vocali sulla rete IP. La progettazione di questo codec ha avuto come obiettivo quello

di rendere il ritardo dovuto alla codifica della trasmissione del segnale vocale il più basso possibile e di mantenere una buona qualità audio. Inoltre le richieste computazionali sono ridotte dalla bassa complessità dell'algoritmo di codifica e decodifica.

- G722: supportano un bit rate di 64, 56 e 48 kbps, possono essere integrati su un chip permettendo un ritardo complessivo dovuto all'elaborazione di all'incirca 3 ms, ritardo abbastanza piccolo da non causare problemi di echo. Questi codec offrono prestazioni accettabili per tassi di errore sui bit trasmessi fino a 10^3. Questo requisito assicura un lieve degrado delle prestazioni anche nelle condizioni peggiori di trasmissione che possono verificarsi all'interno di una rete.

- GSM: i codec di tipo GSM full rate operano ad un rate di 13 kbits/s e fanno uso di codec Regular Pulse Excited (RPE). Codificano un segnale vocale con una buona qualità sebbene sia inferiore a quella prodotta da codec che utilizzano un rate più alto come G728. Il principale vantaggio dell'utilizzo di codec con un basso rate è la relativa semplicità di implementazione che permette loro di lavorare in real time su sistemi con basse prestazioni.

- SILK: codec proprietari della società Skype, distribuiti con una licenza royalty free permettono di codificare una sorgente audio con un'ottima qualità, si adattano alle condizioni di traffico della rete in tempo reale scalando la larghezza di banda del segnale generato.

- SPEEX: codec che si adattano in modo ottimale per le applicazioni internet, possiedono caratteristiche non presenti in molti altri codec come la codifica stereo, l'integrazione di più frequenze di campionamento all'interno dello stesso stream di byte e dispone della modalità VBR (variable bit rate). D'altro canto richiedono risorse hardware superiori rispetto ad altri codec.

La connessione al server SIP prescelto e il processo di autenticazione e signaling con quest'ultimo viene effettuata utilizzando la libreria MjSip, introdotta nel capitolo precedente. Gli sviluppatori di Sipdroid, a partire dalla libreria MjSip, ha sviluppato tutta la parte relativa al trasporto dei media, supportando sia il protocollo TCP che UDP, gestendo il processo di acquisizione e codifica audio così come la ricezione, la decodifica e la riproduzione dello stesso, garantendo il pieno supporto allo scambio real-time di contenuti multimediali. SipDroid supporta inoltre la videochiamata permettendo all'utente in possesso di un disposivo in grado di effettuare un'acquisizione video di inoltrare la stessa ad un utente remoto.

4.2 Struttura

Sipdroid contiene i seguenti pacchetti:

- **com.jtsun.core.attribute**, **com.jstun.core.header**, **com.core.util**, **com.jtsun.demo**. Fanno parte della libreria open source JSTUN, distribuita sotto le licenze GPL e Apache 2.0, permettono alle applicazioni che ne fanno uso di rilevare la presenza e riconoscere il tipo di firewall o struttura NAT che si interpone tra l'applicazione e la rete pubblica. In presenza di un NAT lo STUN può essere utilizzato per carpire l'indirizzo IP assegnato al NAT.

- **org.sipdroid.codecs**: contiene le classi necessarie per l'utilizzo dei codec. Alcune di queste classi rappresentano dei "wrapper" perchè non implementato direttamente gli algoritmi di codifica/decodifica del segnale vocale ma incapsulano le librerie scritte in linguaggio nativo che vengono caricate con l'utilizzo del metodo *System.loadLibrary("<libraryName_jni>")*. Queste librerie vengono compilate come "JNI shared library" con l'ausilio dell'NDK (Native Development Kit) che permette quindi allo sviluppatore di inserire all'interno della propria applicazione algoritmi implementati in linguaggio nativo.

- **org.sipdroid.media**: fornisce gli strumenti necessari alla riproduzione/registrazione del segnale vocale e alla ricezione/invio dello stream audio gestito attraverso il protocollo RTP. É costituito dalle seguenti classi:

  - **MediaLauncher**: interfaccia che definisce i metodi per la gestione del contenuto multimediale.
  - **JAudioLauncher**: Classe che implementa l'interfaccia MediaLauncher e che si occupa della gestione di contenuti multimediali audio attraverso l'uso combinato di oggetti di tipo RtpStreamReceiver/RtpStreamSender.
  - **RtpStreamReceiver**: Classe che si occupa della ricezione di pacchetti RTP opportunamente decapsulati e della successiva decodifica attraverso il codec selezionato nel modulo SIP al fine di avviare la riproduzione del contenuto multimediale audio attraverso le API dell'ambiente Android.
  - **RtpStreamSender**: Classe che si occupa dell'acquisizione del flusso audio da una periferica di ingresso (microfono) e della successiva

codifica attraverso il codec selezionato nel modulo SIP al fine di inviare tale flusso codificato opportunamente incapsulato al modulo RTP sottostante.

- **org.sipdroid.net**: contiene le classi di supporto all'operazione di invio/ricezione del flusso multimediale. Le classi più importanti sono:
  - **RtpSocket**: ha il compito di decapsulare la struttura dati di un pacchetto RTP, incapsularlo in un pacchetto UDP (User Datagram Protocol) e inviarlo tramite la socket di tipo SipdroidSocket o viceversa.
  - **RtpPacket**: classe che ha il compito di incapsulare/decapsulare i dati da inviare/ricevere attraverso il protocollo RTP.
  - **SipdroidSocket**: si occupa del processo di invio e ricezione di pacchetti tramite protocollo di trasmissione UDP.
- **org.sipdroid.sipua**: contiene le classi rappresentati lo User Agent e il Register Agent più ulteriori classi di supporto come il SipdroidEngine.
- **org.sipdroid.sipua.ui**: contiene tutte le classi necessarie per la costruzione dell'interfaccia grafica come Sipdroid più altre classi per la gestione di eventi interni al sistema come la classe Receiver.

4.2.1 UserAgent

Rappresenta una delle classi più importanti all'interno dell'applicazione Sipdroid in quanto implementa il comportamento dell'entità SIP User Agent.

Questa classe estende la classe **CallListenerAdapter** appartenente al package **org.zoolu.sip.call** che a sua volta implementa l'interfaccia **CallListener** contenente i metodi di callback necessari per la comunicazione di eventi dallo stack sip al livello applicativo.

```
public class UserAgent extends CallListenerAdapter {
    [...]
    public static final int UA_STATE_IDLE = 0;
    public static final int UA_STATE_INCOMING_CALL = 1;
    public static final int UA_STATE_OUTGOING_CALL = 2;
    public static final int UA_STATE_INCALL = 3;
    public static final int UA_STATE_HOLD = 4;
    [...]
}
```

Lo User Agent è stato implementato in modo da dover gestire quattro stati all'interno di una sessione SIP:

- UA_STATE_IDLE: in questo stato lo User Agent è in attesa dell'inizializzazione di una sessione da parte dell'utente locale o di un utente remoto.
- UA_STATE_INCOMING_CALL: lo User Agent ha ricevuto una richiesta di INVITE da parte di un utente remoto. In questo caso funge da server.
- UA_STATE_OUTGOING_CALL: in questo stato l'utente locale si appresta a inizializzare una sessione. In questo caso funge da client.
- UA_STATE_INCALL: la richiesta di INVITE inoltrata o ricevuta è stata accettata. In questo caso la sessione risulta essere inizializzata ed è quindi possibile iniziare lo scambio di contenuti multimediali.
- UA_STATE_HOLD: lo scambio dei contenuti multimediali viene interrotto ma la sessione inizializzata non viene cancellata.

Il cambiamento di uno stato viene inoltrato alla classe Receiver attraverso il metodo di tipo synchronized

```
/** Changes the call state */
protected synchronized void changeStatus(int state,String
    caller) {
     call_state = state;
     Receiver.onState(state, caller);
}
```

dove l'argomento *caller* rappresenta l'URI dell'entità che si vuole contattare nel caso di un OUTGOING_CALL o l'URI dell'entità chiamante nel caso di un INCOMING_CALL. Per gli altri stati il valore viene posto a **null**.

Uno User Agent viene creato invocando il suo costruttore

```
/** Costructs a UA with a default media port */
public UserAgent(SipProvider sip_provider, UserAgentProfile
    user_profile) {
     this.sip_provider = sip_provider;
     log = sip_provider.getLog();
     this.user_profile = user_profile;
     realm = user_profile.realm;
     // if no contact_url and/or from_url has been set,
         create it now
     user_profile.initContactAddress(sip_provider);
}
```

che inizializza il sipProvider e le informazioni riguardanti il profilo dell'utente, le stesse che vengono utilizzate in fase di registrazione dello stesso presso un registar server.

Una volta creato è possibile utilizzare lo User Agent per iniziare una sessione. Nel caso in cui si voglia contattare un utente remoto si utilizza il metodo

```
public boolean call(String target_url, boolean mode) {
    [...]
    changeStatus(UA_STATE_OUTGOING_CALL, target_url);
    String from_url=null;
    if (mode){
        from_url = user_profile.from_url;
    }else{
    from_url = "sip:anonymous@anonymous.com";
    }
    //change start multi codecs
    createOffer();
    call = new ExtendedCall(sip_provider, from_url,
        user_profile.contact_url, user_profile.username,
        user_profile.realm, user_profile.passwd, this);
    [...]
    if (user_profile.no_offer){
        call.call(target_url);
    }else{
        call.call(target_url, local_session);
        }
    return true;
}
```

che riceve in ingresso l'URL dell'utente che si vuole contattare e la modalità con cui si vuole iniziare la sessione: **false** per la modalità anonima e **true** per la modalità visibile. Con la chiamata di questo metodo si transita dallo stato IDLE allo stato OUTGOING_CALL, viene creato un oggetto di tipo ExtendedCall sul quale viene chiamato il metodo call per l'inizializzazione dell'invite dialog. SipDroid utilizza la classe ExtendedCall invece della classe Call illustrata nel capitolo precedente perchè al contrario di quest'ultima supporta i metodi REFER/NOTIFY.

Se si vuole poter ricevere una chiamata è necessario invocare il metodo

```
/** Waits for an incoming call (acting as UAS). */
public boolean listen() {
    [..]
```

```
    call = new ExtendedCall(sip_provider, user_profile.
        from_url, user_profile.contact_url, user_profile.
        username, user_profile.realm, user_profile.passwd,
        this);
    call.listen();
    return true;
}
```

Come avviene per il medodo **call()**, viene creato un oggetto di tipo **ExtendedCall** sul quale però viene invocato il metodo *listen()* che permette la creazione di un invite dialog che a sua volta dovrà inizializzare un oggetto di tipo transaction server.

Il metodo

```
/** Closes an ongoing, incoming, or pending call */
public void hangup(){
    printLog("HANGUP");
    closeMediaApplication();
        if (call != null){
            call.hangup();
        }
    changeStatus(UA_STATE_IDLE);
}
```

viene utilizzato per chiudere una sessione precedentemente avviata. In particolare il metodo interrompe il flusso multimediale agendo sull'oggetto di tipo **MediaLauncher** e riporta lo User Agent sullo stato IDLE.

Per quanto riguarda i metodi di callback citiamo

```
/** Callback function called when arriving a 2xx (call
    accepted)
*/
public void onCallAccepted(Call call, String sdp, Message
    resp){
     [...]
    changeStatus(UA_STATE_INCALL);
     [...]
    launchMediaApplication();
     [...]
    }
}
```

che viene invocato nel momento in cui un invito precedentemente inoltrato viene accettato dall'utente remoto. Questo metodo porta lo User Agent nello stato INCALL e attiva l'oggetto di tipo **MediaLauncher** per poter iniziare lo scambio di contenuti multimediali con l'utente remoto.

