ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
Corso di Laurea Triennale in Informatica

Pjproject su Android: uno scontro su più livelli

Tesi di Laurea in Architettura degli Elaboratori

Relatore:
Chiar.mo Prof.
Ghini Vittorio

Presentata da:
Bergami Giacomo

Sessione II
Anno Accademico 2011-2012

Indice

Non senza fatiga si giunge al fin.

G. Frescobaldi
"Toccata Nona" dalle «Toccate e partite d'intavolatura di cimbalo, Libro Secondo»

CAPITOLO 1

Premesse

Indice

1.1. Introduzione

L'uso in costante aumento di dispositivi mobili computerizzati spinge ogni sviluppatore a domandarsi quali possano essere le loro applicazioni pratiche; conseguentemente conoscendone potenzialità e caratteristiche tecniche, si chiede se gli stessi programmi che utilizza quotidianamente, possano essere utilizzati senza la necessità - e la voglia - di reinventare la ruota.

Questo desiderio, spinto dalla necessità pratica, lo orienta a scegliere sistemi di sviluppo che gli permettano di indagare le problematiche ed approfondirne le soluzioni accreditate: per questo le "piattaforme di sviluppo" che si vanno via via affermando - se non già consolidando - sono costituite dai sistemi operativi GNU/Linux ed Android.

La seconda pone delle prosepettive interessanti, vuoi per l'utilizzo di una versione del Kernel ottenuta dal *fork* della prima, vuoi per il crescente interesse nel mondo delle imprese, vuoi per le limitazioni architetturali imposte dagli stessi sviluppatori. Queste ultime consentono agli hacker di sfruttare - vedi il rooting - le conoscenze sulle debolezze dei sistemi GNU/Linux allo scopo di superare tali restrizioni, consentendo all'appassionato di conoscere meglio il mondo GNU/Linux tramite l'analisi delle differenze architetturali tra i due sistemi.

Entrando ora nel merito di questa Tesi di Laurea, il tentativo di *porting* di Pjproject - di per se già iniziato dalla comunità di sviluppatori, ma non ancora portato a termine durante la redazione della presente - ha permesso di conoscere sia l'oggetto iniziale di analisi, sia di scoprire nuovi aspetti sul *porting* e sulla piattaforma Android. Sebbene esistano tentativi di *porting* basati sull'ideazione di Applicazioni Java o Applicazioni Native con JNI, lo scopo perseguito è quello di utilizzare il linguaggio C, allo scopo di interagire direttamente con il livelli

architetturali il più possibile vicini al Kernel e di distaccarsi dall'emulazione dei programmi tramite la DALVIK VIRTUAL MACHINE.

Questa tesi vorrà mostrare come questo tentativo non consenta l'evasione completa dalla "sandbox" imposta da Google - benché questa si ottenga parzialmente tramite tecniche di rooting - in quanto le librerie che vengono rese disponibili all'interno dei dispositivi interagiscono direttamente con gli stati di controllo, che limitano la configurabilità e l'utilizzo degli stessi.

Voglio inoltre mostrare quale sia l'architettura interna di tale sistema operativo, senza tuttavia perdere di vista le difficoltà riscontrate nello svolgimento del progetto, dovute sia all'immaturità del progetto Pjproject, sia sul mancato testing dello stesso su alcuni aspetti, sia da mie considerazioni iniziali errate, che sono state corrette durante il processo di analisi.

Questa sarà inoltre l'occasione per discorrere su quali strumenti utilizzare - ed in quale modo - allo scopo di ottenere i risultati che mostrerò via via.

1.2. Terminologia adottata all'interno della tesi

È necessario definire a priori una terminologia, onde chiarire alcune termini ambigui. Per quanto riguarda le applicazioni, definisco:

Applicazioni Java: Questo genere di applicazioni sono quelle più diffuse all'interno del mondo Android: tutto il codice è scritto in linguaggio Java, compreso l'accesso ai servizi di sistema, ed è necessario stabilire tramite Android Manifest file i permessi che si ritiene opportuno utilizzare. Per la compilazione è sufficiente disporre dell'Android SDK.

Applicazioni Native con JNI: In genere si fa riferimento a questo tipo di applicazioni come ad "Applicazioni Native" assieme alle successive [vedi Sil09, pp. 27-54] anche se, in questo caso, si accede alla Java Native Interface allo scopo di interagire con il codice Java. La compilazione di queste applicazioni avviene tramite lo script `ndk-build`, in ogni caso, viene sempre prodotto un file apk, e vengono stabiliti i permessi di accesso nel Manifest. Per la compilazione è necessario disporre dell'Android NDK.

Applicazioni Native: Con questo termine faccio riferimento unicamente alle applicazioni compilate in codice binario, ed eseguibile direttamente dal processore. In genere si effettua la compilazione dei binari tramite Android NDK, ma è possibile utilizzare altri tool per il *crosscompiling* preesistenti quali CodeSourcery Lite o la creazione di nuovi, in particolare tramite crosstool-ng.

Si utilizzerà invece il termine generico di *applicazione* qualora si faccia riferimento indistintamente a tutte le tipologie di applicazione di cui sopra.

Kernel Android: Con questo termine si fa riferimento alla componente del sistema operativo che comprende la versione modificata del Kernel Linux, facilmente ottenibile dal repository Git `https://android.googlesource.com/kernel/common`.

AOSP Source: Con questo termine (acronimo di ANDROID OPEN SOURCE PROJECT), si intende la sovrastruttura costruita al di sopra del Kernel Android. Questo sorgente tuttavia non include lo strato del Kernel Android, come tra l'altro provato dalla Figura 3.1.1 a pagina 22.

Questi sorgenti mettono a disposizione le librerie di sistema per le applicazioni native in genere, le API per le applicazioni Java ed i *frameworks* (v. 3.1) per gestire lo stato del sistema: in genere con il termine **Android Middleware** [Ong+09] si fa riferimento all'insieme delle *features* messe a disposizione dall'AOSP Source, e che forniscono alle applicazioni in esecuzione un'astrazione del sistema operativo.

Con il termine *host* intendo il dispositivo sul quale si sta effettuando il processo di cross-compilazione per la macchina di destinazione, detta appunto *target*.

Con il termine *upsyscall* (o *Up System Call*) mi riferirò alla possibilità di richiamare, da strati di codice più elementari come quelli preposti dal codice nativo ed in esecuzione nello *user space*, funzioni fornite all'interno di strati di sistema maggiormente elaborati, come ad esempio il codice Java in esecuzione all'interno di una macchina virtuale.

Con il termine *rooting* intendo la procedura necessaria all'interno dei dispositivi Android per ottenere i privilegi di superutente, in modo da impedire alcune operazioni che debbono essere consentite previa richiesta di adeguati permessi.

Con il termine *service* intendo degli oggetti Java o C++ contenenti delle collezioni di metodi che, tramite l'interazione con altri service o con le librerie native di sistema, forniscono un supporto all'interazione con il dispositivo.

1.3. Dispositivi adottati in fase di testing e sviluppo

Elenco di seguito quali sono stati gli strumenti utilizzati all'interno della tesi:

Emulatore Android SDK: Si è tentato lo sviluppo all'interno di una macchina con versione 4.0 del sistema operativo Android. Si è tuttavia riscontrato che questo non supporta il kernel Linux nella versione 3.x, benché tutti i dispositivi in commercio con quella versione ne siano provvisti.

Su questo dispositivo si parlerà nei seguenti frangenti:

- ◇ *Procedura di Rooting*: v. Sottosezione 5.2.1 a pagina 75.

- ◇ *Comunicazione tra Emulatori Android*: v. Sottosezione 4.2.1 a pagina 68.
- ◇ *Motivazioni per le quali tale dispositivo non è stato ritenuto adatto al testing di Pjproject*: v. Sottosezione 7.4.1 a pagina 101.

Olivetti Olitab 110 (Android 3.1): Si è effettuato in una prima fase testing su di questo dispositivo: il suo aggiornamento da parte del *vendor* con le librerie OpenSL-ES, ha consentito il testing dell'applicazione `pjsua` su due dispositivi distinti: tale libreria compare infatti nelle *platforms* dell'NDK solamente a partire dalla versione 14 delle API di Android.

Su questo dispositivo si parlerà nei seguenti frangenti:

- ◇ *Procedura di Rooting*: v. Sottosezione 5.2.3 a pagina 77.
- ◇ *Motivazioni per le quali tale dispositivo non è stato ritenuto adatto al testing di Pjproject*: v. Sottosezione 7.4.2 a pagina 101.

Samsung Galaxy Nexus (Android 4.1): In virtù di quanto affermato precedentemente, si è reso quindi necessario spostare l'attenzione sul questo dispositivo, il quale supporta una versione 3.x del Kernel Linux. In quanto è un "Developer Phone", è già previsto un comando per effettuare l'*unlocking* del bootloader allo scopo di eseguire il *ramdisk*. Per altri dispositivi si rende invece necessario sfruttare vulnerabilità del Kernel, di fastboot o nella comunicazione tra computer e dispositivo [Bia10].

Su questo dispositivo si parlerà nei seguenti frangenti:

- ◇ *Procedura di Rooting*: v. Sottosezione 5.2.2 a pagina 75.
- ◇ *Procedura di Flashing*: v. Sottosezione 3.1.2 a pagina 25.

FIGURA 1.3.1. *Strumenti utilizzati per la realizzazione della tesi.*

CAPITOLO 2

Programma ed illustrazione

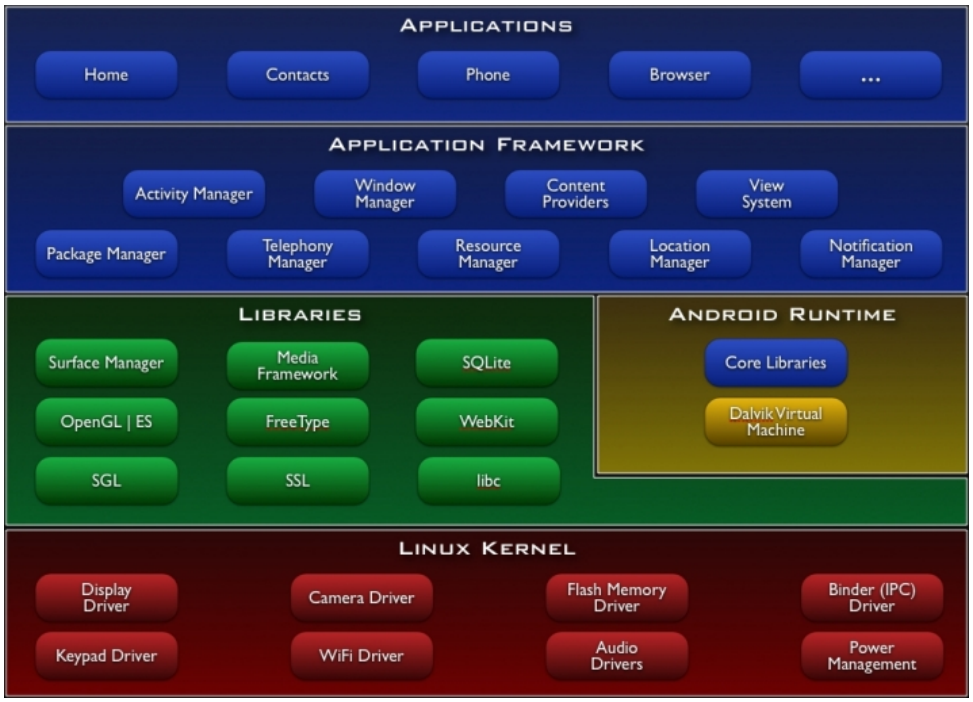
Indice

2.1. Architettura Android

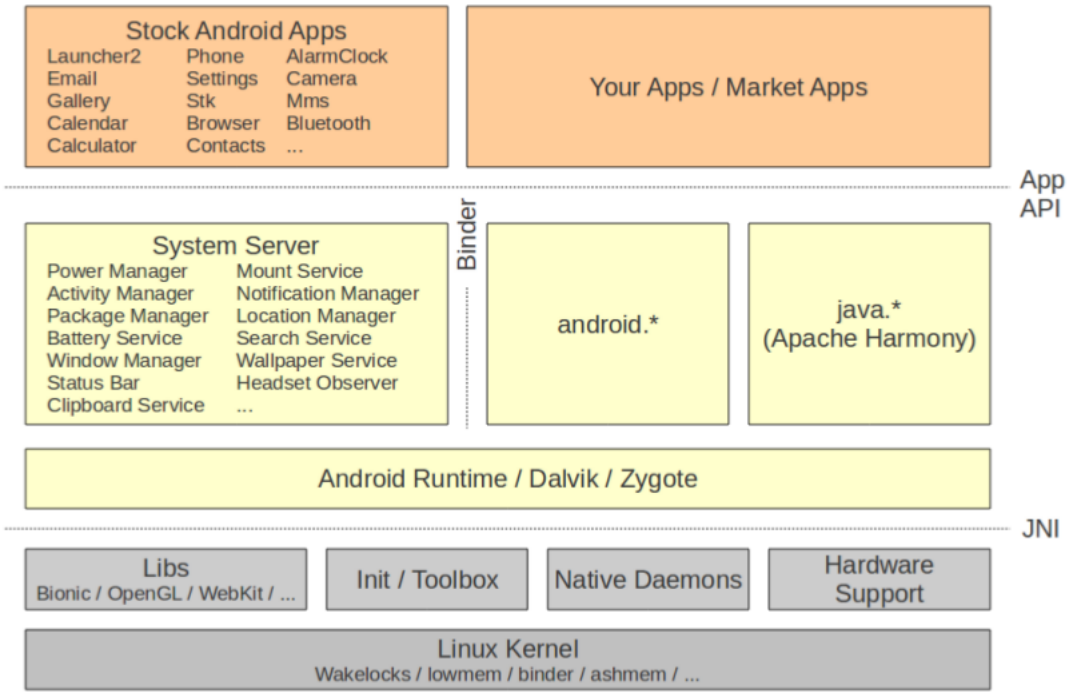
Il *porting* dell'applicazione nativa `pjsua` di Pjproject, oltre ai problemi di cross-compilazione descritti in Sezione 7.1 a pagina 89, ha permesso di riscontrare le limitazioni sulla gestione delle periferiche audio (ovvero l'impossibilità di effettuare l'accesso al mirofono da parte di due istanze di applicazioni native: v. Sezione 7.3 a pagina 98) ed i controlli dei permessi utente, mettendo quindi in luce la differenza strutturale della gestione dei permessi tra i sistemi Android e quelli GNU/Linux propriamente detti (Sezione 3.3 a pagina 30). Di fatti, come dice lo stesso [PP10], Android è una versione del Kernel Linux che ha subito sostanziali modifiche, quali l'incorporazione di *udev* dentro ad *init*.

Per indagare come questo sia possibile è quindi opportuno analizzare la composizione dei *layer* software forniti da Android: in genere ci si riferisce all'architettura di tale sistema con la descrizione in Figura (a) 2.1.1 nella pagina successiva. Questa visualizzazione tuttavia esclude del tutto dalla gerarchia le applicazioni native, mostrando unicamente la gerarchia logica-funzionale necessaria allo sviluppatore Android Java, allo scopo di gestire unicamente l'interazione tra le applicazioni (il primo strato in alto) ed i *service* (quelli immediatamente sotto). Nell'immagine inoltre non viene affatto mostrato come questo meccanismo avvenga: come mostra il livello di strutturazione del sorgente proposto in Figura (b) 2.1.1 nella pagina seguente, le applicazioni native sono effettivamente situate allo stesso livello delle altre applicazioni di sistema, in quanto anch'esse debbono scontrarsi con *Android Middleware*, che regola anche il funzionamento degli stessi applicativi.

L'esistenza dello strato dei *service* e la possibilità che questi ultimi siano effettivamente fruibili dai processi di sistema, sono garantiti dalla presenza del **Binder**, un driver virtuale implementato a livello Kernel e che verrà descritto in seguito nel dettaglio (Sezione 3.5 a pagina 35); per ora è sufficiente sapere che questo meccanismo implementa il sistema utilizzato da Android per effettuare l'IPC tramite il device Binder per consentire l'interazione tra programmi



(A) *Visione High-Level dell'Architettura di Android.* `http://androidteam.googlecode.com/files/system-architecture.jpg`



(B) *Visione Low-Level dell'Architettura di Android. [Yag11]*

FIGURA 2.1.1. Vari livelli della visualizzazione dell'Architettura di Android.

e servizi.

Si può inoltre osservare come la registrazione di tali *service* venga effettuata direttamente nei rispettivi linguaggi di definizione, ovvero sia in Java, sia in C++ (Sottosezione 3.5.1 a pagina 40). A questo punto mentre risulta naturale intuire come ciò sia possibile da codice nativo (ovvero tramite l'utilizzo indiretto delle system call per l'interazione con i device), non è immediato comprendere come sia possibile effettuare la registrazione di *service* Java e l'interazione con questi.

A questo scopo i progettisti Android hanno dovuto ricorrere all'utilizzo della JAVA NATIVE INTERFACE (JNI) per permettere l'interazione tra Java e librerie native (Sezione 3.2 a pagina 27) mediante uno strato di *Adapter* predisposti dalle librerie del C++ Middleware. Tuttavia l'introduzione di questo strato necessita di una preparazione preventiva la riscrittura della JAVA VIRTUAL MACHINE nella DALVIK VIRTUAL MACHINE (Sezione 3.3 a pagina 30).

Una panoramica sull'inizializzazione di sistema è fornita dalla Figura 2.1.4 a pagina 18 dove sia l'inizializzazione della DVM, sia quella del `media_server`, avvengono tramite lo script `init.rc`. Come si può vedere dall'immagine, dopo aver avviato la DVM si inizia l'esecuzione di **Zygote**: questo è un demone il cui compito principale è quello di iniziare il processo **System Server**, che a sua volta inizializza i principali *service* Java. Si può inoltre notare come detto Zygote gestisca l'avvio delle applicazioni Java e Native con JNI mettendosi in ascolto delle richieste sul socket `/dev/socket/zygote` [Yag11].

In Figura 2.1.4 a pagina 18 mostro l'interazione occorrente tra librerie di sistema e *service* per consentire il campionamento audio, analizzando solo una parte dell'architettura messa in evidenza precedentemente: si può notare come il processo `media_server` inizializza i *service* **AudioPolicyService** che predispone l'interazione con i driver audio ed **AudioFlinger**, che in particolare si occupa della gestione dei permessi per l'accesso al sampling audio. Questi due servizi vengono quindi acceduti da processi nativi tramite un'istanza della classe **AudioRecorder** della libreria `libmedia`, che viene a sua volta utilizzato dalla libreria Wilhelm, implementazione delle API OpenSL-ES per l'interazione con le periferiche audio. In questo modo posso quindi mostrare come quest'ultima interagisca direttamente con i servizi Java per il controllo dei permessi utente (Sottosezione 3.6.2 a pagina 59) per la mediazione della gestione delle periferiche. Questi *service* quindi impediscono ai processi concorrenti nativi l'accesso al campionamento audio del microfono, non prevedendo una politica di caching delle informazioni pervenute dal microfono. Per la gestione della politica dei permessi predisposta dalle stesse librerie di sistema, si rende necessario l'utilizzo di tecniche di rooting per bypassare i suddetti controlli (Sezione 5.2 a pagina 74).

Come daltronde evidenziato in [Bia10], è sempre a livello Java che avviene la limitazione nella selezione dell'interfaccia di rete di default, non rendendo

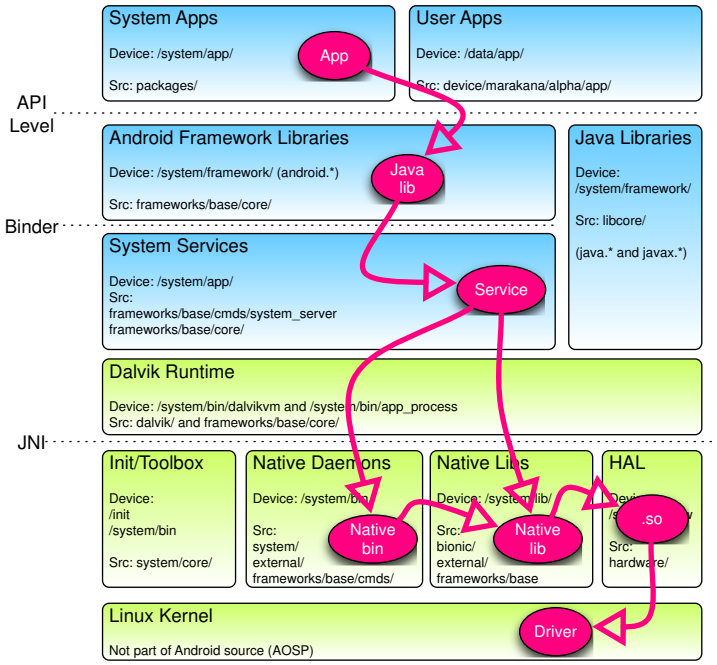


FIGURA 2.1.2. *Visione dello stack di Android dal punto di vista dell'interazione tra strati architetturali. [Gar07]*).

possibile, a meno di modifica dello strato dei servizi, l'utilizzo contemporaneo di due interfacce. Una visione generale di questo meccanismo, suggerirebbe la ricompilazione di alcune librerie già disponibili in Android allo scopo di bypassare suddetti controlli.

Un'*overview* dell'interazione dei vari strati occorrente nell'esecuzione di applicativi Java è evidenziato in Figura 2.1.2, che ben descrive la situazione generale del comportamento del sistema operativo.

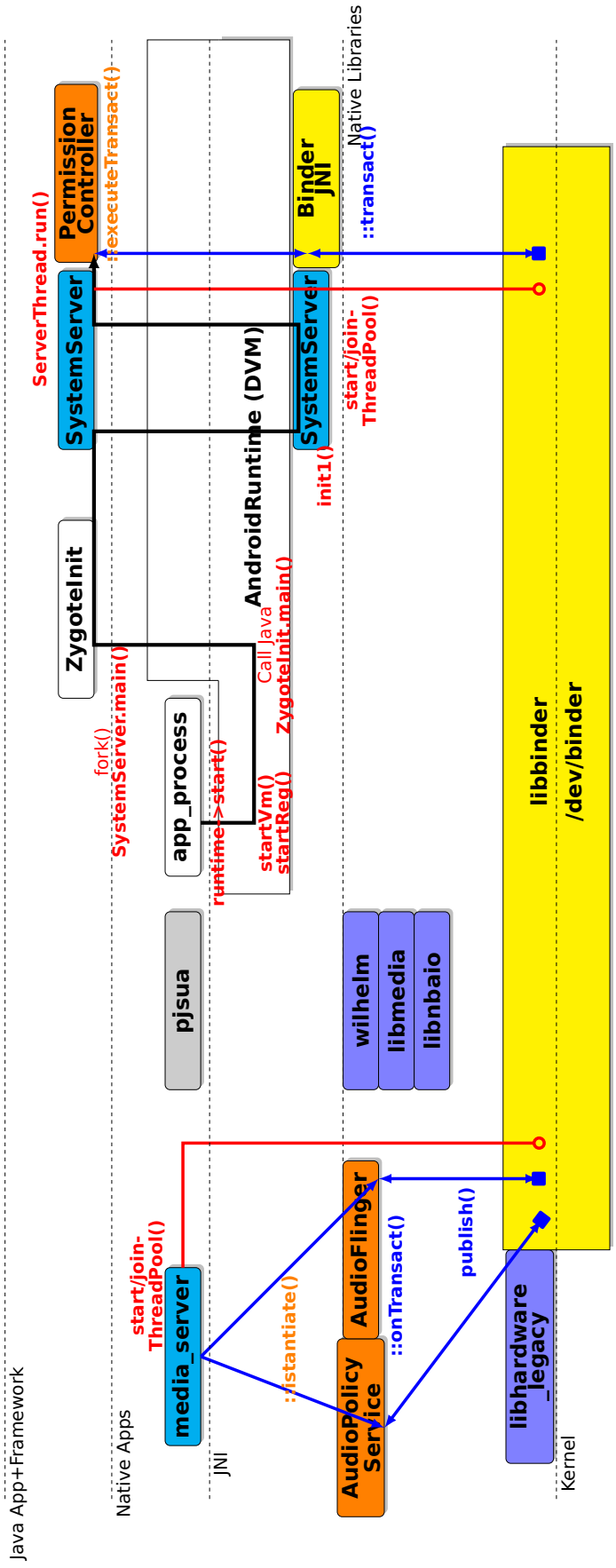


FIGURA 2.1.3. *Inizializzazione del sistema.*

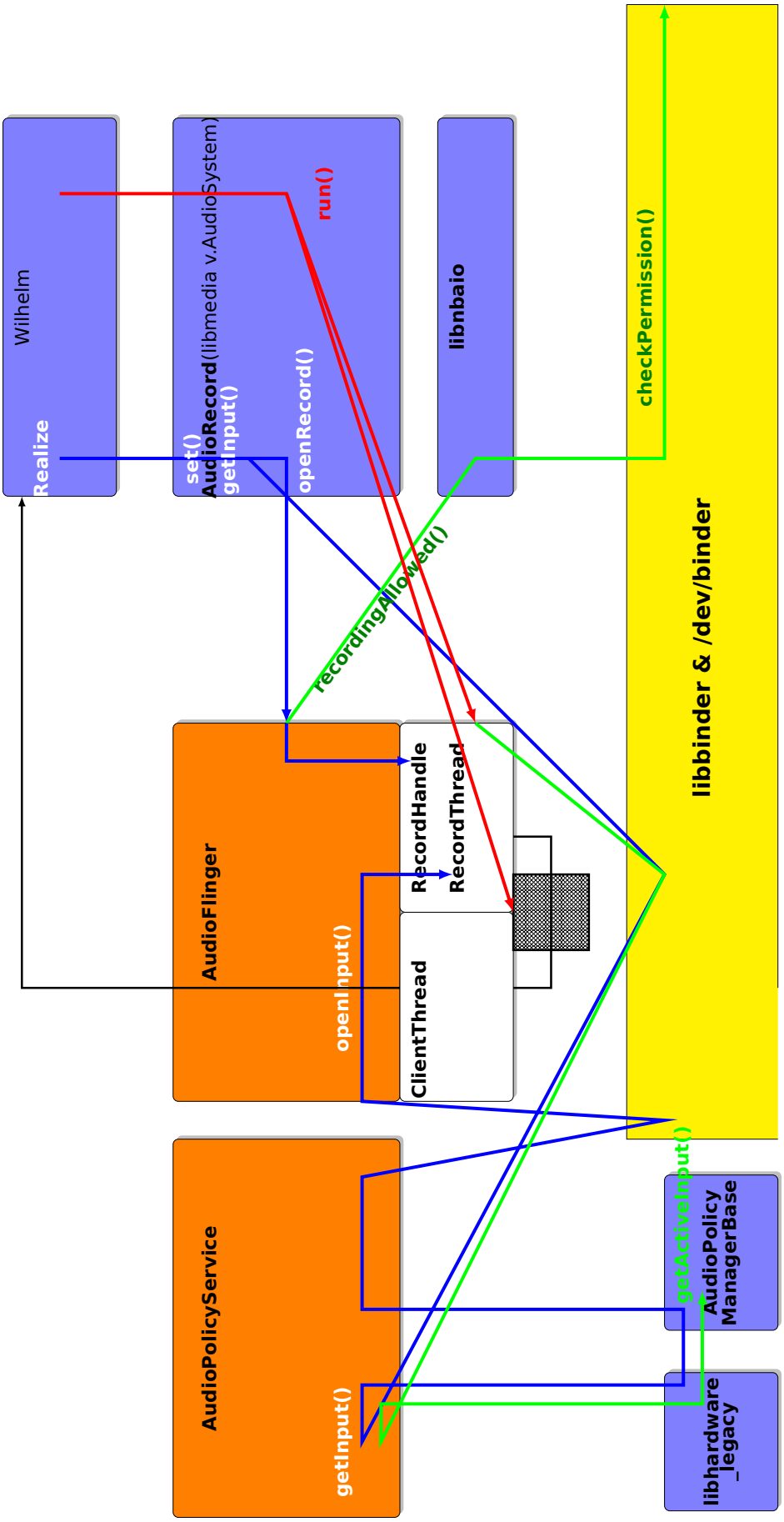


FIGURA 2.1.4. *Interazione tra servizi di sistema per la registrazione dell'audio.*

Parte 1

Esposizione

CAPITOLO 3

Architettura Android

Fully understanding the internals of Android's system services is like trying to swallow a whale

K. Yaghmour
Embedded Android

Indice

Lo scopo di questo capitolo è quello di dettagliare come avvengano le interazioni all'interno del sistema operativo, allo scopo di mostrare nei capitoli successivi come sia possibile l'interazione tra codice nativo e servizi di sistema.

3.1. Struttura del sorgente Android

Possiamo avvicinarci alla struttura dei sorgenti Android guardando alla Figura 3.1.1 nella pagina seguente: come possiamo vedere, la parte del Kernel Android non è inclusa all'interno del sorgente AOSP di Android, che contiene unicamente i sorgenti per l'*Android Middleware*. É sempre possibile reperire la versione del Kernel più appropriata al dispositivo, scegliendo tra le seguenti varianti (`$PATH`); tra le più rappresentative:

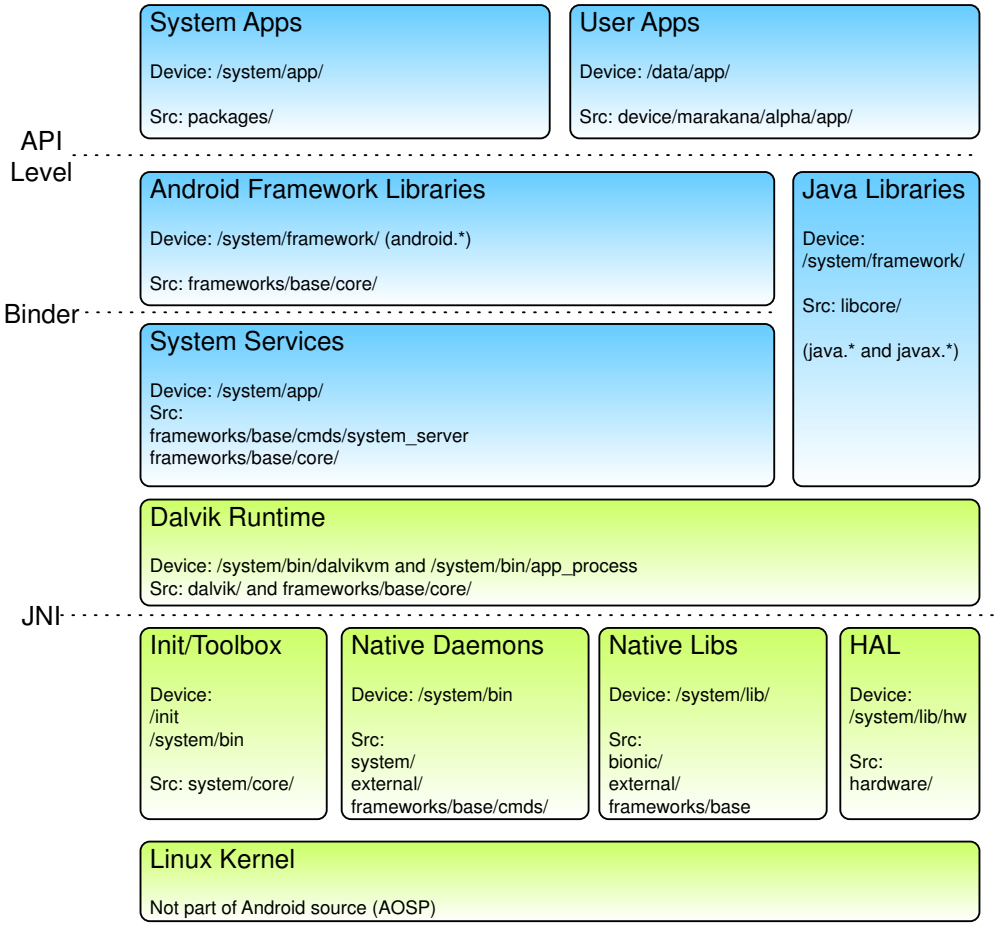


FIGURA 3.1.1. Visualizzazione dell'Architettura di Android dal punto di vista dei Device e dei sorgenti. [Gar07]

kernel/common: è il branch ufficiale del Kernel Linux, che viene utilizzato come base per tutte le altre versioni seguenti.
kernel/goldfish: Kernel *tree* da utilizzare per ottenere le immagini Kernel per l'architettura dell'emulatore, detta appunto "Goldfish".
kernel/msm: Kernel *tree* da utilizzare per i Chipset Qualcomm, tra i quali annoveriamo il Nexus One [Bia10].
kernel/samsung: Kernel *tree* per i sistemi Samsung.
kernel/tegra: Kernel *tree* per i chipset Tegra.