4.2.2 SipdroidEngine

La classe SipdroidEngine si pone come interfaccia tra il livello applicativo di Sipdroid e lo stack SIP. Infatti tutte le funzioni che possono essere chimate sullo User Agent o sul Register Agent passano attraverso questa classe. Il metodo più importante è

```
public boolean StartEngine(){
    [...]
    user_profile = new UserAgentProfile(null);
    user_profile.username = PreferenceManager.
       getDefaultSharedPreferences(getUIContext()).getString
       (Settings.PREF_USERNAME, Settings.DEFAULT_USERNAME);
    //modified
    user_profile.passwd = PreferenceManager.
       getDefaultSharedPreferences(getUIContext()).getString
       (Settings.PREF_PASSWORD, Settings.DEFAULT_PASSWORD);
    if(PreferenceManager.getDefaultSharedPreferences(
       getUIContext()).getString(Settings.PREF_DOMAIN,
       Settings.DEFAULT_DOMAIN).length() == 0){
         user_profile.realm = PreferenceManager.
            getDefaultSharedPreferences(getUIContext()).
            getString(Settings.PREF_SERVER, Settings.
            DEFAULT_SERVER);
    }else{
         user_profile.realm = PreferenceManager.
            getDefaultSharedPreferences(getUIContext()).
            getString(Settings.PREF_DOMAIN, Settings.
            DEFAULT_DOMAIN);
    }
    [...]
    IpAddress.setLocalIpAddress();
    sip_provider = new SipProvider(IpAddress.localIpAddress,
         0);
    user_profile.contact_url = user_profile.username + "@" +
         IpAddress.localIpAddress + (sip_provider.getPort()
       != 0?":"+sip_provider.getPort():"") + ";transport=" +
         sip_provider.getDefaultTransport();
    if (PreferenceManager.getDefaultSharedPreferences(
       getUIContext()).getString(Settings.PREF_FROMUSER,
       Settings.DEFAULT_FROMUSER).length() == 0){
    user_profile.from_url = user_profile.username + "@" +
       user_profile.realm;
    }else{
```

```
        user_profile.from_url = PreferenceManager.
            getDefaultSharedPreferences(getUIContext()).
            getString(Settings.PREF_FROMUSER, Settings.
            DEFAULT_FROMUSER) + "@" + user_profile.realm;
    }
    CheckEngine();
    ua = new UserAgent(sip_provider, user_profile);
    ra = new RegisterAgent(sip_provider, user_profile.
        from_url, user_profile.contact_url, user_profile.
        username, user_profile.realm, user_profile.passwd,
        this, user_profile);
    ka = new KeepAliveSip(sip_provider,100000);
    register();
    listen();
    }catch (Exception E){}
    return true;
}
```

Esso ha il compito di costruire il profilo dell'utente partendo dai dati che lo stesso ha inserito all'interno del menu Settings e di utilizzarlo per creare uno User Agent, un Register Agent e un KeepAliveSip, ovvero quella classe che ha il compito di mantenere viva una connessione verso un'entità SIP. Infine viene avviata la procedura di registrazione verso il server SIP prescelto e viene chiamata la funzione **listen()** per informare lo User Agent che il sistema è pronto a ricevere richieste INVITE da parte di utenti remoti.

I metodi **public boolean call(String target_url,boolean force)** e **public void listen()** richiamano i metodi omonimi implementati all'interno dello User Agent mentre **public void rejectcall()** fa riferimento al metodo **public void hangup()**.

4.2.3 Receiver

La classe Receiver estende la classe **BroadcastReceiver** per poter ricevere notifiche relative ad eventi di sistema e quindi avere la possibilita di poterli gestire. La gestione avviene attraverso la sovrascrittura del metodo

```
public void onReceive(Context context, Intent intent) {
    String intentAction = intent.getAction();
    [...]
    if (intentAction.equals(Intent.ACTION_BOOT_COMPLETED)){
        on_vpn(false);
        engine(context).register();
    } else if (intentAction.equals(ConnectivityManager.
        CONNECTIVITY_ACTION)) {
```

```
        engine(context).register();
    [...]
}
```

Nel frammento di codice illustrato è possibile vedere come la classe Receiver utilizzi eventi come l'avvio di sistema o cambiamenti nello stato della connessione per avviare la procedura di registrazione presso un server SIP. Questo permette di automatizzare determinate operazioni e rendere quindi più semplice l'utilizzo dell'applicazione.

Mediante il metodo

```
public static synchronized SipdroidEngine engine(Context
    context) {
    mContext = context;
    if (mSipdroidEngine == null) {
        mSipdroidEngine = new SipdroidEngine();
        mSipdroidEngine.StartEngine();
    }
    [...]
    return mSipdroidEngine;
}
```

viene creata un'istanza della classe **SipdroiEngine** sulla quale verrà chiamato il metodo **StartEngine** per l'esecuzione delle operazioni descritte nella sottosezione precedente.

Attraverso il metodo

```
public static boolean isFast() {
    is_fast = isFastWifi();
    if (!is_fast)
        is_fast = isFastGSM();
    return is_fast;
}
```

la classe Receiver riesce a determinare la presenza di un interfaccia che può essere utilizzata dal dispositivo per connettersi alla rete (WiFi o GSM).

Il metodo

```
public static void onState(int state,String caller) {
    [...]
    switch(call_state){
    case UserAgent.UA_STATE_INCOMING_CALL:
    [...]
    moveTop();
    if (wl == null) {
```

```
            PowerManager pm = (PowerManager) mContext.
                getSystemService(Context.POWER_SERVICE);
            wl = pm.newWakeLock(PowerManager.
                SCREEN_BRIGHT_WAKE_LOCK | PowerManager.
                ACQUIRE_CAUSES_WAKEUP, "Sipdroid.onState");
        }
        wl.acquire();
        Checkin.checkin(true);
        break;
        case UserAgent.UA_STATE_OUTGOING_CALL:
        [...]
        engine(mContext).register();
        [...]
        moveTop();
        break;
        case UserAgent.UA_STATE_IDLE:
        [...]
        if (wl != null && wl.isHeld())
            wl.release();
        mContext.startActivity(createIntent(InCallScreen.class))
            ;
        [...]
        engine(mContext).listen();
        break;
        case UserAgent.UA_STATE_INCALL:
        [...]
        progress();
        [...]
        if (wl != null && wl.isHeld())
            wl.release();
        mContext.startActivity(createIntent(InCallScreen.class))
            ;
        break;
        case UserAgent.UA_STATE_HOLD:
        [...]
        mContext.startActivity(createIntent(InCallScreen.class))
            ;
        break;
        }
        [...]
        }
}
```

viene chiamato ogni qualvolta lo User Agent cambia di stato. Infatti questo metodo serve a gestire i cambiamenti dell'interfaccia grafica associati ad uno specifico evento. Le classi principali adibite al mantenimento dell'interfaccia sono Sipdroid e InCallScreen. Quest'ultima viene creata e lanciata attraverso la funzione **startActivity(createIntent(InCallScreen.class))**.

All'interno del metodo **onState()** viene gestito il power management di sistema attraverso l'oggetto di tipo **PowerManager** che viene creato quando lo User Agent si trova nello stato INCOMING_CALL. Il wakeLock associato al PowerManager viene impostato in modo tale da mantenere la luminosità dello schermo brillante. Il tutto serve a segnalare visivamente all'utente che è in arrivo una telefonata. Il wakeLock viene infine rilasciato quando la telefonata viene terminata con il conseguente spostamento dello User Agent nello stato IDLE. É da notare come il sistema cerchi di registrare l'entità SIP ogni qual volta si decida di attivare una telefonata.

Infine il metodo **progress()** viene utilizzato per determinare l'uscita audio che può essere attraverso le casse di sistema (modalità vivavoce) o attraverso l'altoparlante standard utilizzato del telefono per le chiamate.

4.2.4 Sipdroid

La classe Sipdroid rappresenta e gestisce l'interfaccia iniziale dell'applicazione, infatti è l'attività che per prima viene lanciata dal sistema quando si avvia il programma. Questa è definita nel file *AndroidManifest.xml* nel seguente modo:

```
<activity android:name=".ui.Sipdroid"
    android:label="@string/app_name
    android:launchMode="singleInstance"
    android:configChanges="orientation|keyboardHidden">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.
            LAUNCHER" />
    </intent-filter>
</activity>
```

dove nell'intent-filter viene evidenziato quanto detto sopra.

Sipdroid estende la classe **Activity** e ne eredita i metodi di callback che dovranno essere sovrascritti per una corretta e personalizzata gestione del ciclo di vita dell'attività stessa.

Nel momento in cui l'applicazione viene lanciata viene eseguito il metodo

```
@Override
public void onStart() {
```

```
    super.onStart();
    if (!Receiver.engine(this).isRegistered())
        Receiver.engine(this).register();
    [...]
}
```

che si preoccupa di determinare lo stato della registrazione dell'applicazione presso un server SIP e di avviarne la procedura nel caso in cui questa non sia stata precedentemente effettuata.

Il metodo seguente viene invocato dal sistema quando l'activity viene eseguita per la prima volta all'interno del task.

```
@Override
public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    requestWindowFeature(Window.FEATURE_NO_TITLE);
    setContentView(R.layout.sipdroid);
```

Ha il compito di collegare l'activity all'interfaccia grafica contenuta all'interno della directory `res/layout` e identificata dal parametro *R.layout.sipdroid*. *R* è una classe che viene generata automaticamente dal sistema in fase di compilazione e che contiene la mappatura delle risorse in riferimenti gestibili da Android.

Inoltre viene completata la costruzione dell'interfaccia con il metodo textbf-findViewById che collega una vista con la quale un utente può interagire ad una variabile di sistema. Nel nostro caso *sip_uri_box* rappresenta una casella di testo editabile dall'utilizzatore dell'applicazione. A questa variabile viene aggiunto un listener che viene attivato nel momento in cui l'utente interagisce con la casella di testo e vengono eseguite le operazioni elencate all'interno del metodo **onKey**.

```
sip_uri_box = (AutoCompleteTextView) findViewById(R.id.
    txt_callee);
sip_uri_box.setOnKeyListener(new OnKeyListener() {
public boolean onKey(View v, int keyCode, KeyEvent event) {
    if (event.getAction() == KeyEvent.ACTION_DOWN && keyCode
        == KeyEvent.KEYCODE_ENTER) {
        call_menu();
        return true;
    }
    return false;
    }
});
```

Le variabili *callButton* e *contactsButton* mostrate in seguito gestiscono rispettivamente il tasto che permette di attivare una chiamata e il tasto che permette di accedere alla lista dei contatti presenti all'interno del telefono. Questa lista non è nativa dell'applicazione e viene lanciata attraverso l'esecuzione di un intent che viene passato al metodo **startActivity(myIntent)** per la visualizzazione a schermo dell'activity associata.

```
Button callButton = (Button) findViewById(R.id.call_button);
callButton.setOnClickListener(new Button.OnClickListener() {
    public void onClick(View v) {
        call_menu();
    }
});
Button contactsButton = (Button) findViewById(R.id.
   contacts_button);
contactsButton.setOnClickListener(new Button.OnClickListener
   () {
    public void onClick(View v) {
        Intent myIntent = new Intent(Intent.ACTION_VIEW,
           People.CONTENT_URI);
            startActivity(myIntent);
        }
});
[...]
}
```

L'interfaccia di Sipdroid è mostrata in figura 4.1 dove è possibile notare i due bottoni per la chiamata e la lista dei contatti e la casella di testo dove di norma va inserito l'URI dell'utente che si vuole chiamare.

Il metodo seguente

```
@Override
public void onResume() {
    super.onResume();
    if (Receiver.call_state != UserAgent.UA_STATE_IDLE)
        Receiver.moveTop();
    [...]
}
```

viene invocato dal sistema subito prima che l'activity sia pronta per l'interazione con l'utente. In questo caso viene controllato che lo stato attuale sia IDLE e in caso contrario viene chiamato il metodo **moveTop()** della classe **Receiver** che a sua volta lancerà a video l'interfaccia inCallScreen. Questo accade quando l'utente durante una telefonata preme il pulsante **back** senza chiudere la comunicazione. Quando l'applicazione viene lanciata nuovamente

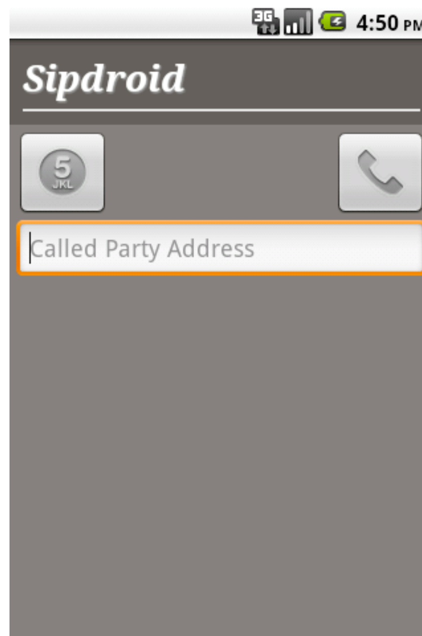


Figura 4.1: Interfaccia di Sipdroid

viene visualizzata direttamente la schermata inCallScreen al posto di quella principale.

Con il metodo

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    boolean result = super.onCreateOptionsMenu(menu);
    MenuItem m = menu.add(0, ABOUT_MENU_ITEM, 0, R.string.
        menu_about);
    m.setIcon(android.R.drawable.ic_menu_info_details);
    m = menu.add(0, EXIT_MENU_ITEM, 0, R.string.menu_exit);
    m.setIcon(android.R.drawable.ic_menu_close_clear_cancel)
        ;
    m = menu.add(0, CONFIGURE_MENU_ITEM, 0, R.string.
        menu_settings);
    m.setIcon(android.R.drawable.ic_menu_preferences);
    return result;
}
```

invocato dal sistema nel momento in cui l'utente preme il bottone *menu*, viene inizializzato per l'appunto il menu. Esso è costituito da tre tasti e il risultato della pressione di uno di questi viene gestito dal metodo

```
@Override
```

```java
public boolean onOptionsItemSelected(MenuItem item) {
    boolean result = super.onOptionsItemSelected(item);
    Intent intent = null;
    switch (item.getItemId()) {
    case ABOUT_MENU_ITEM:
        [...]
        m_AlertDlg = newAlertDialog.Builder(this)
        .[...].
        .show();
        break;
    case EXIT_MENU_ITEM:
        [...]
        Receiver.engine(this).unregister();
        [...]
        Receiver.engine(this).halt();
        Receiver.mSipdroidEngine = null;
        stopService(new Intent(this,RegisterService.class));
        finish();
        break;
    case CONFIGURE_MENU_ITEM: {
        try {
            intent = new Intent(this, org.sipdroid.sipua.ui.
               Settings.class);
            startActivity(intent);
            } catch (ActivityNotFoundException e) { }
        }
        break;
    }
    return result;
}
```

Il primo item stampa a video mediante un *alert dialog* informazioni sul software e sulla sua licenza. Questo messaggio, secondo i termini di rilascio dell'applicazione, non può essere modificato o eliminato.

Il secondo item serve a chiudere l'applicazione. In particolare il client viene deregistrato dal server, viene stoppata l'attività del **sipdroidEngine** e viene posto a **null** la variabile che ne deteneva l'istanza, viene fermato il servizio di registrazione che solitamente gira in backgroung e infine viene chiamato il metodo **finish()** che chiude l'applicazione liberando spazio in memoria centrale.

Il terzo item lancia il menu di configurazione di sistema che permette di settare parametri per la registrazione presso il server oltre a mettere a disposizione dell'utente utili opzioni.

Il metodo

```
void call_menu(){
    String target = this.sip_uri_box.getText().toString();
    [...]
    if (target.length() == 0)
        m_AlertDlg = new AlertDialog.Builder(this)
        .[...].
        .show();
    else if (!Receiver.engine(this).call(target,true))
        m_AlertDlg = new AlertDialog.Builder(this)
        .[...].
        .show();
}
```

serve per iniziare una chiamata. L'URI da contattare viene estratto dall'oggetto *sip_uri_box* attraverso il metodo **getText().toString()**. Viene in seguito controllato che la stringa contenga qualche carattere e infine con **Receiver.engine(this).call(target,true)** viene attivata la chiamata.

4.3 Compilazione e installazione di Sipdroid

Per ottenere i sorgenti di Sipdroid basta digitare su una shell il seguente comando

```
svn checkout http://sipdroid.googlecode.com/svn/trunk/
  sipdroid-read-only
```

Questo creerà all'interno della directory dalla quale è stato lanciato il comando la cartella *sipdroid-read-only* che verrà poi popolata con il sorgente dell'applicazione. Come accennato in precedenza, i codec, anch'essi rilasciati sotto licenza open source, per preservare le prestazioni non sono stati tradotti in java ma sono stati lasciati in linguaggio nativo (C, C++).

Questo però non rappresenta un problema in quanto per gli sviluppatori di applicazioni su Android è stato rilasciato l'NDK (Native Development Kit) che può essere scaricato senza vincoli dal sito di Android. Questo kit permette di compilare sorgenti in linguaggio nativo in modo da creare delle librerie utilizzabili mediante il meccanismo JNI di Java.

Per compilare questi sorgenti occorre modificare il file *Application.mk* contenuto all'interno della cartella `/path-to-sipdroid-read-only/jni` inserendo al posto della prima riga la seguente dicitura

```
APP_PROJECT_PATH := /path-to-sipdroid-read-only/
```

dove `/path-to-sipdroid-read-only/` rappresenta il cammino assoluto per la directory `sipdroid-read-only`.

Una volta modificato il file occorre entrare nella cartella dell'NDK ed eseguire il seguente comando

```
./ndk-build -C /path-to-sipdroid-read-only/
```

Le librerie generate verranno salvate all'interno della cartella `/path-to-sipdroid-read-only/libs/armeabi`.

Una volta eseguita la compilazione dei codec è possibile procedere alla compilazione dell'intera applicazione importando il progetto con Eclipse. Una volta importato il progetto, Eclipse provvederà in automatico a lanciare il processo di compilazione.

Nella cartella `bin` situata all'interno della directory `/path-to-eclipse-workspace/SipUA` è possibile trovare il risultato del processo di compilazione, ovvero i file .class, un unico file .dex, il file compresso resources.ap_ contenente le risorse e l'applicazione vera e propria che ha estensione .apk.

Un applicazione così compilata può essere installata su un dispositivo reale o sull'emulatore direttamante con Eclipse eseguendo il comando *Run –> Run As –> Android Application* dopo aver selezionato il giusto pacchetto all'interno del package explorer.

É possibile installare l'applicazione anche al di fuori di Eclipse a patto che si disponga del file con estensione .apk. Basta digitare il comando

```
$ adb install <file>
```

Nel caso in cui l'installazione non vada a buon fine occorre controllare che sul dispositivo sia abilitata la funzione di debug e l'installazione di applicazioni non scaricate dal market. Per far questo entrare nel menu settings di Android e abilitare l'opzione *Unknown Source* all'interno del menu Applications e *USB Debugging* all'interno del menu Applications/Development.

4.4 Asterisk

Asterisk è nato come progetto Open Source nel 2001 per la realizzazione di un centralino VoIP e TDM in grado di gestire le moderne comunicazioni VoIP e interfacce per la gestione di linee PSTN (analogiche e digitali). Il suo nome, Asterisk, proviene dal mondo Unix e Dos e rappresenta un cosiddetto "carattere jolly" (*) cioè la possibilità di rappresentare ogni file. In modo simile, Asterisk è stato progettato per interfacciare qualsiasi tipo di apparato telefonico standard (sia hardware che software) con qualsiasi applicazione telefonica standard, in modo semplice e consistente. Trattandosi di un progetto completamente Open Source, Asterisk è rilasciato sotto licenza GNU GPL ed il

suo contenuto è liberamente distribuibile e modificabile da chiunque favorendo così lo sviluppo di moltissime soluzioni e il suo continuo miglioramento grazie a centinaia di sviluppatori in tutto il mondo.

4.4.1 File di configurazione

Asterisk viene configurato mediante un certo numero di file situati nella directory `/etc/asterisk`. Questi hanno una sintassi formata da una o più sezioni (il cui titolo è racchiuso tra parentesi quadre ([ ]) e da variabili e oggetti (che si dichiarano con “=” e con “=>”); i commenti sono preceduti da punto e virgola. Un semplice dialplan può essere configurato con la modifica dei file **sip.conf** ed textbfextension.conf.

sip.conf

Il file di configurazione **sip.conf** contiene tutte le configurazioni dei telefoni o provider SIP per la comunicazione. Il file ha l'aspetto di tutti i file di configurazione del server Asterisk, ovvero è suddiviso in sezioni: normalmente è presente una sezione generale, chiamata *general*, dove vengono definiti dei parametri standard utilizzabili per tutti i terminali SIP, inoltre sono presenti varie sezioni, una per ogni telefono o provider SIP, in cui vengono configurati i parametri necessari a quel terminale.

Nella sezione [general] possono essere presenti i seguenti parametri:

- **allow=<codec>**: attiva quel determinato codec per la comunicazione.
- **disallow=all**: disattiva tutti i codec.
- **bindaddr=0.0.0.0**: è l'indirizzo IP usato dal server per l'ascolto delle richieste, il suo valore di default è 0.0.0.0, cioè l'ascolto viene effettuato su tutte le interfacce esistenti sul server.
- **bindport=5060**: è la porta utilizzata dal server per la ricezione dei messaggi sip. Di default viene utilizzata la porta 5060.
- **context=<nomecontesto>**: è il nome del contesto che viene usato quando nella definizione di un terminale SIP non viene specificato alcun contesto.
- **nat=yes/no**: identifica se il terminale è dietro un NAT.
- **language=<tipo>**: definisce la lingua da utilizzare nelle comunicazioni.

- **register=>username:password@host:porta/estensione**: parametro utilizzato per la registrazione ad un provider SIP. L'estensione definisce quale estensione viene chiamata quando arriva una chiamata da quel provider. Username e password sono i parametri usati per la connessione al provider.

Il file prosegue con la definizione dei terminali SIP, cioè telefoni o provider. Ogni sezione viene rappresentata da un nome racchiuso tra parentesi quadre, e definisce i parametri utilizzati per quel terminale. I parametri che si possono trovare sono:

- **username**: è lo username utilizzato per identificare il terminale connesso al server Asterisk;

- **secret**: è la password associata allo username per identificare il terminale connesso al server Asterisk;

- **callerid=nome <numtel>**: definisce il nome utente e il numero di telefono che viene visualizzato quando viene effettua una chiamata;

- **type**: definisce la relazione che deve avere il terminale con il server. I valori possibili sono: *peer* che permette all'entità di ricevere solo le chiamate, *user* che permette di eseguire solo le chiamate e textitfriend che permette all'entità sia di ricevere che di effettuare le chiamate;

- **context=<nomecontesto>**: se viene specificato questo contesto verrà utilizzato al posto di quello definito nella sezione general per l'esecuzione di una chiamata.

- **defaultip=<indirizzoIP>**: viene specificato l'indirizzo IP di default del telefono.

- **host**: campo che definisce l'indirizzo IP del terminale oppure il nome dell'host. Normalmente se si fa riferimento a un telefono viene definito con la parola *dynamic*, in modo tale da non imporre un vincolo sul telefono che si vuole configurare.

- **mailbox**: indica quale mailbox utilizzare per questo telefono. La mailbox definita in questo campo deve essere configurata nel file **voicemail.conf**.

- **fromuser**: viene usato per definire un provider SIP e sostituisce il parametro textitcallerid. In questo campo normalmente viene inserito il numero di telefono assegnato dal provider.

- **fromdomain**: come il campo *fromuser* anche questo viene usato quando si definisce un provider SIP. Questo parametro definisce il campo "From:" nei messaggi SIP scambiati.

- **dtmfmode**: definisce in quale modo vengono gestiti i toni DTMF. Si possono definire quattro tipi di toni DTMF: inband, rfc2833, info, auto.

extensions.conf

Il file di configurazione "extensions.conf " contiene il dialplan di Asterisk, cioè il piano di controllo e il flusso di esecuzione per tutte le sue operazioni. Questo file controlla le modalità con cui le chiamate in ingresso e quelle in uscita vengono gestite e instradate. Il suo contenuto è organizzato in sezioni, di cui due sono essenziali: *general* e *global*, in cui troviamo le impostazioni generali per tutto il DialPlan. Oltre a queste due sezioni ci sono i *contesti*, cioè i gruppi di lavoro che definiscono i comportamenti del DialPlan e le macro, ovvero delle procedure configurabili mediante istruzioni messe a disposizione da Asterisk.

La sezione *general* permette di configurare le poche opzioni riguardanti il DialPlan:

- **static=yes/no**: attiva o meno la possibilità di salvare il dialplan modificato, utilizzando il comando da console "save dialplan".

- **writeprotect=yes/no**: se writeprotect=no e static=yes allora si può salvare il dialplan direttamente da console con il comando "save dialplan".

La sezione globals, successiva a quella general, definisce le variabili globali utilizzabili in tutto il DialPlan. Queste variabili vengono utilizzate come delle costanti, quindi una volta inizializzate mantengono il loro valore per tutto il DialPlan. Esse possono essere modificate durante l'esecuzione del DialPlan, ma una volta terminata l'operazione, il valore ritorna quello definito nella sezione. Dopo la sezione "general" e "global", il DialPlan definisce un insieme di contesti ciascuno dei quali fornisce un insieme di estensioni.

Quando Asterisk riceve o effettua una connessione per una chiamata da parte di un utente, la chiamata viene indirizzata al contesto specificato nel file di configurazione del telefono. Per esempio se un telefono interno SIP esegue una chiamata, la chiamata viene indirizzata al contesto specificato nel file di configurazione "sip.conf ", mentre se arriva una chiamata dall'esterno, la chiamata viene indirizzata al contesto specificato nel file "zapata.conf ". Tutti i contesti specificati nei file di configurazione dei canali (es. sip, zapata, etc.)

devono essere definiti nel file “extensions.conf ”. Quando si definisce un’estensione all’interno di un contesto, si possono usare dei modelli di estensione come:

- **X**: qualsiasi numero da 0 a 9
- **Z**: qualsiasi numero da 1 a 9
- **N**: qualsiasi numero da 2 a 9
- **[12-5]**: qualsiasi numero appartenente alla lista 1, 2, 3, 4, 5
- **.**: è un carattere jolly, che può essere uno o più caratteri. Tali modelli devono essere preceduti dal carattere “ ”.

In alcuni casi si possono utilizzare delle estensioni predefinite che Asterisk mette a disposizione, tra le quali:

- **i**: invalid, usata quando viene digitata una estensione non valida.
- **s**: start, usata per accettare qualsiasi tipo di estensione. Normalmente viene usata nelle macro.
- **h**: hangup, usata quando un canale viene chiuso.
- **t**: timeout, usata quando scade il tempo in un menu.
- **T**: timeout assoluto, usata quando scade il tempo di una chiamata
- **operator**: usata nella voicemail quando si digita lo 0 per contattare l’operatore.

Un’estensione viene definita come una riga di comando che viene eseguita con una certa priorità, espressa dal tag *priority*. Quando un’estensione viene chiamata vengono eseguiti i comandi in ordine crescente di priorità. Questa procedura termina quando:

- la chiamata viene chiusa;
- un comando restituisce un risultato di -1 (indica che è fallito);
- non esiste un comando con priorità superiore;
- la chiamata viene instradata su una nuova estensione.

La sintassi usata per ogni comando è la seguente:

```
exten => estensione , priorità ,comando( parametri )
```

Nel DialPlan si trovano moltissime applicazioni che servono per costruire e gestire il PBX, nel seguito verranno prese in esame solo alcune applicazioni che vengono usate più comunemente:

- **Answer()**: risponde a un canale che squilla.
- **Dial(type/group/identifier,timeout,options,URL)**: effettua una chiamata sul canale type (es. SIP), nel gruppo group (es. g2) e con estensione identifier (es. 200). Se dopo timeout secondi il canale non risponde, allora la chiamata viene chiusa.
- **Hangup**: chiude il canale attivo.

4.4.2 Installazione e configurazione del server Asterisk

Il server Asterisk è stato utilizzato per testare l'applicativo Sipdroid. I test sono stati realizzati sia in ambiente virtuale che in ambiente reale. Nel primo caso è stato utilizzata la versione di Asterisk denominata Asterisk Now (comprensiva di sistema operativo basato sulla distribuzione Centos e fornito come immagine .iso) e una macchina virtuale con motore Qemu per sfruttare la compatibilità con l'emulatore fornito insieme all'SDK, il tutto su un sistema Macintosh. Nel secondo caso si è installato Asterisk su una macchina reale con sistema operativo Ubuntu.

Test virtuale

Il test virtuale è stato realizzato soprattutto per testare il signaling del protocollo Sip e il comportamento dell'interfaccia grafica di Sipdroid durante una chiamata.

Per quanto riguarda questo test occorre per prima cosa configurare la macchina Qemu. Questo può essere fatto attraverso l'interfaccia grafica dell'applicazione Q-emulator.

Le caratteristiche principali della macchina virtuale utilizzata sono:

- un processore x86
- 256 MB di ram
- scheda di rete Ne2000 PCI
- Harddisk_1.qcow2

- CD-ROM AsteriskNOW-1.5.0-i386-1of1.iso
- Boot from harddisk

Per quanto riguarda la rete è stato abilitato il redirect che mappa la porta 50600 dell'host verso la porta 5060 del guest per poter utilizzare il protocollo SIP.

Una volta terminata l'installazione del server occorre configurare il file sip.conf ed extensions.conf. Per far questo basta aprire con un editor il file `/etc/asterisk/sip.conf` e modificarlo inserendovi le seguenti stringhe:

```
[general]
context=local
bindport=5060
bindaddr=0.0.0.0
disallow=all
allow=ulaw
allow=alaw
allow=gsm
language=it
canreinvite=no
nat=yes

[200]
type=friend
secret=200
host=dynamic
qualify=no

[201]
type=friend
defaultuser=201
callerid="201" <201>
secret=201
host=dynamic
```

mentre il file */etc/asterisk/extensions.conf* va editato nel modo seguente:

```
[general]
static=yes
writeprotect=no
clearglobalvars=no

[globals]
```

```
[local]
exten => _2XX,1,Dial(SIP/${EXTEN})
exten => _2XX,2,Hangup()
```

Per rendere effettive le modifiche occorre riavviare la macchina virtuale.

Una volta configurato il server vanno configurati i client per poter effettuare la registrazione su di esso. Sipdroid, installato sull'emulatore, va configurato nel modo seguente:

- Authorization Username = 200
- Password = 200
- Server or Proxy = 10.0.2.2
- Port = 50600

Dopo aver inserito questi parametri dovrebbe comparire un pallino verde in alto a sinistra dell'interfaccia dell'emulatore confermante l'avvenuta registrazione. Se ciò non dovesse succedere controllare che sia spuntata la casella 3G all'interno del menù Call Option di Sipdroid.

Come telefono remoto può essere utilizzato un qualsiasi softphone scaricabile da internet. Bisogna ricordarsi di configurarlo con i seguenti parametri:

- Authorization Username = 201
- Password = 201
- Server or Proxy = 127.0.0.1
- Port = 50600

A questo punto è possibile effettuare una chiamata tra i due client come mostrato in figura 4.2

Test reale

Per installare il server Asterisk su una macchina con sistema operativo Ubuntu occorre eseguire le seguenti operazioni:

- prima di tutto bisogna installare o aggiornare i pacchetti necessari al corretto funzionamento di Asterisk e i tools da utilizzare per la sua compilazione.

  ```
  $ sudo apt-get install make gcc g++ bison linux-sources
      -2.6.18 linux-headers-2.6.18-5-686 libncurses5-dev
  ```

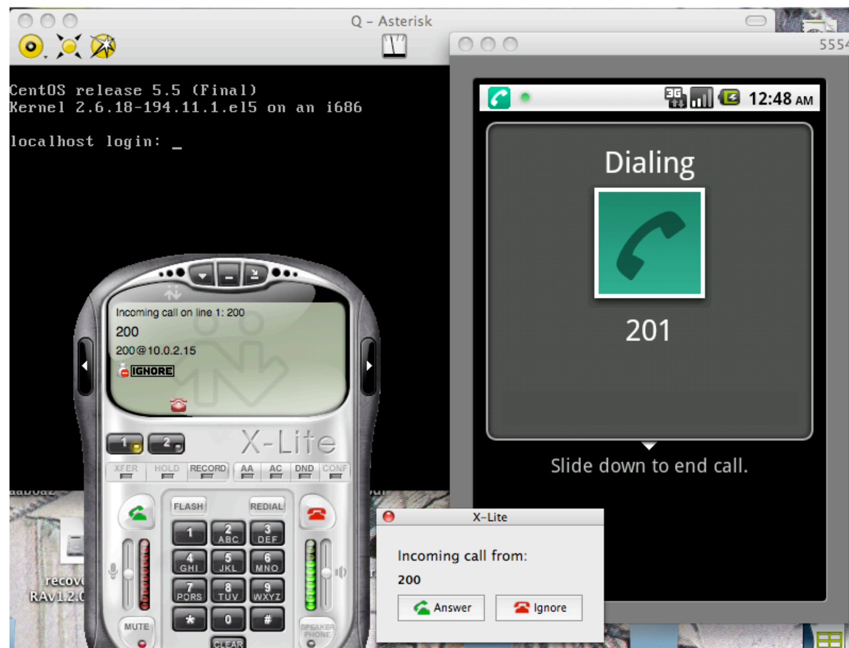


Figura 4.2: Test virtuale

- Si può ora procedere allo scaricamento dei sorgenti, alla loro compilazione e installazione.

```
$ cd /usr/src/
$ mkdir asterisk/
$ cd asterisk/
$ wget http://downloads.digium.com/pub/asterisk/releases
    /asterisk-1.6.2.13.tar.gz
$ tar -xvf asterisk-1.6.2.13.tar.gz
$ cd asterisk-1.4.18/
$ sudo ./configure
$ sudo make
$ sudo make install
$ sudo make samples
```

Una volta eseguita l'installazione occorre configurare i file *sip.conf* ed *exstensions.conf* come mostrato nel caso del test virtuale. Nel file *sip.conf* non è necessaria la stringa **nat=yes** che in questo caso può essere rimossa.

Terminata la configurazione avviare il daemon Asterisk digitando su una shell il seguente comando

```
$ sudo asterisk
```

Per verificare il corretto avvio del server digitare

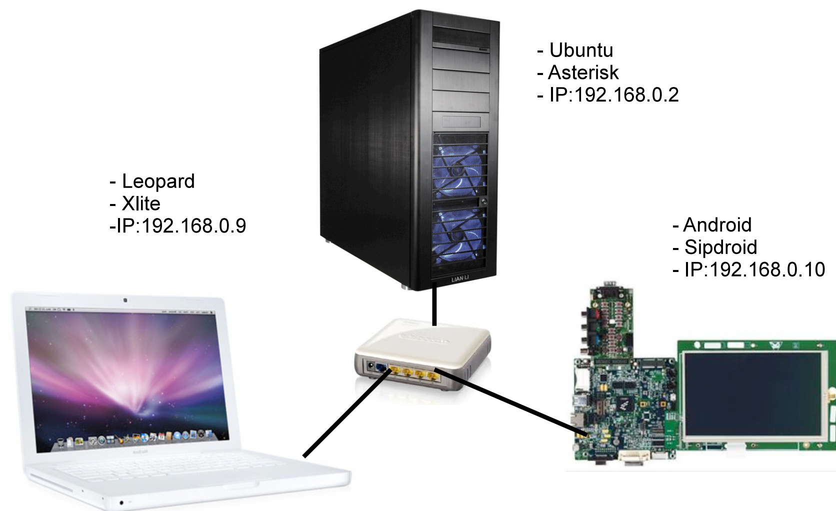


Figura 4.3: Test reale

```
$ netstat -al | grep sip
```

che deve restituire la seguente riga

```
udp        0          0 *:sip  *:*
```

La configurazione del test è mostrata in figura 4.3.
Il softphone Xlite è stato configurato con i seguenti parametri:

- Authorization Username = 201
- Password = 201
- Server or Proxy = 192.168.0.2
- Port = 5060

e Sipdroid con questi parametri:

- Authorization Username = 200
- Password = 200
- Server or Proxy = 192.168.0.2
- Port = 5060

Le interfacce di rete utilizzate per la realizzazione del test sono di tipo Ethernet. Sipdroid non gestisce in modo nativo questo tipo di interfaccia essendo stato sviluppato per dispositivi mobili che solitamente ne sono sprovvisti. Per ovviare a questo problema, che implica il mancato riconoscimento da parte dell'applicativo del collegamento alla rete, occorre modificare la classe **Receiver** con l'aggiunta del metodo

```
static boolean isFastEth() {
    boolean on_eth = false;
    try {
        for (Enumeration<NetworkInterface> en =
           NetworkInterface.getNetworkInterfaces(); en.
           hasMoreElements();){
            NetworkInterface intf = en.nextElement();
            if (intf.getName() != null && intf.getName().
               startsWith("eth")){
            on_eth = true;
            on_wlan = true;
            break;
            }
        }
    }catch (SocketException ex) { }
    return on_eth;
}
```

che scandisce le interfacce di rete fino a trovarne una inizializzata con prefisso “eth”. In caso affermativo viene posta a **true** la variabile identificante l'interfaccia WiFi e l'interfaccia ethernet viene trattata come se fosse quest'ultima.

Poi bisogna modificare il metodo **isFast()** nel modo seguente