Per scaricare tali versioni del kernel, è sufficiente postporre i percorsi di cui sopra al seguente comando:

```
git clone https://android.googlesource.com/$PATH
```

Ricordo che, dopo aver effettuato il `clone` sul *repository*, è possibile visualizzarne i branch remoti tramite il comando `git branch -a`, e quindi effettuare il

`checkout` dei files per poterli visualizzare sul computer locale:

```
git checkout remotebranchpath
```

Tuttavia non è nell'interesse di questa tesi descrivere in un modo più approfondito quale sia la struttura dei sorgenti del Kernel, in quanto mi concentrerò maggiormente sull'*AOSP Source*, il quale ci permetterà di generare il *ramdisk* alla fine di effettuare il *flashing* sul dispositivo Galaxy Nexus. Andando ora ad analizzare la struttura dello AOSP Source, possiamo identificare i seguenti *folder* principali:

bionic: Contiene la libreria Bionic per il supporto della *libc*; di questa si parlerà nella Sezione 3.4 a pagina 34.
bootable: Contiene il supporto a *bootloader*, *diskinstaller*, e al *recovery image*.
build: Contiene gli script di supporto alla compilazione.
cts: Contiene la ANDROID'S COMPATIBILITY TEST SUITE, allo scopo di effettuare dei testing sul sorgente.
dalvik: Contiene la definizione della DALVIK VIRTUAL MACHINE (DVM).
development: Contiene *tool* per lo sviluppo, file di configurazione ed applicazioni d'esempio.
device: Contiene binari e sorgenti specifici per il dispositivo sul quale portare l'AOSP Source.
external: Contiene le librerie native e Java di terze parti, ottenute tramite il la sincronizzazione con *repository* remoti.
frameworks: Contiene utilità native per Android, demoni (`installd`, `servicemanager`, `system_server`), l'utilizzo di librerie (quali quelle di supporto all'architettura tramite *wrapper* JNI), l'implementazione delle API Android e dei `services`.
hardware: Fornisce le librerie di supporto all'astrazione dello hardware (HAL - HARDWARE ABSTRACTION LAYER), fornendo alcuni sorgenti ed oggetti binari.
ndk: v. Sezione 4.3 a pagina 70
out: Contiene il risultato del processo di compilazione
packages: Contiene le applicazioni e le utility di sistema comuni a tutti i dispositivi Android commercializzati.
prebuilt: Contiene binari precompilati, quali kernel ed altri binari di terze parti.
sdk: Contiene i tool per l'Android SDK (v. Sezione 4.1 a pagina 67).
system: Contiene la *root* del FileSystem di Android, alcuni "demoni nativi", la definizione di `init` ed i file di configurazione.

Identificherò la cartella che contiene tali *folder* come `$AOSP`.

3.1.1. AOSP: Configurazione dell'ambiente ed ottenimento dei sorgenti. Per la compilazione di questo sorgente non è necessario fornire, come nel caso della compilazione del kernel, un *crosscompiler*, in quanto questo verrà generato automaticamente dalla compilazione tramite i sorgenti messi a disposizione. Per rendere possibile la compilazione degli *host tool* forniti nel sorgente, è necessario effettuare l'installazione dei seguenti pacchetti:

```
sudo apt-get install git-core gnupg flex bison gperf build-essential \
  zip curl libc6-dev libncurses5-dev:i386 x11proto-core-dev \
  libx11-dev:i386 libreadline6-dev:i386 libgl1-mesa-glx:i386 \
  libgl1-mesa-dev g++-multilib mingw32 openjdk-6-jdk tofrodos \
  python-markdown libxml2-utils xsltproc zlib1g-dev:i386
```

Per risolvere inoltre un problema nella compilazione, è necessario effettuare il seguente *link*:

```
sudo ln -s /usr/lib/i386-linux-gnu/mesa/libGL.so.1 /usr/lib/i386-linux-gnu/
    libGL.so
```

Per quanto concerne invece il compilatore di Java, non si garantisce la corretta compilazione dei sorgenti tramite l'utilizzo di `IcedTea` per `OpenJDK`, implementazione GNU GPL del linguaggio Java: è necessario pertanto scaricare i binari ufficiali di Oracle al seguente indirizzo:

`http://www.oracle.com/technetwork/java/javase/downloads/jdk6-downloads-1637591.html`

Se il proprio sistema supporta già un'implementazione Open di Java, sarà successivamente necessario eseguire questi comandi allo scopo di utilizzare la versione standard scaricata dall'indirizzo di cui sopra:

```
sudo update-alternatives --install /usr/bin/java java /usr/lib/jvm/jdk1.6.0_33/
    bin/java 1
sudo update-alternatives --install /usr/bin/javac javac /usr/lib/jvm/jdk1.6.0
    _33/bin/javac 1
sudo update-alternatives --install /usr/bin/javaws javaws /usr/lib/jvm/jdk1.6.0
    _33/bin/javaws 1
sudo update-alternatives --config java
sudo update-alternatives --config javac
sudo update-alternatives --config javaws
```

Per evitare ulteriori e possibili problemi in fase di compilazione, come ad esempio *Could not stat out/target/product/generic/system*, ovvero l'assenza della cartella ove vengono immessi i binari dai quali produrre poi l'immagine del *filesystem* di sistema, è opportuno installare i seguenti programmi.

```
sudo apt-get install xmlto doxygen
```

Per poter scaricare il sorgente AOSP completo, si rivela necessario utilizzare lo script `repo`, reso disponibile dalla stessa Google all'indirizzo

`https://dl-ssl.google.com/dl/googlesource/git-repo/repo`

Per scaricare l'ultima versione del sorgente, è sufficiente eseguire i comandi che seguono da terminale all'interno della cartella `$AOSP`; mentre il primo permette l'inizializzazione del *repository*, il secondo permette il *download* effettivo dell'intero sorgente.

```
repo init -u https://android.googlesource.com/platform/manifest
repo sync
```

Leggendo sul sito attinente alla documentazione ufficiale sui sorgenti Android, `http://sources.android.com`, si osserva la seguente nota informativa:

> *Starting with Ice Cream Sandwich, the Android Open-Source Project can't be used from pure source code only, and requires additional hardware-related proprietary libraries to run, specifically for hardware graphics acceleration.*
>
> *Each set of binaries comes as a self-extracting script in a compressed archive. After uncompressing each archive, run the included self-extracting script from the root of the source tree, confirm that you agree to the terms of the enclosed license agreement, and the binaries and their matching makefiles will get installed in the vendor/ hierarchy of the source tree.*

Per quanto concerne i telefoni riconosciuti da Google come Developer Phones, è possibile scaricare gli script a loro dedicati tramite il sito:

`https://developers.google.com/android/nexus/drivers`

Tuttavia, anche in questo modo non vengono forniti tutti i driver atti ad un utilizzo completo del dispositivo[1]:

> *Camera, GPS and NFC don't work on Galaxy Nexus.*
>
> *Symptom: Camera, GPS and NFC don't work on Galaxy Nexus. As an example, the Camera application crashes as soon as it's launched.*
>
> *Cause: Those hardware peripherals require proprietary libraries that aren't available in the Android Open Source Project.*
>
> *Fix: None.*

Per altri dispositivi, quali il Samsung Galaxy SIII, è la stessa Samsing a rilasciare i sorgenti all'indirizzo `http://opensource.samsung.com`.

3.1.2. AOSP: compilazione dei sorgenti e *flashing* del dispositivo. Dopo la procedura di configurazione e di ottenimento dei sorgenti, è possibile procedere con la loro compilazione. Per assicurarsi in ogni momento che la cartella di *output* sia effettivamente pulita, è possibile eseguire il comando:

`make clobber`

[1] v. `http://source.android.com/source/known-issues.html`

Devo però avvertire a questo punto che questa operazione cancella completamente le configurazioni precedenti. Pertanto è opportuno eseguire di nuovo la procedura di configurazione come illustrato di seguito. Per iniziare la creazione dell'ambiente di cross-compilazione, è opportuno eseguire il comando:

```
. build/envsetup.sh
```

Successivamente, fornendo il comando `lunch`, si può scegliere per quale dispositivo effettuare la compilazione. Nel caso specifico del dispositivo Galaxy Nexus oggetto di Tesi, ho preferito utilizzare la configurazione `full_maguro-userdebug`, in quanto `full` indica la compilazione completa del sorgente, `maguro` è il *codename* del dispositivo e `userdebug` consente la visualizzazione di maggiori stampe a video di debug tramite LogCat, assieme all'attivazione dei permessi di root di default. In seguito lanciando il comando

```
make -jn
```

si avvia la compilazione, dove in particolare `n` identifica il numero di processi desiderati per effettuare la compilazione, terminata la quale si otterrà l'immagine prodotta all'interno del percorso `$ANDROID/out/target/product/generic`. Se questa non fosse sufficiente per la generazione del file (es.) `system.img`, un secondo processo di compilazione dovrebbe portare alla loro corretta compilazione.

Per proseguire ora con la procedura di *flashing* della *custom image*, è opportuno dichiarare in prima istanza una variabile globale `ANDROID_PRODUCT_OUT` con la quale definire quale sia il percorso contentente le immagini di sistema, tramite le quali consentire l'operazione di *flashing*, ed opzionalmente indicare quale sia il percorso dove sono stati compilati i binari per effettuare l'interazione con i dispositivi[2].

```
export PATH=\$AOSP/out/host/linux-x86/bin:\$PATH
export ANDROID_PRODUCT_OUT=\$AOSP/out/target/product/maguro
cd \$ANDROID_PRODUCT_OUT
```

Prima di effettuare l'operazione di flashing, è opportuno effettuare un backup del dispositivo, come illustrato nella Sottosezione 5.2.2 a pagina 75, allo scopo di poter ripristinare in un secondo momento le configurazioni precedenti del dispositivo e le applicazioni Java installate. Per proseguire con il *flashing* è tra l'altro necessario effettuare l'*unlocking* del dispositivo ed in seguito accedere alla modalità *bootloader*, come tra l'altro già illustrato in quella Sottosezione per il Samsung Galaxy Nexus.

Per inviare le immagini prodotte all'interno del dispositivo, sarà sufficiente eseguire il seguente comando:

```
fastboot -w flashall
```

[2]Questi binari sono tuttavia gli stessi proposti all'interno dell'SDK di Google.

L'operazione di flashing verrà avviata solamente se tutti i file(s) necessari saranno stati prodotti dalla precedente compilazione: in questo modo non si rischia di effettuare un'operazione parzialmente corretta.

3.2. Java Native Interface (JNI)

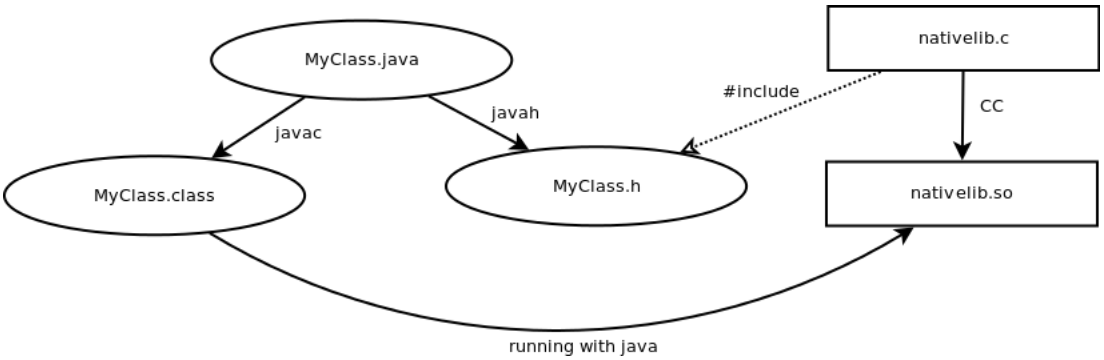


FIGURA 3.2.1. Passi di compilazione e di esecuzione del codice Java con l'utilizzo dell'interfaccia JNI. Tratto da [Lia99].

La JAVA NATIVE INTERFACE (JNI) è utilizzata principalmente per incorporare al codice Java il *native code* scritto con alcuni linguaggi quali il C ed il C++, e compilato in un codice binario per il *target*. Ciò implica indirettamente che il codice prodotto non sarà più eseguibile in modalità *multi-host*, rendendo pertanto necessaria in quel caso la ricompilazione della parte dipendente dall'architettura del sistema.

In questa trattazione non mi soffermerò sulle procedure di compilazione poiché queste sono già automatizzate dal *Makefile* dell'AOSP Source[3], ma farò comunque riferimento alla strutturazione del codice per permettere una comunicazione tramite l'interfaccia JNI: tale procedura è comunque riassunta in Figura 3.2.1.

All'interno di una classe Java possiamo specificare l'esistenza di un metodo nativo, e quindi non implementato all'interno di tale sorgente, utilizzando la seguente sintassi:

```
class MyClass {
        private native void method();
        public void othermethod() {
                /* no further ancillary data is provided */
        }
}
```

[3]Per ulteriore approfondimento, queste possono essere apprese in [Lia99, pp. 11-17], mentre per l'automatizzazione di questa procedura fornita dall'NDK si veda la Sottosezione 4.3.1.2 a pagina 72.

Possiamo invece definire tale metodo all'interno del codice nativo nel modo che segue:

```
#include <jni.h>
#include "MyClass.h"

JNIEXPORT void JNICALL Java_MyClass_method(JNIEnv *env, jobject this) {
        jclass class = (*env)->GetObjectClass(env,this);
        jmethodID callee = (*env)->GetMethodID(env,class,"othermethod"."()V");
        (*env)->CallVoidMethod(end,obj,callee);
}
```

In particolare, il primo parametro fa riferimento ad un puntatore ad un array ove sono contenuti tutti i metodi che è possibile chiamare da codice nativo, mentre il secondo è un'istanza dell'oggetto che ha invocato tale metodo.

All'interno di questo codice ho fornito un esempio di chiamata a metodo all'interno della classe `MyClass` Java chiamato `othermethod`.

Tuttavia questo non è l'unico modo di effettuare transizioni all'interno della JNI: si è per tanto in grado di creare un'istanza della JAVA VIRTUAL MACHINE all'interno del codice nativo, allo scopo di eseguire codice Java. Un esempio di come ciò possa essere utilizzato per eseguire metodi dichiarati nel codice Java, è fornito dalla funzione `main` di inizializzazione della DVM nel Listato 3.1 nella pagina successiva.

3.2.1. Esempi di interazione tra AOSP Source e Kernel Android.

3.2.1.1. *UIEventObserver.* Fornisco ora un esempio piuttosto immediato per mostrare come, tramite l'utilizzo della JNI, sia possibile intercettare direttamente all'interno di codice Java, del tutto indipendente dall'architettura della macchina, eventi sollevati dallo stesso Kernel. La classe `UEventObserver` di fatti costituisce un ponte tra i servizi offerti dall'AOSP ed eventi sollevati dal Kernel: utilizzando infatti la libreria *libhardware_legacy*, è in grado di raccogliere gli eventi generati dall'interazione dell'utente con il dispositivo, tramite l'accesso alla *systemcall* `socket` come mostrato dal file:

`$AOSP/hardware/libhardware_legacy/uevent/uevent.c`

In particolare si predisporrebbe l'ascolto degli eventi Netlink creando un socket apposito nel seguente modo:

```
socket(PF_NETLINK,SOCK_DGRAM,NETLINK_KOBJECT_UEVENT)
```

Questo mostra in parte come alcune interazioni non vengono gestite unicamente dal Kernel, ma di come spesso il comportamento di base sia modificato dalle funzioni definite in Java, come nell'esempio.