```
public static boolean isFast() {
    is_fast = isFastWifi();
    if (!is_fast) is_fast = isFastGSM();
    if (!is_fast) is_fast = isFastEth();   //aggiunto
    return is_fast;
}
```

che avvisa il chiamante della disponibilità di una interfaccia di rete. Una volta ricompilato sarà possibile installare ed eseguire con successo Sipdroid sulla board si sviluppo i.mx51.

Capitolo 5

Sipbell

5.1 Introduzione

In questo capitolo verranno discusse le scelte e le motivazioni principali che hanno portato alla realizzazione del citofono. Le scelte hanno riguardato soprattutto l'implementazione dell'interfaccia principale e la gestione degli stati appartenenti ad una chiamata. In un sistema dotato di uno schermo che permette un'interazione del sistema con il suo utilizzatore non basato solamente sui segnali vocali ma anche visivi si è pensato di gestire con notifiche gli eventi appartenenti ai cambiamenti di stato del citofono. Questi sono stati ulteriormente semplificati rispetto agli originali per un'interazione più familiare e intuitiva anche per un utilizzatore alle prime armi.

Inoltre, è stato implementato un sistema di sicurezza per la prevenzione e rilevamento di accessi non autorizzati volti a modificare le impostazioni del citofono. Questo perchè il sistema viene considerato pubblico dal punto di vista dell'utilizzo ma privato dal punto di vista della configurazione.

5.2 Interfaccia principale

L'interfaccia dell'applicazione Sipbell è stata realizzata in modo tale da rendere il suo utilizzo estremamente semplice e familiare, infatti ricalca le sembianze di un comune citofono. L'interfaccia viene definita nel file *SipBell/src/res/layout/sipdroid.xml* nel modo seguente:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
    android"
     android:layout_width="fill_parent"
     android:layout_height="fill_parent"
     android:background="#736f6e"
```

```
    android:orientation="vertical"
    android:id="@+id/screen">
    <LinearLayout
        android:orientation="vertical"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:background="#504D4C"
        android:padding="10dip">
        <TextView
            android:id="@+id/account_string"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:textColor="#cccccc"
            android:textSize="12dip"
            android:paddingTop="5dip"
            android:visibility="gone"
            android:text="\?\?\?"/>
    </LinearLayout>
    <LinearLayout
        android:padding="10dip"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal"
        android:layout_gravity="center_vertical|center"
        android:gravity="center_horizontal">
        <Button
            android:id="@+id/call_button"
            android:layout_height="fill_parent"
            android:layout_width="fill_parent"
            android:drawableLeft="@android:drawable/ic_menu_call"/
               >
    </LinearLayout>
</LinearLayout>
```

Essa è costituita da un **TextView** utilizzato per stampare a video una stringa rappresentante l'URI utilizzato dal client per la registrazione al server e da un **Button**, rappresentante il tasto utilizzato per attivare una telefonata.

Questo file viene renderizzato da Android come nella figura 5.1.

Il bottone deve anche essere definito all'interno dell'activity che gestisce tale interfaccia, ovvero la classe Sipdroid. Il bottone viene creato e inizializzato all'interno del metodo **onCreate()**:

```
Button callButton;
@Override
public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    [...]
    callButton = (Button) findViewById(R.id.call_button);
```

```
    callButton.setOnClickListener(new Button.OnClickListener
        () {
         public void onClick(View v) {
              mode = NORMAL;
              call_menu(mode);
              }
    [...]
});
```

Nel momento in cui viene premuto il tasto, il sistema chiama il metodo **onClick()** della classe **OnClickListener**, che a sua volta invoca il metodo **call_menu(mode)** che effettua una chiamata con la modalità "normal". Con il termine normal viene definita una chiamata attivata intenzionalmente da un utente. Per poter supportare diverse modalità di chiamata il metodo **call_menu** è stato modificato nel seguente modo:

```
void call_menu(int mode) {
    [...]
    if (target.length() == 0)
         m_AlertDlg = new AlertDialog.Builder(this).
         .[...].
         .show();
    else if (!Receiver.engine(this).call(target,mode))
         m_AlertDlg = new AlertDialog.Builder(this)
         .[...].
         .show();
}
```

dove *target* è la variabile rappresentante l'URI collegato al bottone. Questa variabile viene configurata attraverso il menu opzioni di Sipbell e inizializzata all'interno del metodo **onResume()**

```
@Override
public void onResume() {
    super.onResume();
    target=getStringAccount();
    setAccountString();
    setButtonAccount1();
    [...]
}
```

Il metodo **getStringAccount()** reperisce le informazioni sull'account costruendo un oggetto di tipo **SharedPreferences** che in seguito passerà al metodo statico **getButtonAccountString()** della classe **Settings**. **setAccountString()** visualizza l'URL del citofono mentre **setButtonAccount1()** rende visibile il nome sul bottone chiamata.

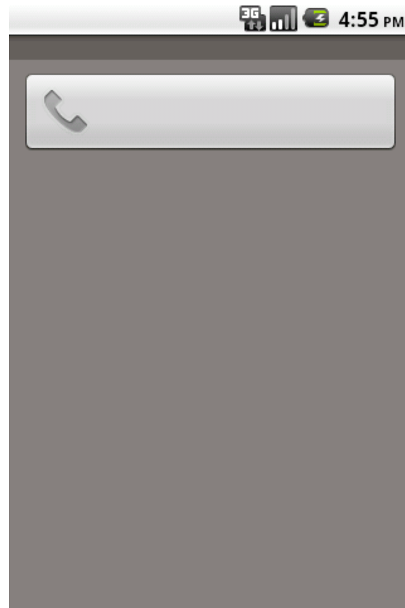


Figura 5.1: Interfaccia iniziale di SipBell

5.2.1 Interfaccia "CALL"

Nel momento in cui l'utente preme il bottone viene attivata una telefonata verso l'URI definito dalla variabile *target*. In questo caso il sistema entra nello stato "OUTGOING_CALL" e tale cambiamento viene notificato all'utente mediante l'utilizzo di un Alert Dialog come mostrato in figura 5.2 (a).

La parte di codice che gestiste un OUTGOING_CALL è stata implementata all'interno del metodo **onResume()** della classe **Sipdroid** nel modo seguente:

```
public static AlertDialog progDialog;
@Override
public void onResume() {
    super.onResume();
    [...]
    switch(Receiver.call_state){
        [...]
        case UserAgent.UA_STATE_OUTGOING_CALL:
        Sipdroid.layout.setKeepScreenOn(true);
        callButton.setVisibility(View.GONE);
        AlertDialog.Builder build = new AlertDialog.Builder(
            this);
        build.setTitle("   Dialing...    ");
        build.setCancelable(false);
```

```
        if(mode){
            build.setMessage("Press hangup to cancel the
                call");
            build.setNeutralButton("Hangup", new
                DialogInterface.OnClickListener() {
                 public void onClick(DialogInterface dialog,
                     int id) {
                      dialog.cancel();
                      hangup();
                     }
                 });
        }
        progDialog = build.create();
        progDialog.setIcon(R.drawable.picture_dialing_new);
        progDialog.show();
        break;
        [...]
}
```

Lo scopo di questa finestra di dialogo è quello di avvisare l'utente che la chiamata è stata inoltrata correttamente e che il telefono destinatario della stessa sta squillando. L'alert dialog è stato configurato in modo tale da disabilitare la funzione associata alla pressione del tasto **back** posto sul dispositivo.

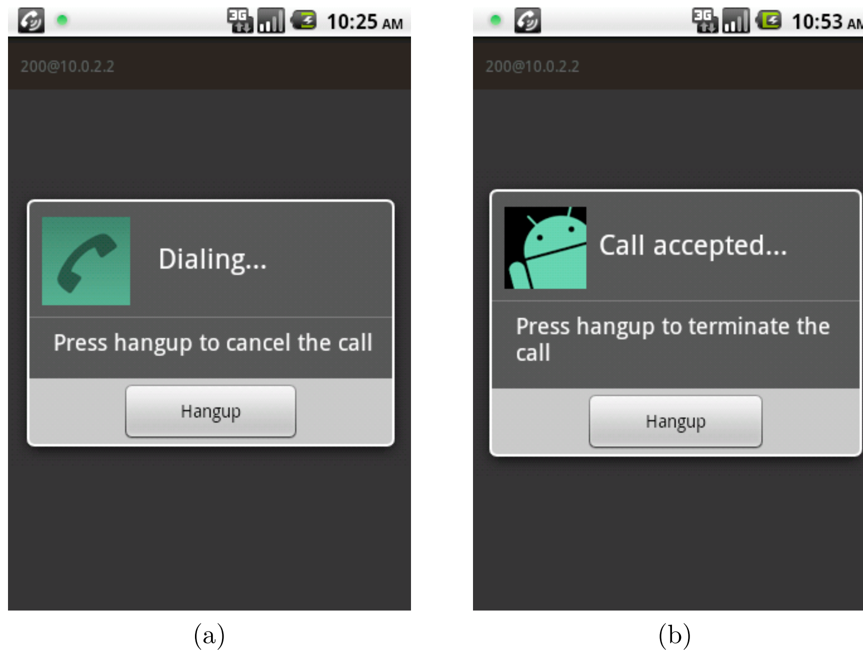
Nel momento in cui l'utente che riceve la chiamata decide di accettare la stessa, il citofono passa dallo stato OUTGOING_CALL allo stato IN_CALL. Questo stato viene gestito in modo molto simile allo stato OUTGOING_CALL, ovvero mediante un alert dialog (figura 5.2 (b)) che avvisa l'utente che la chiamata è stata accettata e che quindi la conversazione può avere inizio.

Sia nel caso dell'OUTGOING_ALL che nel caso IN_CALL è data l'opportunità all'utente di terminare la chiamata premento il bottone **Hangup**. Quest'azione produce la cancellazione dell'alert dialog e la chiusura della sessione mediante l'invocazione del metodo **hangup()**

```
protected void hangup(){
    Receiver.engine(this).rejectcall();
}
```

che chiude la telefonata agendo sul metodo **rejectcall()** della classe **SipdroidEngine**.

Il metodo **Sipdroid.layout.setKeepScreenOn(true)** serve a tenere lo schermo attivo durante la conversazione.



(a) (b)

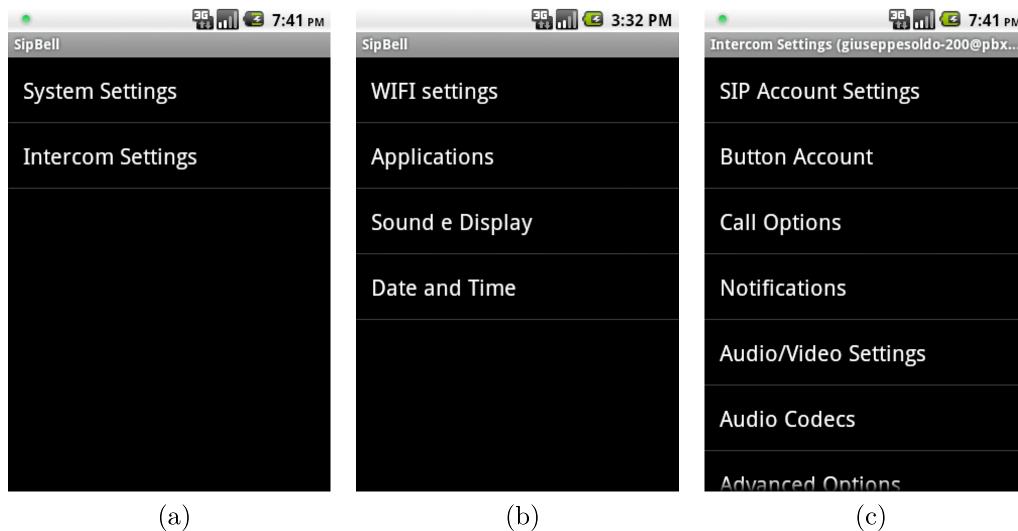
Figura 5.2: Interfaccia CALL

5.2.2 Menu opzioni

Sipdroid mette a disposizione dell'utente un comodo menu per la cofigurazione del telefono. Oltre all'inserimento dei dati relativi alla registrazione sul server è data la possibilità all'utente di scegliere, per esempio, il tipo di codec da utilizzare, il modo di gestire una chiamata in entrata, per esempio con risposta automatica, il tipo di connessione (Wifi, 3G, EDGE), l'abilitazione dello STUN, l'impostazione del guadagno del microfono o dell'altoparlante, ecc... Oltre a queste opzioni, proprie di Sipdroid, si è cercato di aggiungere al menu generale anche alcune impostazioni, ritenute importanti per la gestione di un citofono, proprie del sistema Android. In questo modo, se l'utente, che ha il proprio citofono allacciato alla rete mediante un'interfaccia ethernet vuole collegare il dispositivo ad una rete wireless protetta, non deve uscire dall'applicazione ma può effettuare le modifiche all'interno della stessa. Questo si integra con l'idea di realizzare un'applicazione che rimpiazzi in modo completo l'application home di Android che non dovrà essere più utilizzata.