Listato 3.1dalvik/dalvikvm/Main.cpp

```
int main(int argc, char* const argv[])
{
    JavaVM* vm = NULL;
    JNIEnv* env = NULL;
    JavaVMInitArgs initArgs;
    JavaVMOption* options = NULL;
    char* slashClass = NULL;
    int optionCount, curOpt, i, argIdx;
    int needExtra = JNI_FALSE;
    int result = 1;

    //Omissis
    /* Start VM.  The current thread becomes the main thread of the VM. */
    if (JNI_CreateJavaVM(&vm, &env, &initArgs) < 0) goto bail;

    //Omissis
    /* We want to call main() with a String array with our arguments in it.
     * Create an array and populate it.  Note argv[0] is not included. */
    jobjectArray strArray;
    strArray = createStringArray(env, &argv[argIdx+1], argc-argIdx-1);
    if (strArray == NULL) goto bail;

    /*Find [class].main(String[]).*/
    jclass startClass;
    jmethodID startMeth;

    /* Omissis: convert "com.android.Blah" to "com/android/Blah" */
    startClass = env->FindClass(slashClass);
    //Omissis
    startMeth = env->GetStaticMethodID(startClass,"main", "([Ljava/lang/String
        ;)V");
    if (startMeth == NULL) {
        fprintf(stderr, "Dalvik VM unable to find static main(String[]) in '%s
            '\n",slashClass);
        goto bail;
    }

    /*Make sure the method is public.  JNI doesn't prevent us from calling
     * a private method, so we have to check it explicitly.*/
    if (!methodIsPublic(env, startClass, startMeth)) goto bail;

    /*Invoke main().*/
    env->CallStaticVoidMethod(startClass, startMeth, strArray);
    if (!env->ExceptionCheck()) result = 0;

bail:
    if (vm != NULL) {
        /* This allows join() and isAlive() on the main thread to work
         * correctly, and also provides uncaught exception handling. */
        if (vm->DetachCurrentThread() != JNI_OK) {
         fprintf(stderr, "Warning: unable to detach main thread\n"); result =
             1;
        }
        if (vm->DestroyJavaVM() != 0)
            fprintf(stderr, "Warning: Dalvik VM did not shut down cleanly\n");
    }
    //Omissis
}
```

3.2.1.2. *Binder.* Un ulteriore esempio di come si utilizzi JNI all'interno di Android è fornito dal Binder (v. Sezione 3.5 a pagina 35): come si può evincere dal sorgente, anche questo meccanismo sia implementato su più livelli:

Definizione di interfaccia API: Si provvedono tra le altre le interfacce `android.os.Parcelable` e `android.os.IBinder`, e le classi `android.os.Parcel`, `android.os.Bundle`, `android.content.Intent` e `android.os.Binder`

JNI: Anche in questo caso si utilizza del codice C++ per effettuare il collegamento tra l'implementazione in C++ e le API Java: tale *bridge* è fornito dal file:

`$AOSP/frameworks/base/core/jni/android_util_Binder.cpp`

C++ "middleware": Questo ulteriore strato di codice si frappone tra il driver di sistema immediatamente sottostante ed il livello delle JNI, creando l'implementazione delle primitive di comunicazione. I sorgenti di questo livello architetturale sono forniti all'interno del percorso:

`$AOSP/frameworks/native/libs/binder/`[4]

Kernel Driver: Il file:

`./frameworks/base/cmds/servicemanager/binder.c`

fornisce un'implementazione sulle *systemcall* sulle operazioni di `open`, `mmap`, `release`, `poll` e `ioctl`.

Questa trattazione preliminare mostra come la descrizione fornita dalla Figura 3.1.1 a pagina 22 possa descrivere solamente come, in prima battuta, si presenti il Binder a livello di applicazioni Java: in realtà anche il Binder si dispone sia a livello di *Android Framework Libraries*, sia a livello di *Native Libs*, sia a livello *Android Kernel* predisponendo un driver.

Un'ulteriore esempio di implementazione di servizio lato Java dal quale poi ricevere le richieste lato Java è fornito da [Gar07]. Tuttavia nel sorgente proposto non sono ancora previste *upsyscall* da programmi C.

3.3. Startup del sistema operativo

Un'*overview* generale dei processi che vengono attivati durante la procedura di startup è fornita nella Figura 3.3.1 a fronte: mi concentrerò sulla fase di attivazione del processo Zygote e del System Server, mentre tratterò del Service Manager nella Sezione 3.5 a pagina 35.

[4] Il posizionamento di tali sorgenti è stata cambiata dalla redazione di [Sch11], dove questi erano localizzati in `$AOSP/frameworks/base/libs/utils/`.

Analogamente vale per lo stato di Kernel Driver, dove in quei tempi tale implementazione era localizzata in `$AOSP/drivers/staging/android/`.

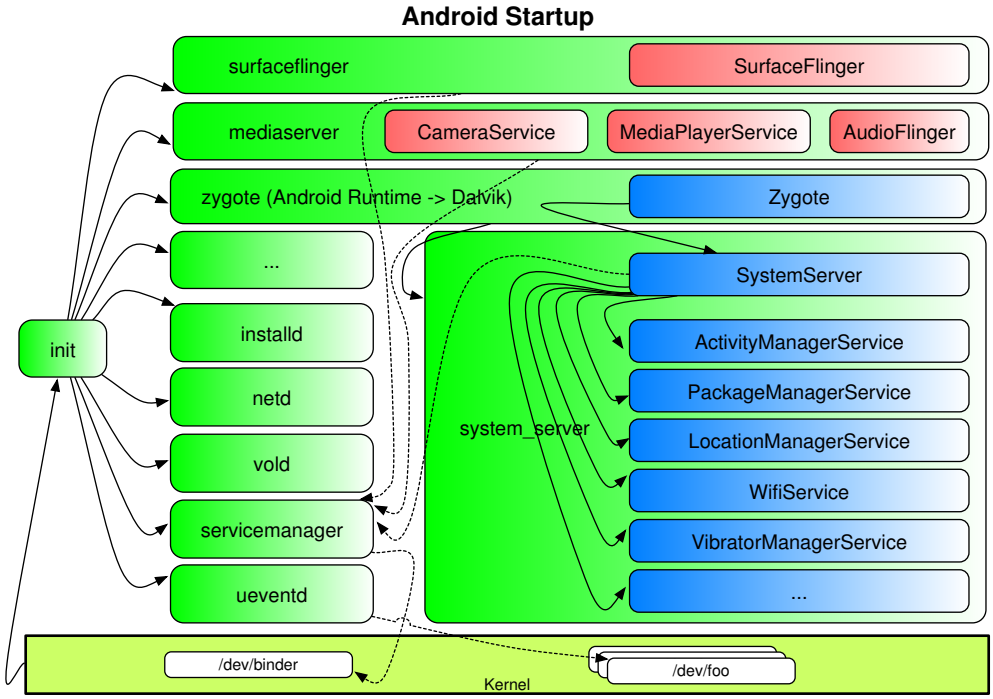


FIGURA 3.3.1. Schematizzazione della procedura di startup di Android. [GJG12]

Un'importante differenza con i sistemi GNU/Linux è la presenza di un file `init.rc` definito secondo un specifico linguaggio, detto ANDROID INIT LANGUAGE (descritto dal file riportato nella Sezione A.1 a pagina 109); possiamo trovare il sorgente del file di inizializzazione al percorso:

`$AOSP/system/core/rootdir/init.rc`

All'interno di questo file vengono effettuate, tra le altre, le seguenti operazioni:

- Vengono inizializzate le variabili d'ambiente
- Viene effettuato il mounting delle varie componenti del *filesystem*: è in questo punto dove, in particolare, si monta il *root filesystem* e la cartella `/system` come di sola lettura.
- Vengono definite le proprietà di sistema tramite il comando `setprop`: tra gli altri parametri, figurano le configurazioni sui permessi utente della Shell, e la quantità di dati utilizzata all'interno dei buffer di sistema per la comunicazione TCP:

```
# Define TCP buffer sizes for various networks
#   ReadMin, ReadInitial, ReadMax, WriteMin, WriteInitial, WriteMax,
```

```
    setprop net.tcp.buffersize.default
        4096,87380,110208,4096,16384,110208
    setprop net.tcp.buffersize.wifi
        524288,1048576,2097152,262144,524288,1048576
    setprop net.tcp.buffersize.lte
        524288,1048576,2097152,262144,524288,1048576
    setprop net.tcp.buffersize.umts
        4094,87380,110208,4096,16384,110208
    setprop net.tcp.buffersize.hspa
        4094,87380,262144,4096,16384,262144
    setprop net.tcp.buffersize.hsupa
        4094,87380,262144,4096,16384,262144
    setprop net.tcp.buffersize.hsdpa
        4094,87380,262144,4096,16384,262144
    setprop net.tcp.buffersize.hspap
        4094,87380,1220608,4096,16384,1220608
    setprop net.tcp.buffersize.edge
        4093,26280,35040,4096,16384,35040
    setprop net.tcp.buffersize.gprs    4092,8760,11680,4096,8760,11680
    setprop net.tcp.buffersize.evdo
        4094,87380,262144,4096,16384,262144
```

- ◇ Vengono avviati i *service* di sistema con gli adeguati permessi e socket di comunicazione.

Tra gli altri *service* viene iniziato Zygote tramite le seguenti istruzioni:

```
service zygote /system/bin/app_process -Xzygote /system/bin --zygote --start-
    system-server
    class main
    socket zygote stream 660 root system
    onrestart write /sys/android_power/request_state wake
    onrestart write /sys/power/state on
    onrestart restart media
    onrestart restart netd
```

Possiamo quindi notare che `app_process` all'interno del sorgente è situato all'interno del file:

`$AOSP/frameworks/base/cmds/app_process/app_main.cpp`

che, come vediamo, ha il compito sia di eseguire Zygote, sia di eseguire una qualsiasi classe Java che contenga al suo interno un metodo `main`. In particolare tale esecuzione è garantita dalle funzioni di inizializzazione della DVM, che sono descritte all'interno del file:

`$AOSP/frameworks/base/core/jni/AndroidRuntime.cpp`

Come si può notare dal suo sorgente, è in questo luogo che avvengono le inizializzazioni dello strato di interazione JNI, effettuando l'invocazione delle funzioni descritte all'interno della variabile:

```
static const RegJNIRec gRegJNI = {
   ...  REG_JNI(funzione), ...
}
```

Queste funzioni verranno invocate tramite l'esecuzione del metodo `start`, che a sua volta invoca il metodo `startReg`, accettando come argomento l'array mostrato sopra. All'interno di quest'ultimo compare anche il riferimento alla funzione `register_android_os_Binder` che, definita tra le componenti del Binder lato JNI, richiama a sua volta le funzioni:

- `int_register_android_os_Binder`
- `int_register_android_os_BinderInternal`
- `int_register_android_os_BinderProxy`

A questo punto posso notare come la necessità della definizione di una nuova macchina virtuale sia dettata, più che per motivi di efficienza e di implementazione come viene mostrato in [Car11, pp. 9-11], dalla necessità di consentire le inizializzazioni dello strato JNI utilizzate dai service Java del sistema operativo.

Passando quindi alla funzione di inizializzazione `int_register_android_os_Binder`, posso notare come venga inizializzata la variabile `gBinderOffsets` definita all'interno del file per la configurazione del Binder lato JNI, effettuando all'interno dei campi della struttura che essa rappresenta le seguenti associazioni:

`mClass`: ottiene un riferimento alla classe Java `Binder`

`mExecTransact`: ottiene il riferimento al metodo `execTransact` della classe di cui sopra.

`mObject`: mentre nella classe Java corrispondente questo corrisponderà ad un intero, lato JNI rappresenterà il puntatore ad un oggetto nativo di tipo `JavaBBinderHolder`.

All'interno della stessa funzione JNI, si procede alla registrazione dei metodi nativi definiti in `gBinderMethods`.

Viene quindi eseguita la classe descritta dal file:
`$AOSP/frameworks/base/core/java/com/android/internal/os/ZygoteInit.java`

Dalla funzione `main` ivi definita viene quindi avviato il System Server, descritto dal file:
`$AOSP/frameworks/base/services/java/com/android/server/SystemServer.java`

Possiamo vedere come da lato Java si esegua il metodo nativo `init1()` definito all'interno del file:
`$AOSP/frameworks/base/services/jni/com_android_server_SystemServer.cpp`

E, dalla cui definizione, si effettua l'esecuzione del metodo `system_init()`, definito all'interno della libreria `system_service` ed in particolare per il file:
`$AOSP/frameworks/base/cmds/system_server/library/system_init.cpp`

in questo luogo, subito dopo aver lanciato il metodo nativo all'interno della macchina virtuale, si entra all'interno del *main loop* per la raccolta di richieste da parte delle applicazioni in esecuzione nel sistema.

Questo poi si preoccupa di inizializzare, tra gli altri, i seguenti gestori lato Java all'interno del metodo `init2()`:

- `DevicePolicyManagerService`
- `NetworkManagementService`
- `ConnectivityService`
- `ThrottleService`
- `AudioService`

Una lista completa dei servizi in esecuzione all'interno del dispositivo è fornita eseguendo l'istruzione `service list` via terminale del dispositivo Android. Tuttavia non è presente una sufficiente documentazione di come questi servizi interagiscano [Yag11]: lo studio dei vari errori durante i tentativi di porting di `pjsua` possono costituire l'occasione per effettuare una prima indagine.

3.4. Differenze tra Bionic e Libc

Analizzando la computazione di `strace` dall'emulatore, possiamo osservare come, all'interno di un sistema operativo Android, avvengano molte più chiamate a systemcall rispetto all'esecuzione dello stesso binario all'interno di un computer con sistema operativo GNU/Linux. Tra queste possiamo notare la seguente:

```
open(/dev/urandom, O_RDONLY|O_LARGEFILE)
```

L'apertura del *device* in questione fa riferimento alla generazione di valori Random: tramite questo *device*, il sistema operativo si assicura che, i numeri casuali generati, siano sempre crittograficamente sicuri. Possiamo notare quindi che la libreria *Bionic* di Google è stata scritta appositamente per utilizzare le peculiarità del sistema operativo Android, che lo differenziano da GNU/Linux.

Un'ulteriore differenza tra queste due librerie è provato dall'utilizzo del driver `ashmem` per effettuare la creazione di aree di memoria condivise tra processi: questa implementazione presenta delle API simili a quelle POSIX, anche se dopo aver ottenuto un *file descriptor* dal driver è necessario effettuarne il `mmap`, come illustrato dall'esempio proposto di seguito:

```
#define MAX 4096
#define NAME "regione"
void* data;
int fd = ashmem_create_region(NAME,MAX);
if (fd<=0) return;
if (data = mmap(NULL,MAX,PROT_READ|PROT_WRITE,MAP_SHARED,fd,0)) {
        /* no further ancillary data is provided */
}
```

Questo device è utilizzato allo scopo di garantire la liberazione della memoria condivisa da parte del Kernel [GJG12]. È anche utilizzata, come mostrerò in seguito, per permettere la comunicazione diretta tra due thread.

Un ulteriore differenza di implementazione è costituita dall'implementazione dei thread, che è inclusa direttamente all'interno della libreria *Bionic*: per la compilazione di sorgente che utilizza thread non è quindi necessario effettuare il linking con una libreria `pthread`.