Il menu implementato ha una struttura ad albero con due nodi posizionati al secondo livello (figura 5.3 (a)), uno che gestisce il setting di Android (figura 5.3 (b)) e uno che gestisce il setting di Sipbell (figura 5.3 (c)).

Il menu deve essere implementato all'interno della stessa classe che gestisce l'interfaccia nella quale verrà visualizzato. Nel nostro caso si tratta della classe



(a) (b) (c)

Figura 5.3: Menu di Sipbell

Sipdroid. La creazione di un menu si concretizza con la chiamata al metodo **onCreateOptionsMenu()**. Questo metodo viene chiamato solo la prima volta che il menu diventa visibile sullo schermo.

```
MenuItem m;
Menu x;
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    boolean result = super.onCreateOptionsMenu(menu);
    x = menu;
    MenuItem m = menu.add(0, ABOUT_MENU_ITEM, 0, R.string.
        menu_about);
    m.setIcon(android.R.drawable.ic_menu_info_details);
    m = menu.add(0, EXIT_MENU_ITEM, 0, R.string.menu_exit);
    m.setIcon(android.R.drawable.ic_menu_close_clear_cancel)
        ;
    m = menu.add(1, CONFIGURE_MENU_ITEM, 0, R.string.
        menu_settings);
    m.setIcon(android.R.drawable.ic_menu_preferences);
    m = menu.add(1, PASSWORD_MENU_ITEM, 0, "Password");
    m.setIcon(android.R.drawable.ic_lock_lock);
    return result;
}
```

Si è deciso di mantenere pubbliche le variabili di gestione del menu in modo da poter essere utilizzate anche all'esterno del metodo **onCreateOptionsMe-**

nu.

Per una gestione più personalizzata del menu sono stati sovrascritti altri metodi della classe Activity:

- ```
  @Override
  public boolean onMenuOpened (int featureId , Menu menu){
      callButton.setVisibility(View.GONE);
      x.setGroupEnabled(1, false);
      customizeDialog.show();
      return enter;
  }
  ```

  Questo metodo viene chiamato un attimo prima che il menu sia visibile sullo schermo. Ha il compito di rendere invisibile il bottone chiamata in modo da creare più spazio all'interno dello schermo, disabilita gli item del menu appartenenti al gruppo 1 definiti nella costruzione del menu (metodo **onCreateOptionsMenu()**) e visualizza a schermo un alert dialog che verrà descritto nella sottosezione successiva.

- ```
  @Override
  public void onOptionsMenuClosed (Menu menu){
      if(callButton.getVisibility()==View.GONE){
          callButton.setVisibility(View.VISIBLE);
      }
  }
  ```

  viene chiamato al momento della chiusura del menu. Riporta l'interfaccia del sistema allo stato IDLE.

- ```
  @Override
  public boolean onOptionsItemSelected(MenuItem item) {
  boolean result = super.onOptionsItemSelected(item);
  Intent intent = null;
      switch (item.getItemId()) {
      case ABOUT_MENU_ITEM:
          [...]
      break;
      case EXIT_MENU_ITEM:
          [...]
      break;
      case PASSWORD_MENU_ITEM:
          [...]
          break;
      case CONFIGURE_MENU_ITEM:
  ```
  ```

```
            try{ intent = new Intent(this, org.sipdroid.
                sipua.ui.GenSettings.class);
                 startActivity(intent);
                 } catch (ActivityNotFoundException e){ }
        }        return result;
    }
```

gestisce la pressione sugli item del menu. Rispetto a Sipdroid è stato aggiunto l'item Password ed è stato modificato l'item CONFIGURE_MENU che non lancia più l'activity **Settings** ma l'activity **GenSettings** che ha preso il suo posto come nodo radice nella struttura del menu.

5.3 Sicurezza

Il normale utilizzo di un citofono prevede un solo proprietario ma, a differenza di un sistema mobile come un cellulare, ha più utilizzatori. Alcuni di questi potrebbero utilizzare il citofono in modo improrio, per esempio per fare delle telefonate private sfruttando la rete del proprietario riconfigurando l'uri legato al bottone call. Poichè si è voluta lasciare una configurazione dinamica del settings si è deciso di proteggere il menu con l'ausilio di un'autenticazione basata su password. In questo modo l'accesso al menù viene lasciato libero solo a chi conosce la suddetta password.

5.3.1 Autenticazione

Nel momento in cui viene premuto il bottone menu, viene chiamato, come accennato in precedenza, il metodo

```
@Override
public boolean onMenuOpened (int featureId, Menu menu){
    [...]
    customizeDialog.show();
    return enter;
}
```

che tra le altre cose ha il compito di visualizzare sullo schermo il **customizeDialog** (figura 5.4).

Questo ha il compito di inizializzare una variabile di tipo stringa con il valore inserito dall'utente all'interno del text edit box e di confrontarlo con la stringa rappresentante la password di sistema. Il dialogo è stato implementato come classe interna della classe Sipdroid. Il suo costruttore **Pass(Context**

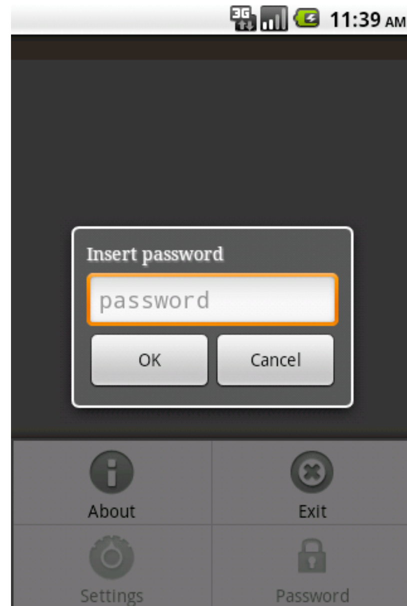


Figura 5.4: Dialogo “insert password”

context) riceve in ingresso un oggetto di tipo Contex e inizializza quelle che sono le View all’interno del dialogo. Mediante il metodo **setContentView** il dialogo si allaccia alla schermata *pass_screen*. Quest’ultima viene utilizzata anche per un’altra funzione, l’inserimento di una nuova password, e per questo nella costruzione della vista finale l’elemento *repeat* viene reso invisibile.

```
private int number = 0;
class Pass extends Dialog implements OnClickListener{
    [...]
    long timer0;
    public Pass(Context context) {
        super(context);
            setContentView(R.layout.pass_screen);
            title = (TextView) findViewById(R.id.TextView01)
                ;
            title.setText("Insert␣password");
            repeat = (EditText)findViewById(R.id.EditText01)
                ;
            repeat.setVisibility(View.GONE);
            okButton = (Button) findViewById(R.id.OK);
            okButton.setOnClickListener(this);
            cancelButton = (Button)findViewById(R.id.Cancel)
                ;
            cancelButton.setOnClickListener(this);
```

```
            text = (EditText) findViewById(R.id.pass);
        }
        [...]
}
```

Una volta inserita la password l'utente deve premere il tasto **OK** per avviare la procedura di autenticazione. Questa viene implementata all'interno del metodo **onClick()**:

```
[...]
public void onClick(View v) {
    passw = text.getText().toString();
    try{ File sett = new File("/data/data/org.sipdroid.sipua
        /files/"+FILENAME);
    [...]
    FileInputStream fos = openFileInput(FILENAME);
    [...]
    fos.read(buffer);
    fos.close();
    p = new String(buffer);
    }catch(IOException e){[...]}
    if (v == cancelButton){
        [...]
    } else {
        if(passw.equals(p)){
            number=0;
            password=true;
            x.setGroupEnabled(1, password);
            dismiss();
        } else {
            password=false;
            x.setGroupEnabled(1, password);
            if(number!=3){
                [...]
                toast.show();
            }
            if(number == 0){
                timer0 = System.currentTimeMillis()
                    /(1000*60);
            }else if(number == 3){
                number = 0;
                long timer1 = System.currentTimeMillis()
                    /(1000*60);
                if(timer1-timer0 < 5){
```

```
                    enter = false;
                    mode = ALERT;
                    dismiss();
                    x.close();
                    text.setText("");
                    call_menu(mode);
                    return;
                    }
                }
                number++;
                }
            }
        text.setText("");
}
}
```

La password di sistema viene letta dal file *"/data/data/org.sipdroid.sipua /files/"+FILENAME* e salvata all'interno della variabile *p*. Se la password inserita dell'utente coincide con quella di sistema viene abilitato il menu (figura 5.5 (a)), altrimenti viene segnalato l'errore all'utente mediante l'utilizzo di un oggetto di tipo **toast** (figura 5.5 (b)) e se questo è il primo tentativo che non va a buon fine viene attivato un timer. All'utente vengono permessi, all'interno di un lasso di tempo pari a cinque minuti, quattro errori consecutivi. Al quarto errore, il sistema individua un tentativo di intrusione ed avvisa il proprietario del telefono mediante l'attivazione di una chiamata in modalità ALERT.

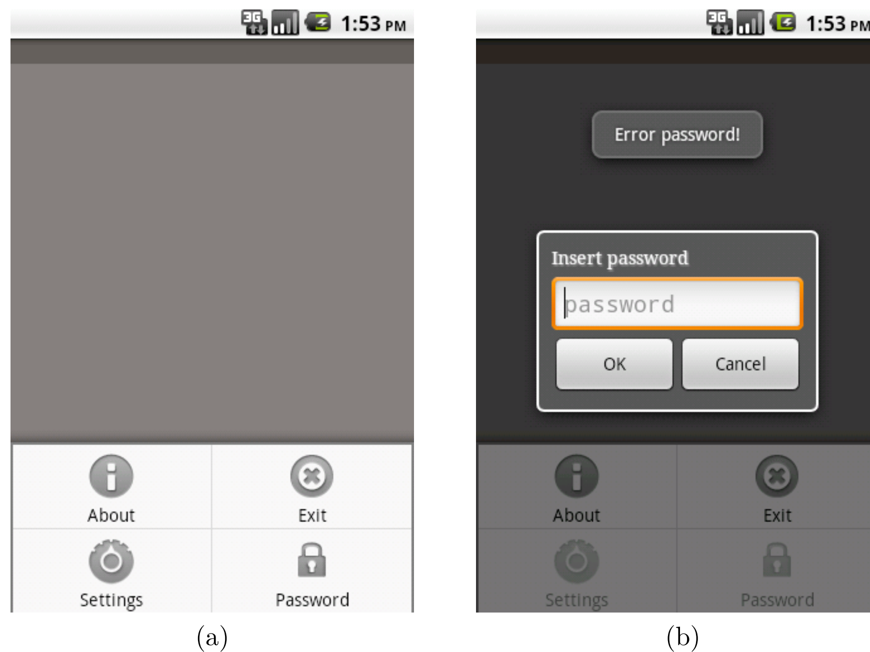
5.3.2 ALERT CALL

Come accennato in precedenza sono state definite due modalità di chiamata, una "NORMAL" e l'altra "ALERT". Una chiamata ALERT viene inoltrata in reazione ad un tentativo di intrusione non autorizzata rilevata dal sistema. Nel nostro caso, per tentativo di intrusione si intende il cercare di indovinare la password che protegge l'accesso al menu opzioni del citofono.

La chiamata si concretizza con l'invocazione del metodo **call_menu(mode)**. Il ricevente di tale chiamata deve poterla distinguere da una chiamata normale senza dover obbligatoriamente premere il tasto accept sul proprio telefono. In questo caso si è utilizzato il display name per segnalare all'utente l'infrazione.

Quando viene attivata una chiamata viene invocato, sull'oggetto di tipo **UserAgent**, il metodo **call()**

```
/** Makes a new call (acting as UAC). */
public boolean call(String target_url, int mode) {
    [..]
```



(a) (b)

Figura 5.5: Dialog "password"

```
    changeStatus(UA_STATE_OUTGOING_CALL, target_url);
    String from_url=null;
    if (mode == Sipdroid.NORMAL){
        from_url = user_profile.from_url;
    }
    if(mode == Sipdroid.ALERT){
        from_url = "sip:Alert@citofono.it";
    }
    createOffer();        //change end
    call = new ExtendedCall(sip_provider, from_url,
       user_profile.contact_url, user_profile.username,
       user_profile.realm, user_profile.passwd, this);
    [...]
    return true;
}
```

che riceve in ingresso, oltre all'URI dell'entità da contattare, la modalità con cui effettuare la chiamata. Nel caso di un ALERT, la variabile *from_url*, rappresentate l'URL del chiamante, viene impostata a *sip:Alert@citofono.it* e viene passata al costruttore dell'oggetto **ExtendedCall**. In questo modo, sul display del telefono del ricevente comparirà un caller id uguale ad Alert (figura 5.6 (b)).

Per evitare che la telefonata venga terminata dall'utilizzatore del citofono prima che questa venga accettata dal proprietario è stato disabilitato il bottone Hangup. In questo modo, l'ipotetico intruso non può più agire sul citofono che rimarrà bloccato finchè il proprietario non decide di terminare la chiamata (figura 5.6 (a)). Questo sistema è stato implementato nel metodo **onResume()**: se *mode != NORMAL* il bottone Hangup non viene gestito.

```
@Override
public void onResume() {
    super.onResume();
    [...]
    switch(Receiver.call_state){
        [...]
        case UserAgent.UA_STATE_OUTGOING_CALL:
        [...]
        if(mode == NORMAL){
            build.setMessage("Press␣hangup␣to␣cancel␣the␣
               call");
            build.setNeutralButton("Hangup", new
               DialogInterface.OnClickListener() {
                public void onClick(DialogInterface dialog,
                   int id) {
                    dialog.cancel();
                    hangup();
                    }
                });
        }
        [...]
        break;
        case UserAgent.UA_STATE_IN_CALL:
        [...]
        if(mode == NORMAL){
            build.setMessage("Press␣hangup␣to␣cancel␣the␣
               call");
            build.setNeutralButton("Hangup", new
               DialogInterface.OnClickListener() {
                public void onClick(DialogInterface dialog,
                   int id) {
                    dialog.cancel();
                    hangup();
                    }
                });
        }
```

(a) (b)

Figura 5.6: ALERT call

```
    [...]
}
```

5.3.3 Inserimento di una nuova password

La scelta della password per la protezione del sistema è stata lasciata all'utente finale. L'inserimento di una nuova password viene gestito all'interno dell'item PASSWORD_MENU_ITEM

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    boolean result = super.onCreateOptionsMenu(menu);
    [...]
    m = menu.add(1, PASSWORD_MENU_ITEM, 0, "Password");
    m.setIcon(android.R.drawable.ic_lock_lock);
    return result;
}
```

la cui selezione viene notificata al metodo

```
NewPass aggPass;
@Override
public boolean onOptionsItemSelected(MenuItem item) {
```

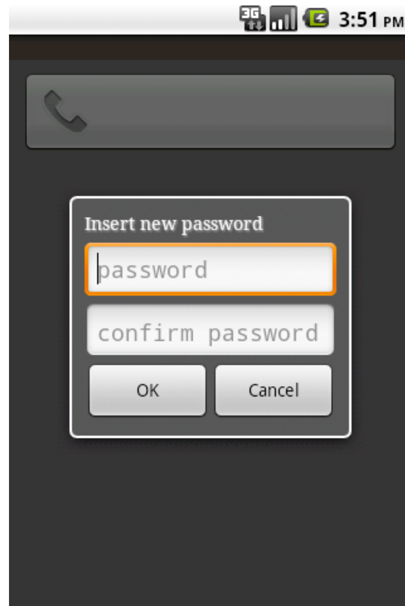


Figura 5.7: New Password

```
    boolean result = super.onOptionsItemSelected(item);
    Intent intent = null;
        switch (item.getItemId()) {
            [...]
        case PASSWORD_MENU_ITEM:
            aggPass = new NewPass(this);
            aggPass.setCancelable(false);
            aggPass.show();
        break;
        [...]
    }
    return result;
}
```

che ha il compito di creare un oggetto di tipo **NewPass** e di visualizzarlo sullo schermo (figura 5.7).

NewPass è una classe interna a Sipdroid ed ha la funzione di inizializzare e gestire un oggetto di tipo **Dialog**.

```
private static final String FILENAME = "password_settings";
private static String PASSWORD;
class NewPass extends Dialog implements OnClickListener{
    [...]
    public NewPass(Context context) {
```

```
        super(context);
        setContentView(R.layout.pass_screen);
        [...]
        text = (EditText) findViewById(R.id.pass);
        repeat = (EditText) findViewById(R.id.EditText01);
        }
        public void onClick(View v) {
            passw = text.getText().toString();
            confPass = repeat.getText().toString();
            if(v == okButton && !passw.equals("")){
                if(passw.equals(confPass)){
                    try{ FileOutputStream fos =
                       openFileOutput(FILENAME, Context.
                       MODE_PRIVATE);
                        fos.write(passw.getBytes());
                        fos.close();
                        dismiss();
                    }catch(IOException e){}
                }else{
                [...]
                toast.show();
            }
            }else{
                dismiss();}
                text.setText("");
                repeat.setText("");
    }
}
```

Il costruttore inizializza la finestra di dialogo collegandolo al layout definito nel file *pass_screen*, lo stesso utilizzato per la finestra di dialogo che gestisce l'autenticazione solo che in questo caso l'elemento *repeat* non viene reso invisibile.

La variabile *passw* viene inizializzata con la stringa inserita nella prima casella di testo mentre la variabile *confPass* viene inizializzata con la stringa inserita nella seconda casella di testo. In seguito alla pressinone del tasto **OK** le due stringhe vengono confrontate e se risultano uguali viene creato un file che conterrà la password che si vuole salvare in modo permanente. Il file viene creato con la modalità *Context.MODE_PRIVATE*. Questo assicura che l'unica applicazione che lo può leggere o modificare è quella che lo ha creato. Questo file viene inoltre salvato all'interno della directory `/data/data/org.sipdroid.sipua/files/` che non può essere aperta attraverso un terminale se non si hanno privilegi di root. L'Android installato sui

dispositivi in commercio non supporta la modalità root. Questa può essere abilitata installando sul dispotivo versioni di Android “non ufficiali” come per esempio CyanogenMod. Per questo motivo si è deciso di non criptare la password, per esempio con l’algoritmo MD5, supportato e utilizzato da Sipdroid per l’autenticazione sul server.

5.4 Installazione

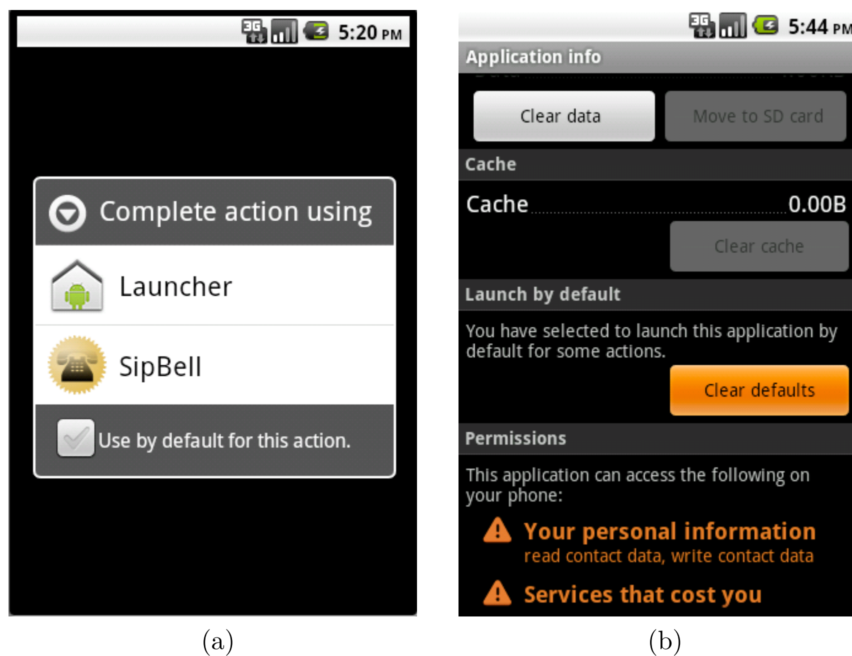
Nel realizzare il citofono sono state prese in considerazione le modalità d’installazione dell’applicazione sul dispositivo. Su questo punto si è pensato di realizzare un’applicazione che potesse soppiantare l’application home di Android, ovvero quell’applicazione che viene avviata subito dopo il completatento del boot di sistema. L’application home permette l’accesso ad altre applicazioni, al settings generale e gestisce l’interfaccia che viene lanciata nel momento in cui viene premuto il tasto **home** del dispositivo.

Un’application home, anche se con caratteristiche diverse da quella originale può essere creata manipolando il file *AndroidManifest.xml* nel seguente modo

```
<manifest xmlns:android="http://schemas.android.com/apk/res/
    android"
     [...]
    <activity android:name=".ui.Sipdroid"
          [...]
    <intent-filter>
         <action android:name="android.intent.action.MAIN" />
         <category android:name="android.intent.category.HOME
             "/>
         <category android:name="android.intent.category.
             DEFAULT"/>
    </intent-filter>
    </activity>
     [...]
</manifest>
```

In quetso caso l’activity Sipdroid viene dichiarata appartenente alla categoria delle activity HOME.

Un’applicazione con un siffatto manifest può essere eseguita solo dal sistema dopo il completamento del boot. Se all’interno del dispositivo sono installate più applicazioni con queste caratteristiche all’avvio del sistema si presenterà un’interfaccia come quella di figura 5.8 (a) che permetterà all’utente di scegliere quale home application lanciare. Se viene spuntata la casella *Use by default for*



(a) (b)

Figura 5.8: Dual boot

this action l'applicazione scelta verrà utilizzata dal sistema di default. Nel caso in cui si dovesse scegliere come default home un'applicazione diversa dalla home originale sarà possibile tornare alle impostazioni precedenti premendo il tasto **Clear Defaults** (figura 5.8 (b)) all'interno del menu *Applications/Manage Applications*.

Si era anche pensato di preinstallare Sipbell nello stack Android durante la compilazione del sistema operativo. Questo può essere fatto aggiungendo la cartella SipBell contenente l'apk, la cartella libs con le librerie .so e il file Android.mk all'interno della directory `/opt/myandroid/packages/apps`. Il file Android.mk deve contenere le seguenti stringhe

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE := SipBell
LOCAL_SRC_FILES := $(LOCAL_MODULE).apk
LOCAL_MODULE_CLASS := APPS
LOCAL_MODULE_SUFFIX := $(COMMON_ANDROID_PACKAGE_SUFFIX)
include $(BUILD_PREBUILT)

include $(CLEAR_VARS)
LOCAL_MODULE := libbv16_jni.so
LOCAL_MODULE_CLASS := SHARED_LIBRARIES
```

```
LOCAL_MODULE_PATH := $(TARGET_OUT_SHARED_LIBRARIES)
LOCAL_SRC_FILES := libs/armeabi/$(LOCAL_MODULE)
include $(BUILD_PREBUILT)

include $(CLEAR_VARS)
LOCAL_MODULE := libg722_jni.so
LOCAL_MODULE_CLASS := SHARED_LIBRARIES
LOCAL_MODULE_PATH := $(TARGET_OUT_SHARED_LIBRARIES)
LOCAL_SRC_FILES := libs/armeabi/$(LOCAL_MODULE)
include $(BUILD_PREBUILT)

include $(CLEAR_VARS)
LOCAL_MODULE := libgsm_jni.so
LOCAL_MODULE_CLASS := SHARED_LIBRARIES
LOCAL_MODULE_PATH := $(TARGET_OUT_SHARED_LIBRARIES)
LOCAL_SRC_FILES := libs/armeabi/$(LOCAL_MODULE)
include $(BUILD_PREBUILT)

include $(CLEAR_VARS)
LOCAL_MODULE := libOSNetworkSystem.so
LOCAL_MODULE_CLASS := SHARED_LIBRARIES
LOCAL_MODULE_PATH := $(TARGET_OUT_SHARED_LIBRARIES)
LOCAL_SRC_FILES := libs/armeabi/$(LOCAL_MODULE)
include $(BUILD_PREBUILT)

include $(CLEAR_VARS)
LOCAL_MODULE := libsilk8_jni.so
LOCAL_MODULE_CLASS := SHARED_LIBRARIES
LOCAL_MODULE_PATH := $(TARGET_OUT_SHARED_LIBRARIES)
LOCAL_SRC_FILES := libs/armeabi/$(LOCAL_MODULE)
include $(BUILD_PREBUILT)

include $(CLEAR_VARS)
LOCAL_MODULE := libsilk16_jni.so
LOCAL_MODULE_CLASS := SHARED_LIBRARIES
LOCAL_MODULE_PATH := $(TARGET_OUT_SHARED_LIBRARIES)
LOCAL_SRC_FILES := libs/armeabi/$(LOCAL_MODULE)
include $(BUILD_PREBUILT)

include $(CLEAR_VARS)
LOCAL_MODULE := libsilk24_jni.so
LOCAL_MODULE_CLASS := SHARED_LIBRARIES
LOCAL_MODULE_PATH := $(TARGET_OUT_SHARED_LIBRARIES)
```