3.5. IPC tramite Binder

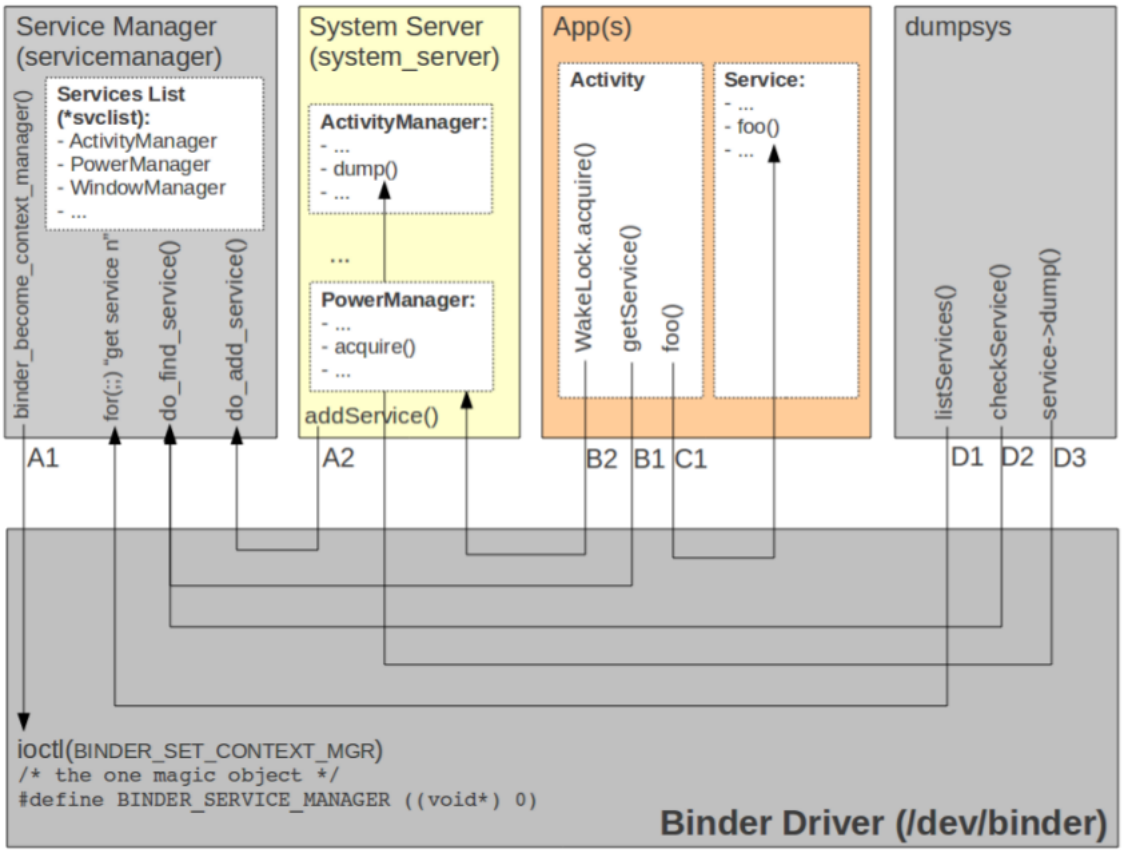


FIGURA 3.5.1. Alcuni meccanismi di IPC tramite `/dev/binder`. [Yag11]

La maggior parte delle interazioni interessanti che avvengono all'interno di Android sono possibili grazie al device Binder, implementazione Google di *OpenBinder* della *Be Inc*, dove quest'ultima non è più supportata. Come riferiscono i primi ideatori, questo è un «componente localizzato a livello di sistema, per fornire un'astrazione di alto livello al di sopra dei tradizionali sistemi operativi». Andando ora nel dettaglio della corrente implementazione, il Binder permette di effettuare IPC per chiamare metodi remoti, in questo caso quello

dei *service*, come se fossero locali, utilizzando un meccanismo di chiamate sincrone tramite meccanismi di richiesta e risposta: ciò implica che, se il chiamato (es.) esegue un ciclo infinito senza fornire una risposta, il chiamante rimarrebbe bloccato; è stato tuttavia previsto un meccanismo di *Death Notification* grazie al quale il chiamante in attesa viene notificato della cessazione del servizio remoto.

In particolare abbiamo che il *Service Manager*, dopo essere stato avviato nello `init.rc` e dopo essersi registrato presso il Binder tramite l'invocazione di `ioctl` con il comando `BINDER_SET_CONTEXT_MGR`, acquisisce la possibilità di diventare il *Context Manager* di sistema, avendo la possibilità di fornire l'indicizzazione di tutti i *service* di sistema. In questo modo è in grado di fornire al richiedente l'istanza del Binder di comunicazione, in modo da interagire con il servizio richiesto. Un esempio pratico dell'utilizzo del *Service Manager* è facilmente riscontrabile all'interno delle Java API, come nel caso dell'ottenimento del `NotificationManager`:

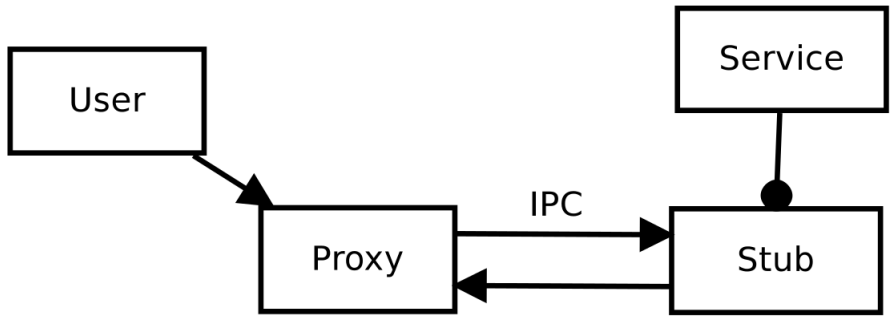
```
NotificationManager mNotificationManager = (NotificationManager)
    getSystemService(Context.NOTIFICATION_SERVICE);
```

Come possiamo vedere dalla Figura (a) 3.5.2 nella pagina successiva, l'applicazione client (nell'immagine *User*) dispone di un *proxy* di comunicazione che funge da tramite per la comunicazione con lo *stub* interagente con il *service*: in genere nelle applicazioni del mondo Java queste implementazioni sono fornite automaticamente dalla compilazione di interfacce dette AIDL (ANDROID INTERFACE DEFINITION LANGUAGE) [Car11], anche se possono essere esse stesse prodotte dallo stesso sviluppatore [Sch11].

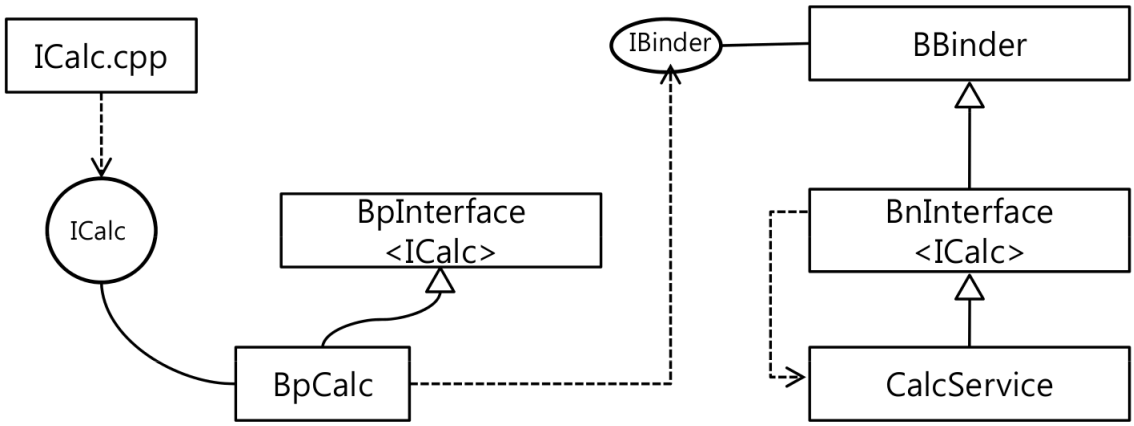
Questa forma di comunicazione è resa possibile sia dal fatto che *proxy* e *stub* hanno un'interfaccia comune, sia perché allo stesso *proxy* viene fornita un'istanza del Binder tramite la quale viene resa possibile la comunicazione IPC. Mostro inoltre di seguito un esempio di codice generato automaticamente dalla definizione della AIDL, del quale mi interesserò unicamente della trattazione dello *stub* lato *service*.

Listato 3.2. IPermissionController

```
/*
 * This file is auto-generated.  DO NOT MODIFY.
 * Original file: frameworks/base/core/java/android/os/IPermissionController.
    aidl
 */
package android.os;
/** @hide */
public interface IPermissionController extends android.os.IInterface
{
        /** Local-side IPC implementation stub class. */
        public static abstract class Stub extends android.os.Binder implements
            android.os.IPermissionController
```



(A) *Architettura High-level della comunicazione tramite Binder lato Java. [Sch11]*



(B) *Generalizzazione della gerarchia tra classi lato client e service.* `http://www.aesop.or.kr/Board_Documents_Android_Frameworks/34984`

FIGURA 3.5.2. *Overview* del protocollo di comunicazione predisposto dal Binder.

```
        {
                private static final java.lang.String DESCRIPTOR = "android.os.
                    IPermissionController";
                /** Construct the stub at attach it to the interface. */
                public Stub()
                {
                        this.attachInterface(this, DESCRIPTOR);
                }
                /**
                 * Cast an IBinder object into an android.os.
                     IPermissionController interface,
                 * generating a proxy if needed.
                 */
```

```
public static android.os.IPermissionController asInterface(
    android.os.IBinder obj)
{
        if ((obj==null)) {
                return null;
        }
        android.os.IInterface iin = (android.os.IInterface)obj.
            queryLocalInterface(DESCRIPTOR);
        if (((iin!=null)&&(iin instanceof android.os.
            IPermissionController))) {
                return ((android.os.IPermissionController)iin);
        }
        return new android.os.IPermissionController.Stub.Proxy(
            obj);
}
public android.os.IBinder asBinder()
{
        return this;
}
@Override public boolean onTransact(int code, android.os.Parcel
     data, android.os.Parcel reply, int flags) throws android.os
    .RemoteException
{
        switch (code)
        {
        case INTERFACE_TRANSACTION:
        {
                reply.writeString(DESCRIPTOR);
                return true;
        }
        case TRANSACTION_checkPermission:
        {
                data.enforceInterface(DESCRIPTOR);
                java.lang.String _arg0;
                _arg0 = data.readString();
                int _arg1;
                _arg1 = data.readInt();
                int _arg2;
                _arg2 = data.readInt();
                boolean _result = this.checkPermission(_arg0,
                    _arg1, _arg2);
                reply.writeNoException();
                reply.writeInt(((_result)?(1):(0)));
                return true;
        }
        }
        return super.onTransact(code, data, reply, flags);
}
```

```
                private static class Proxy implements android.os.
                    IPermissionController
                {
                        private android.os.IBinder mRemote;
                        Proxy(android.os.IBinder remote)
                        {
                                mRemote = remote;
                        }
                        public android.os.IBinder asBinder()
                        {
                                return mRemote;
                        }
                        public java.lang.String getInterfaceDescriptor()
                        {
                                return DESCRIPTOR;
                        }
                        public boolean checkPermission(java.lang.String
                            permission, int pid, int uid) throws android.os.
                            RemoteException
                        {
                                android.os.Parcel _data = android.os.Parcel.
                                    obtain();
                                android.os.Parcel _reply = android.os.Parcel.
                                    obtain();
                                boolean _result;
                                try {
                                        _data.writeInterfaceToken(DESCRIPTOR);
                                        _data.writeString(permission);
                                        _data.writeInt(pid);
                                        _data.writeInt(uid);
                                        mRemote.transact(Stub.
                                            TRANSACTION_checkPermission, _data,
                                            _reply, 0);
                                        _reply.readException();
                                        _result = (0!=_reply.readInt());
                                }
                                finally {
                                        _reply.recycle();
                                        _data.recycle();
                                }
                                return _result;
                        }
                }
                static final int TRANSACTION_checkPermission = (android.os.
                    IBinder.FIRST_CALL_TRANSACTION + 0);
        }
        public boolean checkPermission(java.lang.String permission, int pid,
            int uid) throws android.os.RemoteException;
}
```

```
// da ActivityManagerService (Utilizza lo stub di cui sopra)

    // ======================================================
    // PERMISSIONS
    // ======================================================

    static class PermissionController extends IPermissionController.Stub {
        ActivityManagerService mActivityManagerService;
        PermissionController(ActivityManagerService activityManagerService) {
            mActivityManagerService = activityManagerService;
        }

        public boolean checkPermission(String permission, int pid, int uid) {
            return mActivityManagerService.checkPermission(permission, pid,
                    uid) == PackageManager.PERMISSION_GRANTED;
        }
    }
```

Posso inoltre sottolineare come ad ogni *service* sia associato un descriptor, ovvero una stringa identificativa, che potrà essere utilizzata da un client per richiedere un'istanza del *proxy* tramite il quale far avvenire la comunicazione.

All'interno della Sottosezione 3.6.2 a pagina 59 mostrerò il meccanismo di IPC che avviene a livello di librerie del sistema operativo, mentre mostrerò molto brevemente nella Sottosezione 4.3.1.2 a pagina 72 come sia possibile compilare i files AIDL in codice Java.

3.5.1. Casi di studio. *All'interno di questa sottosezione, mostrerò i principali meccanismi di registrazione ed ottenimento di servizi, sia lato nativo, sia lato Java, allo scopo di tralasciare questi dettagli nella descrizione delle interazioni osservate in Sezione 3.6 a pagina 54. Discorrerò quindi di:*

- ◇ Registrazione di *service* nativi: v. 3.5.1.1 nella pagina successiva.
- ◇ Invocazione di RPC da codice nativo: v. 3.5.1.2 a pagina 45.
- ◇ Registrazione dei servizi lato Java: v. 3.5.1.3 a pagina 48.
- ◇ Invocazione di metodi Java da Native Code: v. 3.5.1.4 a pagina 50.

Per una prima visione d'insieme, mostro la Figura 3.5.3 nella pagina successiva: come si può notare, il codice C++ fornisce l'implementazione per interagire con i metodi definiti da un'interfaccia `IXXX`: vengono allo scopo definiti le classi `BpXXX` e `BnXXX`, del quale il Listato 3.3 a pagina 42 fornisce un esempio. Come possiamo notare, il programma cliente interagirà con l'oggetto del tipo `BpXXX` il quale, grazie al Binder, riuscirà ad interagire con un altro oggetto `BnXXX`, esteso da un servizio che lo implementa: questo è necessario in quanto questa classe non implementa tutti i metodi necessari all'effettiva esecuzione del codice che sono lasciati `virtual`, come il metodo `checkPermission` in esso definito.

Listato 3.3IPermissionController.cpp

```
namespace android {

// ---------------------------------------------------------------------

class BpPermissionController : public BpInterface<IPermissionController>
{
public:
    BpPermissionController(const sp<IBinder>& impl)
        : BpInterface<IPermissionController>(impl)
    {
    }

    virtual bool checkPermission(const String16& permission, int32_t pid,
        int32_t uid)
    {
        Parcel data, reply;
        data.writeInterfaceToken(IPermissionController::getInterfaceDescriptor
            ());
        data.writeString16(permission);
        data.writeInt32(pid);
        data.writeInt32(uid);
        remote()->transact(CHECK_PERMISSION_TRANSACTION, data, &reply);
        // fail on exception
        if (reply.readExceptionCode() != 0) return 0;
        return reply.readInt32() != 0;
    }
};

IMPLEMENT_META_INTERFACE(PermissionController, "android.os.
    IPermissionController");

// ---------------------------------------------------------------------

status_t BnPermissionController::onTransact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    //printf("PermissionController received: "); data.print();
    switch(code) {
        case CHECK_PERMISSION_TRANSACTION: {
            CHECK_INTERFACE(IPermissionController, data, reply);
            String16 permission = data.readString16();
            int32_t pid = data.readInt32();
            int32_t uid = data.readInt32();
            bool res = checkPermission(permission, pid, uid);
            reply->writeNoException();
            reply->writeInt32(res ? 1 : 0);
            return NO_ERROR;
        } break;
        default:
            return BBinder::onTransact(code, data, reply, flags);
    }
}

}
```

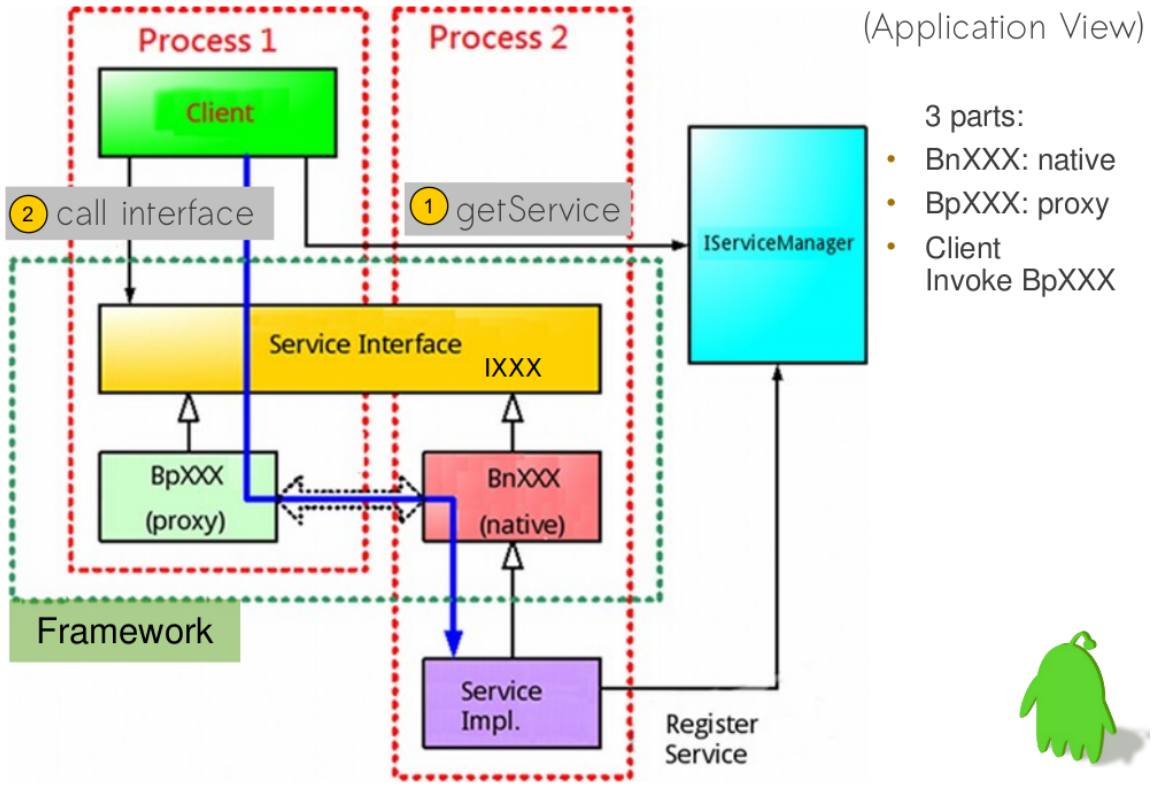


FIGURA 3.5.3. *Gerarchia delle classi lato applicazioni in codice nativo C++. [Hua12].*

3.5.1.1. *Registrazione di service nativi.* Prima di poter effettuare la registrazione di un *service* nativo, è opportuno ottenere un *proxy* per la comunicazione con il *Service Manager*: questo è ottenibile tramite il metodo `defaultServiceManager` il quale, o richiama un'istanza *singleton* precedentemente creata per tutto il sistema, oppure ricorre a crearne una nuova tramite il metodo `getContextObject(caller)` tramite `ProcessState`.

Come osservabile dal Listato che segue, si utilizza un particolare oggetto, detto `ProcessState`, per mettersi in comunicazione con il Binder, ed in particolare per aprire il suddetto device [5]

Listato 3.4. IServiceManager.cpp

```
sp<IServiceManager> defaultServiceManager()
{
    if (gDefaultServiceManager != NULL) return gDefaultServiceManager;
```

[5]In particolare all'interno dell'ultimo sorgente AOSP si riscontrano delle differenze di implementazione da quelle evidenziate in [saf10].

```
    {
        AutoMutex _l(gDefaultServiceManagerLock);
        if (gDefaultServiceManager == NULL) {
            gDefaultServiceManager = interface_cast<IServiceManager>(
                ProcessState::self()->getContextObject(NULL));
        }
    }

    return gDefaultServiceManager;
}
```

In particolare il costruttore di tale oggetto viene definito nel seguente modo:

Listato 3.5. (1) ProcessState.cpp

```
ProcessState::ProcessState()
    : mDriverFD(open_driver())
    , mVMStart(MAP_FAILED)
    , mManagesContexts(false)
    , mBinderContextCheckFunc(NULL)
    , mBinderContextUserData(NULL)
    , mThreadPoolStarted(false)
    , mThreadPoolSeq(1)
{
    if (mDriverFD >= 0) {
        // XXX Ideally, there should be a specific define for whether we
        // have mmap (or whether we could possibly have the kernel module
        // availabla).
#if !defined(HAVE_WIN32_IPC)
        // mmap the binder, providing a chunk of virtual address space to
            receive transactions.
        mVMStart = mmap(0, BINDER_VM_SIZE, PROT_READ, MAP_PRIVATE |
            MAP_NORESERVE, mDriverFD, 0);
        if (mVMStart == MAP_FAILED) {
            // *sigh*
            ALOGE("Using /dev/binder failed: unable to mmap transaction memory
                .\n");
            close(mDriverFD);
            mDriverFD = -1;
        }
#else
        mDriverFD = -1;
#endif
    }

    LOG_ALWAYS_FATAL_IF(mDriverFD < 0, "Binder driver could not be opened.
        Terminating.");
}
```

Si riscontra come si apra il device in questione e si effettui il *mapping* di una regione di memoria, allo scopo di ricevere le transazioni: quest'ultima utility è in particolare utilizzata dai *service* per ricevere le richieste dai client. In particolare dal metodo `getContextObject` viene sempre restituito un oggetto di tipo `BpBinder` il quale, conoscendo lo *hanlde* associato, ovvero il numero assegnato dal Binder al servizio richiesto, che nel caso del *Service Manager* corrisponde sempre a zero, richiede di ottenere un *proxy* tramite il sorgente qui proposto. Come risulta dal Listato fornito, si ottiene comunque un oggetto di tipo `BpBinder`:

Listato 3.6. (2) ProcessState.cpp

```
sp<IBinder> ProcessState::getStrongProxyForHandle(int32_t handle)
{
    sp<IBinder> result;

    AutoMutex _l(mLock);

    handle_entry* e = lookupHandleLocked(handle);

    if (e != NULL) {
        // We need to create a new BpBinder if there isn't currently one, OR we
        // are unable to acquire a weak reference on this current one. [The
        // attemptIncWeak() is safe because we know the BpBinder destructor
            will
        // always call expungeHandle(), which acquires the same lock we are
            holding
        // now. We need to do this because there is a race condition between
        // someone releasing a reference on this BpBinder, and a new reference
        // on its handle arriving from the driver].

        if (b == NULL || !e->refs->attemptIncWeak(this)) {
            b = new BpBinder(handle);
            e->binder = b;
            if (b) e->refs = b->getWeakRefs();
            result = b;
        } else {
            // This little bit of nastyness is to allow us to add a primary
            // reference to the remote proxy when this team doesn't have one
            // but another team is sending the handle to us.
            result.force_set(b);
            e->refs->decWeak(this);
        }
    }

    return result;
}
```

L'oggetto così ottenuto può essere quindi utilizzato dal metodo `interface_cast` definito in `IInterface.h`, e definito come segue:

```
template<typename INTERFACE>
inline sp<INTERFACE> interface_cast(const sp<IBinder>& obj)
{
    return INTERFACE::asInterface(obj);
}
```

In particolare, senza entrare nel dettaglio delle macro, tale funzione è dichiarata con la macro `DECLARE_META_INTERFACE` ed implementata tramite `IMPLEMENT_META_INTERFACE`, passando `obj` come argomento al costruttore di `BpServiceInterface`, e che quindi tramite supercostruttore lo associa all'attributo `mRemote`, e che verrà poi restituito dal metodo `remote()`.

A questo punto mostro come avvenga la registrazione di service nativi quali lo AudioFlinger, che estende BnAudioFlinger. Effettuando quindi l'invocazione del metodo `addService` sul Proxy così ottenuto, provoco l'invocazione del metodo `transact` dell'oggetto `BpBinder`. Come si può vedere dal codice di BpBinder, e che in questa sede non riporto[6], si mostra come sia possibile instaurare la comunicazione con il Binder tramite l'utilizzo della classe `IPCThreadState`. Brevemente seguono le seguenti operazioni:

- ◇ Il Parcel[7] di comunicazione viene passato al driver, il quale lo fornisce al *Service Manager* in attesa.
- ◇ Quest'ultimo ottiene dal Parcel il puntatore al `BnXXX`, ad esempio `BnAudioFlinger`, e trattiene l'oggetto: ciò è possibile poiché viene passato come indirizzo, come mostrato dall'implementazione della funzione `flatten_binder`.

Quest'operazione è inoltre possibile in quanto il *Service Manager*, dopo aver effettuato con successo la richiesta di diventare Context Manager, si mette in ascolto lato driver delle eventuali richieste di ottenimento o di registrazione dei servizi, come mostrato in Figura (a) 3.5.4 a pagina 49.

3.5.1.2. *Invocazione di RPC da codice nativo. Analizzerò brevemente come avvenga l'IPC tra clienti e service all'interno del codice nativo*. Facendo riferimento ancora alla Figura 3.5.3 a pagina 41 e alla Figura (b) 3.5.5 a pagina 51 facente riferimento alla gerarchia delle classi native lato chiamante, posso osservare come l'invocazione del metodo del servizio avvenga nel seguente modo:

`BBinder::transact()` → `BnXXX::onTransact()` → `XXX::method()`

In particolare voglio mostrare come sia possibile gestire i messaggi lato richiedente utilizzando un server che, grazie all'utilizzo di `ProcessState` e di `IPCThreadState`, rimane in attesa dei messaggi provenienti dai chiamanti tramite driver Binder.

[6] Per i dettagli implementativi dell'interazione con il Binder si veda [saf10] ed il sito Cinese `http://www.linuxidc.com/Linux/2011-07/39274.htm`.

[7] Un `Parcel` rappresenta i contenuti del messaggio che devono essere spediti al destinatario tramite il Binder.

Passando ora all'atto della creazione di un *service* lato codice nativo, posso mostrare come, per il *media_server*, questo sia implementato all'interno del file:

`$AOSP/frameworks/av/media/media_server/main_media_server.cpp`

Riporto il codice qui sotto:

```
using namespace android;

int main(int argc, char** argv)
{
    sp<ProcessState> proc(ProcessState::self());
    sp<IServiceManager> sm = defaultServiceManager();
    ALOGI("ServiceManager: %p", sm.get());
    AudioFlinger::instantiate();
    MediaPlayerService::instantiate();
    CameraService::instantiate();
    AudioPolicyService::instantiate();
    ProcessState::self()->startThreadPool();
    IPCThreadState::self()->joinThreadPool();
}
```

In particolare i metodi `instantiate()` delle classi sono definiti all'interno della classe dichiarata all'interno del file che segue, in quanto estendono la classe `BinderService`:

`$AOSP/frameworks/native/include/binder/BinderService.h`

Si può quindi notare come il metodo in questione, che richiama a sua volta quello definito `publish()`, effettui sostanzialmente la sottoscrizione del servizio, come indicato dal Listato:

```
static status_t publish(bool allowIsolated = false) {
        sp<IServiceManager> sm(defaultServiceManager());
        return sm->addService(String16(SERVICE::getServiceName()), new SERVICE
            (), allowIsolated);
}
```

Guardando ora all'implementazione di `joinThreadPool` che fornisce il *main loop* ai processi nativi (come tra l'altro già accennato per il *System Server*), possiamo notare come:

- Tramite il metodo `talkWithDriver()` si ottenga il pacchetto che il richiedente ha inviato tramite il driver Binder.
- Tramite `executeCommand` si ottiene, entrando nel *case* `BR_TRANSACTION`, il riferimento al `BnXXX` che è stato registrato in precedenza, eseguendo quindi su di esso il metodo `transact`.

```
void IPCThreadState::joinThreadPool(bool isMain)
{
    LOG_THREADPOOL("**** THREAD %p (PID %d) IS JOINING THE THREAD POOL\n", (
        void*)pthread_self(), getpid());
```

```
mOut.writeInt32(isMain ? BC_ENTER_LOOPER : BC_REGISTER_LOOPER);

// This thread may have been spawned by a thread that was in the background
// scheduling group, so first we will make sure it is in the foreground
// one to avoid performing an initial transaction in the background.
set_sched_policy(mMyThreadId, SP_FOREGROUND);

status_t result;
do {
    int32_t cmd;

    // When we've cleared the incoming command queue, process any pending
        derefs
    if (mIn.dataPosition() >= mIn.dataSize()) {
        size_t numPending = mPendingWeakDerefs.size();
        if (numPending > 0) {
            for (size_t i = 0; i < numPending; i++) {
                RefBase::weakref_type* refs = mPendingWeakDerefs[i];
                refs->decWeak(mProcess.get());
            }
            mPendingWeakDerefs.clear();
        }

        numPending = mPendingStrongDerefs.size();
        if (numPending > 0) {
            for (size_t i = 0; i < numPending; i++) {
                BBinder* obj = mPendingStrongDerefs[i];
                obj->decStrong(mProcess.get());
            }
            mPendingStrongDerefs.clear();
        }
    }

    // now get the next command to be processed, waiting if necessary
    result = talkWithDriver();
    if (result >= NO_ERROR) {
        size_t IN = mIn.dataAvail();
        if (IN < sizeof(int32_t)) continue;
        cmd = mIn.readInt32();
        IF_LOG_COMMANDS() {
            alog << "Processing top-level Command: "
                << getReturnString(cmd) << endl;
        }


        result = executeCommand(cmd);
    }
```

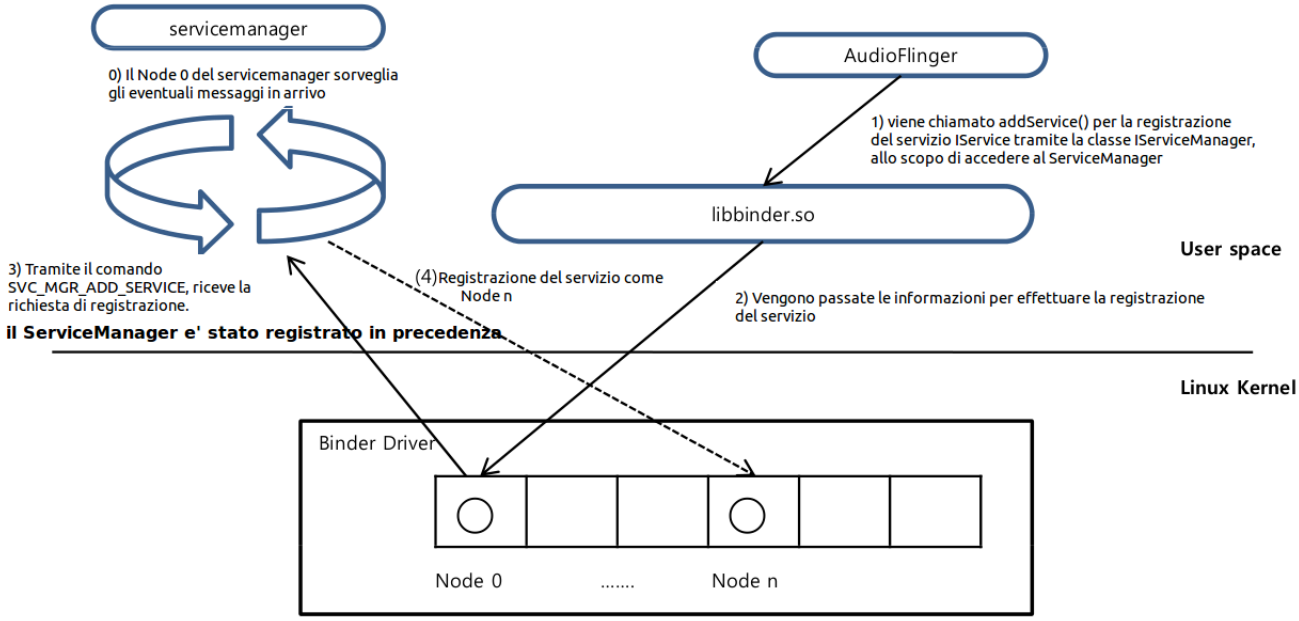
```
        // After executing the command, ensure that the thread is returned to
            the
        // foreground cgroup before rejoining the pool.  The driver takes care
            of
        // restoring the priority, but doesn't do anything with cgroups so we
        // need to take care of that here in userspace.  Note that we do make
        // sure to go in the foreground after executing a transaction, but
        // there are other callbacks into user code that could have changed
        // our group so we want to make absolutely sure it is put back.
        set_sched_policy(mMyThreadId, SP_FOREGROUND);

        // Let this thread exit the thread pool if it is no longer
        // needed and it is not the main process thread.
        if(result == TIMED_OUT && !isMain) {
            break;
        }
    } while (result != -ECONNREFUSED && result != -EBADF);

    LOG_THREADPOOL("**** THREAD %p (PID %d) IS LEAVING THE THREAD POOL err=%p\\
        n",
        (void*)pthread_self(), getpid(), (void*)result);

    mOut.writeInt32(BC_EXIT_LOOPER);
    talkWithDriver(false);
}
```

In questo modo posso inoltre evidenziare come più service possano usufruire di una stessa *main pool* condivisa all'interno di un processo *server*, come nel caso di `media_server`.