```
LOCAL_SRC_FILES := libs/armeabi/$(LOCAL_MODULE)
include $(BUILD_PREBUILT)

include $(CLEAR_VARS)
LOCAL_MODULE := libspeex_jni.so
LOCAL_MODULE_CLASS := SHARED_LIBRARIES
LOCAL_MODULE_PATH := $(TARGET_OUT_SHARED_LIBRARIES)
LOCAL_SRC_FILES := libs/armeabi/$(LOCAL_MODULE)
include $(BUILD_PREBUILT)
```

nelle quali si specificano i moduli che si vogliono installare e il tipo di classe alla quale appartengono (app, shered library).

Per includere l'applicazione nello stack Android occorre modificare il file */home/<username>/opt/myandroid/build/target/product/core.mk* eliminando l'applicazione Launcher all'interno della sezione PRODUCT_PACKAGES e il file */home/<username>/opt/myandroid/build/target/product/generic.mk* aggiungendovi, sempre all'interno della sezione PRODUCT_PACKAGE l'applicazione SipBell. Una ricompilazione di Android produrrà un stack privo dell'applicazione Launcher che sarà sostituita dall'applicazione SipBell.

Tra i due tipi di installazione si è infine preferito utilizzare il primo perchè consente una gestione indipendente dello stack Android e dell'applicazione SipBell, la quale può essere aggiornata o sostituita con una versione più recente senza ricorrere alla ricompilazione di Android. Inoltre in questo modo viene aumentata notevolmante la portabilità del software che dipende esclusivamente dal sistema operativo installato e non dall'hardware sottostante fatta eccezione per le librerie conivise (compilate con l'NDK per un'architettura ARM).

5.5 Ulteriori caratteristiche

Schermo Per facilitare l'interazione dell'utente con il citofono senza intaccare il risparmio energetico si è deciso di mantenere lo schermo attivo per tutta la durata di una telefonata. Questo permette all'utente di poter visualizzare tutti i messaggi relativi allo stato di una telefonata senza dover intervenire sui pulsanti posti sul dispositivo.

Per una configurazione ottimale si può impostare uno Screen Timeout pari a trenta secondi all'interno del menu *System Settings/Sound e Display*. In questo modo lo schermo rimarrà acceso durante una chiamata ma si spegnerà subito dopo la terminazione della stessa preservando il risparmio energetico. Il tutto è stato implementato modificando il metodo **onResume()** della classe **Sipdroid** nel seguente modo:

```
@Override
```

```
public void onResume() {
    [...]
    switch(Receiver.call_state){
        case UserAgent.UA_STATE_IDLE:
            Sipdroid.layout.setKeepScreenOn(false);
            [...]
        break;
        case UserAgent.UA_STATE_OUTGOING_CALL:
            Sipdroid.layout.setKeepScreenOn(true);
            [...]
        break;
    [...]
}
```

Il monitor viene attivato quando il citofono si trova nello stato OUTGOING_CALL e disattivato nello stato IDLE.

Audio Per quato riguarda l'audio è stata attivata di default la modalità vivavoce. In questo modo l'uscita selezionata per il segnale sonoro è quella delle casse di sistema, di solito utilizzate per l'ascolto della musica o per le suonerie, al posto dell'altoparlante standard che ha un guadagno inferiore precludendo così un'ottima ricezione del flusso sonoro. Questo è stato implementato nella classe **Receiver** modificando nel modo seguente in metodo **speakermode()**:

```
public static int speakermode() {
    return AudioManager.MODE_NORMAL;
}
```

Chiamate in ingresso É stata inltre disabilitata la chiamata verso il citofono con la modifica del metodo **onState()** della classe **Receiver** e del metodo **startEngine()** della classe **SipdroidEngine** commentando la funzione **listen()**

```
public static void onState(int state,String caller) {
    [...]
    case UserAgent.UA_STATE_IDLE:
    [...]
        mContext.startActivity(createIntent(Sipdroid.class))
            ;
        //engine(mContext).listen();
    break;
    [...]
}
```

```
public boolean StartEngine() {
     [...]
     register();
     //listen();
     } catch (Exception E) { }
          return true;
}
```

Skype É possibile utilizzare il citofono per contattare utenti Skype. Gli sviluppatori di Sipdroid, per questioni di compatibilità, consigliano di servirsi del server **pbxes.org** per la registrazione. Questo server integra un servizio che permette ai client sip di contattare un client skype prescelto utilizzando come URI da contattare un indirizzo della forma <Contatto_Skype>@skype. La registrazione al server, per le funzioni base, è gratuita e permette la configurazione e gestione fino a cinque interni.

In alternativa, è possibile integrare all'interno del prorio server Asterisk il gateway SipToSis. Questo si basa su un'architettura costituita da un client SIP realizzato con lo stack MjSip, un client skype e le API skype. Il client SIP del gateway viene registrato sul server Asterisk ed ha il compito di sfruttare le API skype per trasformare il signaling SIP in signaling Skype.

Capitolo 6

Conclusioni

Questo lavoro di tesi ha riguardato la realizzazione di un terminale audiocitofonico SIP installato su una piattaforma Android. Il sistema risulta essere facilmente configurabile e soprattutto facilmente utilizzabile dall'utente finale. Questo aspetto risulta essere estremamente importante nella realizzazione di un software che verrà utilizzato da un target di utenti molto vasto che comprende anche persone non abituate all'utilizzo delle moderne tecnologie come gli smartphone.

Uno schermo touch screen può a volte rappresentare un ostacolo da non sottovalutare per un utilizzatore alle prime armi ed è per questo che l'attenzione nella progettazione del software si è spostata sulla costruzione di un'interfaccia semplice ed intuitiva. La visualizzazione su schermo di un unico bottone con su scritto il nome del proprietario dell'abitazione porterà l'utilizzatore inesperto a premerlo per effettuare la chiamata. L'inoltro della chiamata e l'accettazione della stessa verranno notificate all'utilizzatore attraverso delle comode finestre di dialogo che permetteranno all'utente, in qualsiasi momento, di chiudere la conversazione. La configurazione dell'apparecchio ricalca la configurazione di un qualsiasi client SIP con l'aggiunta di un utile menu per editare il settings di Android, agendo per esempio sulla politica del risparmio energetico, problematica molto importante per un sistema "always on" alimentato ventiquattro ore su ventiquattro da una rete elettrica.

Per prevenire manomissioni, il sistema è stato protetto da un'autenticazione basata su password che permette un buon livello di sicurezza se affiancata da una gestione dei permessi oculata sull'accesso alle risorse della singola applicazione e da un sistema di intrusion detection che previene il tipo di attacco *brute force*. Diversamente dalle normali piattaforme desktop, su Android viene assegnato uno user ID ad ogni applicazione nel momento in cui questa viene installata. Questo valore, espresso in forma numerica, rimarrà costante per tutta la durata della vita dell'applicazione all'interno della piattaforma. I dati

salvati da un'applicazione con un determinato user ID possono essere letti o scritti soltanto dalla medesima applicazione.

I tentativi di intrusione con attacchi di tipo forza bruta vengono rilevati dal sistema e comunicati al proprietario del citofono attraverso un alert call identificabile da un caller id diverso da quello utilizzato per le chiamate standard. Durante un alert call, lo schermo non presenta bottoni con cui interagire, permettendo un blocco temporaneo del sistema che può tornare ad una situazione normale con la chiusura della telefonata da parte del proprietario del citofono, al quale si dà il tempo di prendere opportune decisioni.

Il sistema presenta un tipo di comunicazione basato unicamente su un flusso di segnali audio ma può essere esteso per includere anche un servizio video. Sipdroid supporta nativamente la videochiamata che può essere attivata dall'utente durante una conversazione. L'attivazione automatica di una video chiamata permette di poter dotare il sistema di una nuova funzione: lo spioncino. L'utente in questo caso può effettuare una chiamata verso il citofono il quale risponderà automaticamente inoltrando all'utente il flusso audio e video catturato dal microfono e dalla videocamera. L'utente può vedere quello che accade fuori dalla sua abitazione in qualsiasi posto egli sia. Se dotato di sensori può essere il citofono stesso ad inoltrare una chiamata se allertato da un rumore strano o da movimenti sospetti nei pressi dell'abitazione.

La configurazione del settings prevede che l'utente interagisca direttamente con il citofono. La configurazione può avvenire anche da remoto installando su Android un server web che permetta, previa autenticazione, il settaggio della configurazione con l'utilizzo, per esempio, di form html simili a quelle utilizzate per la configurazione di un moderno modem/router.

L'utilizzo di Android come piattaforma base predispone il sistema ad una vasta personalizzazione che può avvenire anche con l'installazione di nuove applicazioni con le quali Sipbell può interagire in modo da rendere il sistema adeguato alle esigenze di un qualsiasi cliente.

Bibliografia

[1] **Introduzione ad Android**

dal sito ufficliale per gli sviluppatori di Android

http://developer.android.com/guide/basics/what-is-android.html

[2] **Architettura di Android**

dal libro *Android: guida per lo sviluppatore* di Massimo Carli

[3] **Sviluppare applicazioni su Android**

dal libro

Hello, Android Introducing Google's Mobile Development Platform

Second Edition, Ed Brunette

[4] **Android Software Development Kit**

http://developer.android.com/guide/developing/tools/index.html

[5] **Android Native Development Kit**

Sito internet *http://developer.android.com/sdk/ndk/index.html*

Come utilizzare l'NDK *File Readme.txt all'interno nell'NDK*

[6] **Introduzione all'application processor I.MX51**

Documento *i.MX51 Applications Processors for Consumer and Industrial Products*

scaricabile all'indirizzo *www.freescale.com*

[7] **Caratteristiche hardware della board**

Documento *i.MX51 EVK Hardware User's Guide*

scaricabile all'indirizzo *http://www.freescale.com*

[8] **Installare Android sulla I.MX51**
dalla pagina html *i_MX_Android_R7_User_Guide.html*
presente all'interno del file *imx-android-r7.tar.gz*
scaricabile all'indirizzo *www.freescale.com*

[9] **Programmare l'U-boot**
Documento *U-Boot Quick Reference for the Lite5200B Development Platform*
presente all'interno del file *imx-android-r7.tar.gz*
scaricabile all'indirizzo *www.freescale.com*

[10] **Installazione della board I.MX51**
Documanto *i.MX51 EVK Android Quick Start Guide*
presente all'interno del file *imx-android-r7.tar.gz*
scaricabile all'indirizzo *www.freescale.com*

[11] **Linux arm boot process**
dal sito *www.simtec.co.uk/products/SWLINUX/files/booting_article.html*

[12] **SIP: Session Initiation Protocol**
dalla rfc 3261 scaricabile all'indirizzo *www.ietf.org/rfc/rfc3261.txt*

[13] **Libreria MjSip**
documento *mjsip_minitutorial_01.pdf*
scaricabile all'indirizzo *www.mjsip.org*

[14] **Sipdroid Free SIP/VoIP client for Android**
informazioni generali reperibili sul sito *sipdroid.org*

[15] **Introduzione ad Asterisk**
dal libro *Asterisk e Dintorni La guida italiana al VoIP Open Source* di Diego Gosmar, Giuseppe Innamorato, Dimitri e Stefano Osler

[16] **Configurare un dialplan con Asterisk**
dal sito *http://www.voip-info.org/wiki/view/Asterisk*