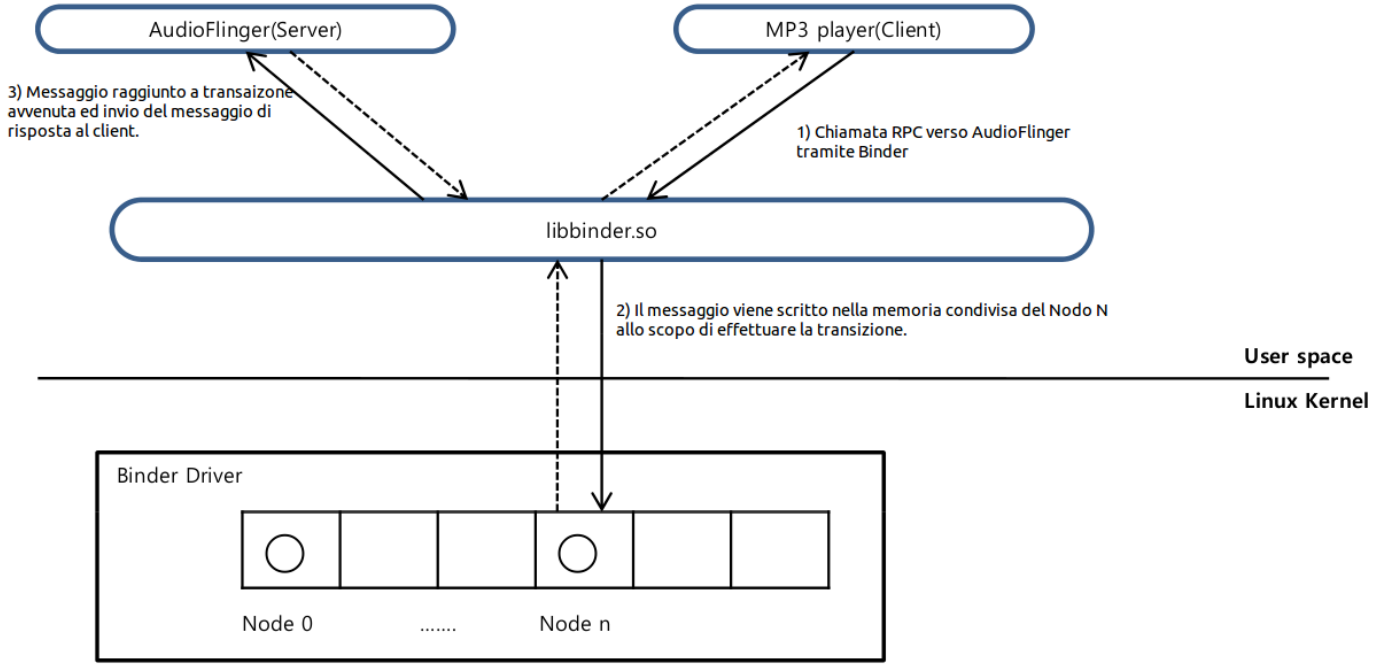
3.5.1.3. *Registrazione dei servizi lato Java.* Il questa sottosezione voglio mostrare come da codice Java sia possibile l'interazione col Binder Driver, allo scopo di effettuare la registrazione dei servizi, durante la quale avviene l'istanziazione di oggetti C++. Quando inoltre vorrò distinguere le classi Java da quelle C++, postporrò alle prime il pedice $_J$ ed alle seconde $_C$.

Mi focalizzerò ora su come avvenga la registrazione di un *service* lato Java, prendendo ad esempio quella del `PermissionController` definito all'interno dell'`ActivityManagerService`, il quale tra l'altro si occupa anche della gestione degli eventi attinenti alle *Activity* delle API Java. All'atto dell'inizializzazione del servizio `ActivityManagerService`, oltre alle restituzione del *context* di sistema tramite l'inizializzazione `init`, avviene la chiamata alla funzione `setSystemProcess` che inizializza il servizio `permission`.

Osservando la Figura 3.5.4 nella pagina precedente, possiamo ottenere una visione generale di come ciò possa avvenire: sia il client sia il *service* condividono un'area di memoria grazie alla predisposizione dei nodi tramite i quali effettuare la comunicazione.

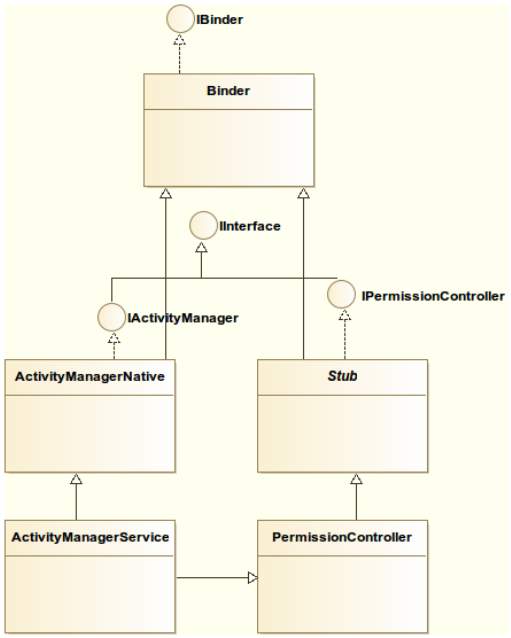


(A) *Meccanismo di registrazione di un Service.*

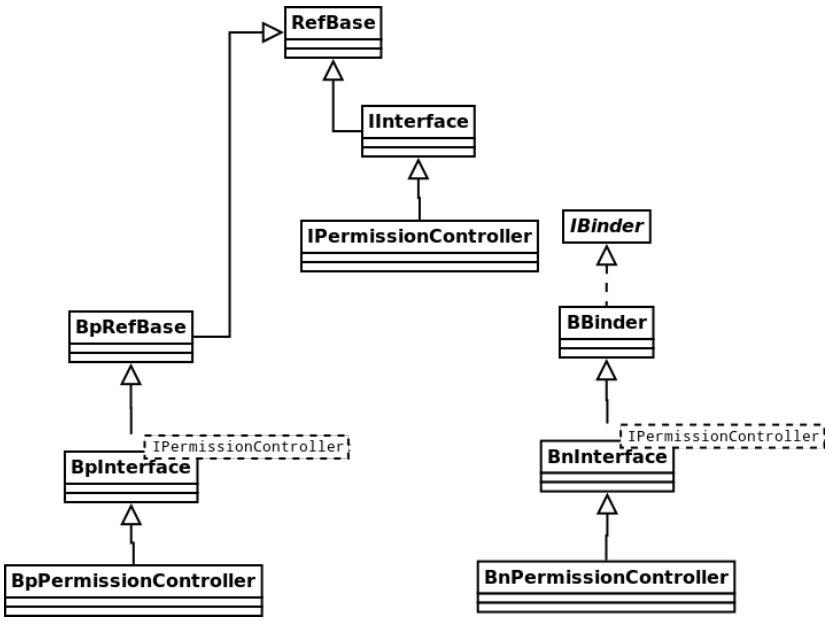


(B) *Visione High-Level del meccanismo di IPC.*

FIGURA 3.5.4. *Overview* generale dei meccanismi di comunicazione predisposti dal Binder. `http://www.aesop.or.kr/Board_Documents_Android_Frameworks/34984`



(A) *Gerarchia delle classi lato Java/Service.*



(B) *Gerarchia delle classi lato codice chiamante.*

FIGURA 3.5.5. Visione *High-Level* della gerarchia delle classi del sistema di IPC.

In quanto:

`Binder`$_J$ <: `IPermissionController.Stub`$_J$ <: `PermissionController`$_J$

il costruttore di `PermissionController` provocherà l'invocazione dei supercostruttori, tra i quali compare quello della classe madre `Binder`. Quest'ultimo provoca l'invocazione del metodo nativo `init()` che corrisponde alla chiamata della funzione `android_os_Binder_init`, dove avviene l'associazione tra la classe Java e l'oggetto nativo `JavaBBinderHolder`.

Quest'oggetto a sua volta conterrà, tramite la variabile `mBinder`, un riferimento all'oggetto `JavaBBinder`$_C$ che, estensione di `BBinder`$_C$, conterrà un riferimento `mObject` all'oggetto Java chiamante.

Faccio ora riferimento al codice di `ServiceManager` fornito nel Listato 3.7 a pagina 52; passando quindi alla procedura di ottenimento del *Service Manager* lato Java, ed in particolare all'invocazione del metodo nativo `getContextObject()` implementato nel metodo `android_os_BinderInternal_getContextObject`, subito dopo aver ottenuto il riferimento dell'oggetto `BpServiceManager`, lo si processa all'interno del metodo `javaObjectforIBinder` dove, dopo aver saltato i primi due controlli ed aver verificato il terzo, si creerà un nuovo oggetto Java `BinderProxy`$_J$ che memorizzerà al suo interno l'oggetto nativo ottenuto, e tramite il quale effettuare la richiesta al Binder.

Riassumendo i passi successivi, non necessari per l'immediato completamento della tesi[8], l'oggetto ottnenuto lato Java tramite lo strato JNI verrà poi utilizzato come argomento per il costruttore della classe `ServiceManagerProxy`$_J$; quest'ultimo oggetto instaurerà poi lato Java l'effettiva comunicazione con il driver, grazie all'associazione di questo all'attributo `mRemote`.

Volendo ora mostrare come sia possibile registrare il servizio `PermissionController`, l'invocazione di `addService` provocherà la chiamata di un metodo definito all'interno di `ServiceManagerNative.java`, il quale causerà la transazione sull'oggetto `mRemote`, provocando una procedura di inserimento dell'oggetto `BBInder`$_C$, non dissimile a quella già descritta per i *service* nativi. In fine si può osservare da codice come l'oggetto effettivamente fornito al Binder sia quello precedentemente memorizzato in `mObject`.

In questo modo ho mostrato come sia possibile, da codice Java, generare oggetti nativi che in seguito possano far riferimento all'oggetto generate.

3.5.1.4. *Invocazione di metodi Java da Native Code.* Voglio mostrare come sia possibile effettuare l'invocazione del metodo `checkPermission` lato Java da parte di codice nativo.

Sapendo che:

`Binder`$_J$ <: `IPermissionController.Stub`$_J$ <: `PermissionController`$_J$

[8] Le informazioni complete sono tuttavia fornite all'interno del sito in Cinese `http://blog.csdn.net/tjy1985/article/details/7408698`.

Listato 3.7ServiceManager.java

```
public final class ServiceManager {
    private static final String TAG = "ServiceManager";

    private static IServiceManager sServiceManager;
    private static HashMap<String, IBinder> sCache = new HashMap<String,
        IBinder>();

    private static IServiceManager getIServiceManager() {
        if (sServiceManager != null) {
            return sServiceManager;
        }

        // Find the service manager
        sServiceManager = ServiceManagerNative.asInterface(BinderInternal.
            getContextObject());
        return sServiceManager;
    }

    /**
     * Place a new @a service called @a name into the service
     * manager.
     *
     * @param name the name of the new service
     * @param service the service object
     * @param allowIsolated set to true to allow isolated sandboxed processes
     * to access this service
     */
    public static void addService(String name, IBinder service, boolean
        allowIsolated) {
        try {
            getIServiceManager().addService(name, service, allowIsolated);
        } catch (RemoteException e) {
            Log.e(TAG, "error in addService", e);
        }
    }

    /**
     * Return a list of all currently running services.
     */
    public static String[] listServices() throws RemoteException {
        try {
            return getIServiceManager().listServices();
        } catch (RemoteException e) {
            Log.e(TAG, "error in listServices", e);
            return null;
        }
    }

    //Omissis
}
```

Listato 3.8 Alcune funzioni JNI in `android_util_Binder.cpp`

```
static void android_os_Binder_init(JNIEnv* env, jobject obj)
{
    JavaBBinderHolder* jbh = new JavaBBinderHolder();
    if (jbh == NULL) {
        jniThrowException(env, "java/lang/OutOfMemoryError", NULL);
        return;
    }
    ALOGV("Java Binder %p: acquiring first ref on holder %p", obj, jbh);
    jbh->incStrong((void*)android_os_Binder_init);
    env->SetIntField(obj, gBinderOffsets.mObject, (int)jbh);
}

jobject javaObjectForIBinder(JNIEnv* env, const sp<IBinder>& val)
{
    if (val == NULL) return NULL;

    if (val->checkSubclass(&gBinderOffsets)) { ... }
    jobject object = (jobject)val->findObject(&gBinderProxyOffsets);
    if (object != NULL) { ... }
    object = env->NewObject(gBinderProxyOffsets.mClass, gBinderProxyOffsets.
        mConstructor);
    if (object != NULL) {
        LOGDEATH("objectForBinder %p: created new proxy %p !\n", val.get(),
            object);
        // The proxy holds a reference to the native object.
        env->SetIntField(object, gBinderProxyOffsets.mObject, (int)val.get());
        val->incStrong(object);
        //Omissis
    }
}

sp<IBinder> ibinderForJavaObject(JNIEnv* env, jobject obj)
{
    if (obj == NULL) return NULL;

    if (env->IsInstanceOf(obj, gBinderOffsets.mClass)) {
        JavaBBinderHolder* jbh = (JavaBBinderHolder*)
            env->GetIntField(obj, gBinderOffsets.mObject);
        return jbh != NULL ? jbh->get(env, obj) : NULL;
    }

    //Omissis
}

static jobject android_os_BinderInternal_getContextObject(JNIEnv* env, jobject
    clazz)
{
    sp<IBinder> b = ProcessState::self()->getContextObject(NULL);
    return javaObjectForIBinder(env, b);
}
```

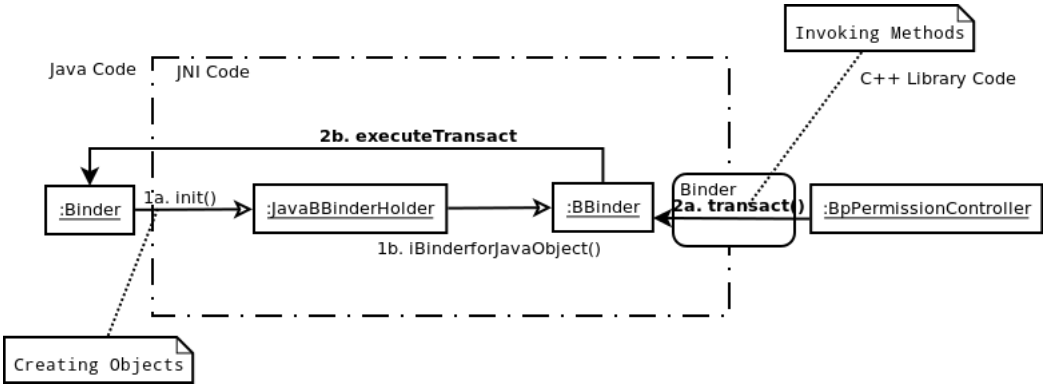


FIGURA 3.5.6. *Visione dell'interazione tra oggetti interagenti nel corso dell'IPC per i* service *Java*.

abbiamo che tale metodo dichiarato all'interno della classe `PermissionController` mostrata in precedenza nel Listato 3.2 a pagina 36, verrà richiamato dal metodo `onTransact` definito in `Stub`, a sua volta richiamato dal metodo `executeTransact` definito all'interno della classe `Binder`.

Ricercando ora come tale metodo possa essere richiamato a livello di JNI, si scopre che questa invocazione viene consentita all'interno del metodo `onTransact` come definito nella classe `JavaBBinder`, a sua volta richiamato dal metodo `transact` definito all'interno della classe `BBinder`, dove:

$$\texttt{BBinder}_{\texttt{C}} \texttt{ <: } \texttt{JavaBBinder}_{\texttt{C}}$$

Questo metodo verrà poi richiamato da del *main loop* fornito lato JNI dal `system_server`, come già mostrato in precedenza per i *service* nativi.

3.6. Privilegi del superutente, differenze e limitazioni architetturali

Come possiamo notare dall'esecuzione della *systemcall* `uname`, la versione del kernel installata all'interno dell'emulatore Android con versione 4.0.3 è la `2.6.36-000064-g8ed501`: possiamo quindi vedere come il Kernel di Android postponga dei numeri di versione aggiuntivi, che la distinguono dal Kernel Linux standard.

Posso inoltre verificare come, all'interno del sistema Android, siano presenti dei problemi di configurazione delle interfacce di rete e di come `iptables` sia disponibile, ma non reso fruibile all'utente [GJG12]. Ho raccolto infatti da più parti il seguente messaggio di errore:

```
W/ThrottleService(  178): unable to find stats for iface rmnet0
```

Dall'analisi del codice sorgente dei *framework*, noto che questo errore è generato dalla funzione `onPollAlarm` di:

`$AOSP/framework/base/services/java/com/android/server/ThrottleService.java`

dove si richiamano le istanze di `NetworkStats` generate dalla classe `NetworkManagerService`. In particolare viene chiamato il metodo `enforceCallingOrSelfPermission`, il quale a sua volta restituisce il valore del metodo `readNetworkStatsSummaryDev` della classe `NetworkStatsFactory`. All'interno della funzione si effettua la lettura, tramite classe `ProcFileReader` preposta alla lettura del *filesystem* virtuale `/proc`, il file `/proc/net/xt_qtaguid/iface_stat_all`, che però non sembra essere disponibile nell'emulatore con Android 4.0.3, ma presente nel Galaxy Nexus con Android 4.1.

A quanto pare, tale errore si verificherebbe sia nei dispositivi dove tale file è assente, sia in dispositivi nei quali, per il momento, non è stata attivata tale interfaccia di rete. Inoltre la classe `ThrottleService`, responsabile della generazione del suddetto errore, sarebbe utilizzata unicamente dal `SystemServer`, che lo inizializza utilizzando il `Context` corrente e l'interfaccia contenuta all'interno della risorsa `R.string.config_datause_iface`, che identifica l'interfaccia di default sulla quale monitorare l'utilizzo dei dati. Segue conseguentemente che, quei messaggi, siano frutto di un costante monitoraggio di tale interfaccia, allo scopo di verificare quando questa sia attivata. Questo mostrerebbe quindi un costante interesse, da parte del sistema, sull'utilizzo dell'interfaccia di rete 3G. Si può infatti notare dal file

`$AOSP/frameworks/base/core/res/res/values/config.xml`

la definizione del valore `R.string.config_datause_iface` come segue:

```
<string name="config_datause_iface" translatable="false">rmnet0</string>
```

Un'ulteriore importante differenza che contraddistingue questo sistema dagli altri GNU/Linux è la completa assenza del file `/etc/passwd`: l'elenco degli utenti di sistema, dei permessi di accesso al FileSystem è contenuta all'interno del file:

`$AOSP/system/core/include/private/android_filesystem_config.h`

Abbiamo quindi che i permessi di sistema vengono compilati all'interno del dispositivo: da ciò segue che se vogliamo modificare definitivamente l'associazione tra `gid` ed ANDROID ID, come mostrato dall'array:

```
static const struct android_id_info android_ids[] = {...};
```

o del *filesystem* come indicato dagli array:

```
static struct fs_path_config android_dirs[] = {...};
static struct fs_path_config android_files[] = {...};
```

si rende necessaria la ricompilazione del Kernel. Un altro file interessante per quanto riguarda la relazione tra permessi di `gid` e permessi richiesti all'interno del Manifest file per le applicazioni Java o Native JNI è il seguente:

`$AOSP/frameworks/base/data/etc/platform.xml`

In quella sede vengono anche descritti quali permessi vengono garantiti alla *shell*; possiamo quindi osservare come l'acquisizione dei permessi di root permetta indirettamente di acquisire tutti i permessi che sarebbero da richiedere all'interno di un Manifest File.

3.6.1. Differenze architetturali: OpenSLES. *In questo punto cercherò di difendere l'ipotesi secondo la quale le librerie interne del sistema operativo Android veicolino indirettamente l'effettuazione dei controlli sui permessi di sistema.* Quest'occasione viene fornita dall'errore riportato dal LogCat durante l'esecuzione di `pjsua` senza particolari privilegi utente, ed evidenziato nella Sezione 5.2 a pagina 74. In questa sede non descriverò la libreria OpenSLES, per la quale faccio riferimento alla Sottosezione 6.2.3 a pagina 84, ma metterò in luce il collegamento esistente tra librerie a supporto del codice nativo e funzioni di monitoraggio del sistema.

In questa sede mi riferirò ad OpenSLES come ad Wilhelm (`$AOSP/frameworks/wilhelm`), in quanto in questo punto si effettua l'implementazione di OpenSLES e si forniscono gli *include* all'interno del percorso `$AOSP/frameworks/wilhelm/include/SLES`.

Ripartirò ora con la trattazione facendo riferimento all'analisi degli errori forniti dal LogCat, e che in particolare fanno riferimento allo strato delle librerie native. Si può vedere come l'output:

android_audioRecorder_realize(...) error creating AudioRecord object

venga fornito all'interno della funzione `wilhelm android_audioRecorder_realize`, definito nel file `AudioRecorder_to_android.cpp`: si può immediatamente notare l'invocazione del costruttore `android::AudioRecord()`, il quale in realtà appartiene ad una classe di un'altra libreria, detta *libmedia*, e definito all'interno del file:

`$AOSP/frameworks/av/media/libmedia/AudioRecord.cpp`

L'invocazione di tale costruttore implica l'invocazione del metodo `set`, causando l'invocazione del metodo `openRecord_l` che è il responsabile dell'output di errore:

AudioFlinger could not create record track, status: ...

In particolare questo è il risultato della seguente invocazione:

```
sp<IAudioRecord> record = audioFlinger->openRecord(getpid(), input, sampleRate,
    format,channelMask,frameCount,IAudioFlinger::TRACK_DEFAULT,&mSessionId,&
   status);
```

dove ci si preoccupa di controllare preventivamente i permessi tramite l'invocazione del metodo `recordingAllowed()` di `ServiceUtilities.cpp`, il quale a sua volta effettua l'invocazione di `checkCallingPermission("android.permission.`

RECORD_AUDIO"). Il tipo di dato che è definito come sp è in particolare l'implementazione degli *Strong Pointer* che, assieme ai *Weak Pointers* definiti nel file:

$AOSP/frameworks/native/libs/utils/RefBase.cpp

implementano tramite "smart pointers" il *Reference Counting* all'interno del codice C++[9]. È comunque a questo livello, ovvero di libreria *libnbaio*, presente tra altre all'interno del percorso:

$AOSP/frameworks/av/services/audioflinger/

che noto le prime chiamate a funzioni della libreria *libbinder*, i cui sorgenti sono situati all'interno del percorso:

$AOSP/frameworks/native/libs/binder/

Riporto l'inclusione dello header di ServiceUtilities.cpp:

```
#include <binder/IPCThreadState.h>
#include <binder/IServiceManager.h>
#include <binder/PermissionCache.h>
```

Sebbene il metodo checkCallingPermission sia definito sia nella classe IServiceManager, sia in PermissionCache, solamente nella prima si ha l'inclusione di tale metodo all'interno del *namespace* android, che quindi riporto qui sotto:

```
bool checkCallingPermission(const String16& permission, int32_t* outPid,
    int32_t* outUid)
{
    IPCThreadState* ipcState = IPCThreadState::self();
    pid_t pid = ipcState->getCallingPid();
    uid_t uid = ipcState->getCallingUid();
    if (outPid) *outPid = pid;
    if (outUid) *outUid = uid;
    return checkPermission(permission, pid, uid);
}
```

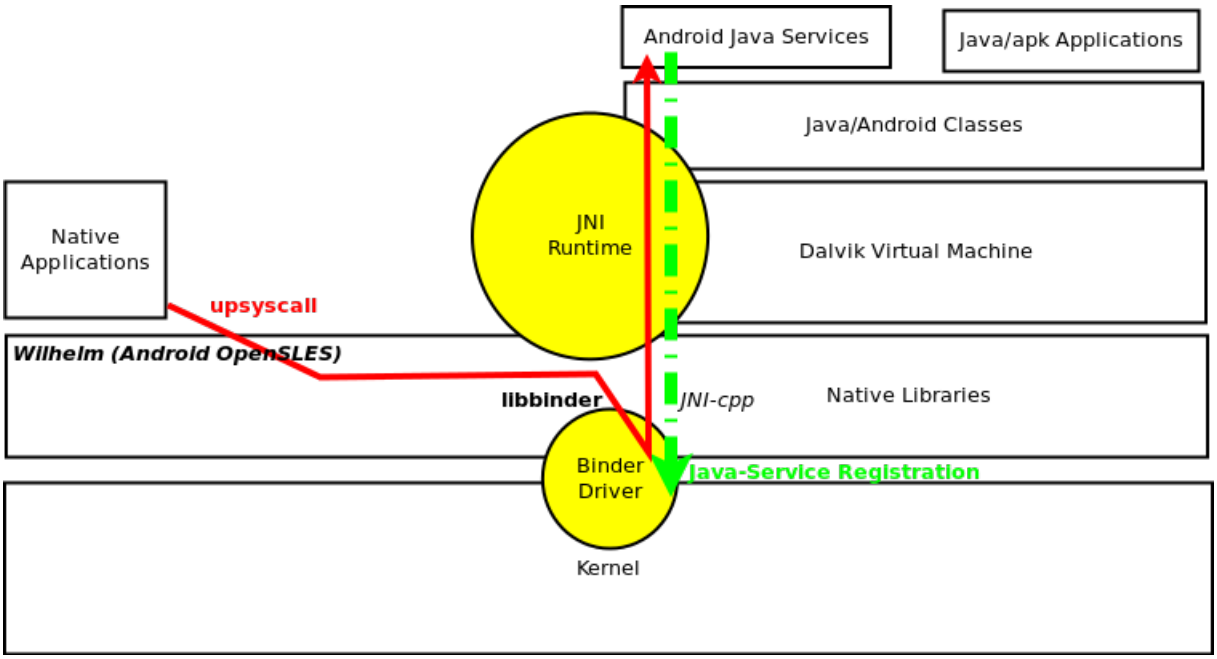
Proseguendo l'analisi dell'output fornito, ottengo che il seguente messaggio:

Permission failure: android.permission.RECORD_AUDIO from uid=10043 pid=2978

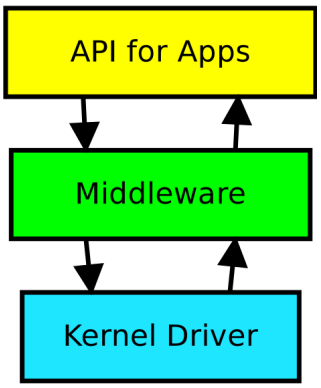
è prodotto ancora una volta dalla classe C++ IServiceManager, della quale metto in evidenza il codice:

```
// Is this a permission failure, or did the controller go away?
if (pc->asBinder()->isBinderAlive()) {
   ALOGW("Permission failure: %s from uid=%d pid=%d", String8(permission).
       string(), uid, pid);
   return false;
}
```

[9]Per non soffermarmi eccessivamente su questo aspetto secondario, faccio riferimento al sito http://www.icepack-linux.com/android-smart-pointer/



(A) *Descrizione finale della gerarchia Android. Sono messi in luce i meccanismi di registrazione del servizio e di IPC.*



(B)
Visione High-Level dell'architettura Android. [Sch11].

FIGURA 3.6.1. Visione dell'interazione tra oggetti interagenti nel corso dell'IPC.

Ciò mostra che effettivamente il Binder è esistente, ma che la restituzione della chiamata a funzione è `false`.

Ritornando ora a seguire il *fil rouge*, otteniamo che la chiamata al `checkPermission` remoto viene garantita dalla chiamata all'omonima funzione definita all'interno di

`IPermissionController`, che effettua l'inserimento dei dati all'interno del `Parcel` e richia il metodo:

```
remote()->transact(CHECK_PERMISSION_TRANSACTION, data, &reply);
```

In questo caso `remote()` non è nient'altro che l'oggetto `mRemote BpBinder` e fornito dal Binder per effettuare la comunicazione tramite l'omonimo driver, e quindi si riuscirà ad interagire con il service registrato, che in particolare risponde al descrittore `android.os.IPermissionController`. Come conclusione posso fornire il diagramma degli oggetti proposto in Figura 3.5.6 a pagina 54, palesando così l'effettuazione delle *upsyscall*.

```
static class PermissionController extends IPermissionController.Stub {
        ActivityManagerService mActivityManagerService;
        PermissionController(ActivityManagerService activityManagerService) {
            mActivityManagerService = activityManagerService;
        }

        public boolean checkPermission(String permission, int pid, int uid) {
            return mActivityManagerService.checkPermission(permission, pid,
                    uid) == PackageManager.PERMISSION_GRANTED;
        }
}
```

Posso quindi ribadire i concetti fin'ora espressi tramite l'ausilio dell'immagine (a) 3.6.1 a fronte, che in pratica dettaglia quello che le fonti avevano già espresso con la Figura (b) 3.6.1 nella pagina precedente.

3.6.2. Limitazioni: Gestione dei driver audio. *L'obiettivo di questa sottosezione è quello di dimostrare come, all'interno dello stesso sistema operativo messo a disposizione da Google, siano presenti forti limitazioni architetturali, che impongono la necessità di applicare estensioni allo stesso sistema operativo.*

Per capire meglio il meccanismo di lettura da `AudioRecorder`, debbo descrivere inizialmente il meccanismo di inizializzazione della procedura. Partendo quindi dalla funzione della libreria `wilhelm android_audioRecorder_realize`, si effettua l'invocazione del metodo `set` dell'oggetto `AudioRecord` definito dall'omonimo file presente all'interno del percorso:

`$AOSP/frameworks/av/media/libmedia`

Accedendo al codice sorgente, possiamo notare come non venga passato un *session Id* (o meglio come venga implicitamente passato il valore 0, allo scopo di generarne uno automaticamente con il metodo `AudioSystem::newAudioSessionId()`) e come venga passato come argomento `cbf` la funzione `audioRecorder_callback` fornita da `wilhelm`, in modo da fornire alla libreria il *sampling* audio ottenuto.

Analizzando ora il metodo `AudioSystem::getInput()`, possiamo notare come venga ad esso passato un numero nuovo di sessione, e di come si invochi il

metodo `getInput` di `AudioPolicyService` dopo aver richiesto al Binder un *proxy* per instaurare la comunicazione con detto *service*.

Dopo aver osservato che questo *service*, come da implementazione all'interno del file:

```
$AOSP/frameworks/av/services/audioflinger/AudioPolicyService.cpp/h
```

effettua l'estensione della classe `BnAudioPolicyService`, si può notare come, nel metodo ivi definito, l'operazione basilare sia quella di ottenere il puntatore alla funzione di ottenimento dell'input dalla scheda audio.

Analizzando quindi la procedura `set` della classe `AudioRecorder`, si effettua l'invocazione del metodo `openRecord_l`, dove ancora una volta si ottiene con il metodo `AudioSystem::get_audio_flinger()` un'istanza del proxy `BpBinder` per comunicare con il *service* `AudioFlinger`.

Come posso osservare dall'invocazione del metodo `checkRecordThread_l`, è necessario che si verifichi la creazione di un `RecordThread`. Tuttavia non è immediato mostrare come sia possibile inserire tale thread. *Voglio quindi mostrare come, oltre all'utilizzo del Binder, anche i metodi di classi possano mettere in comunicazione diversi oggetti tramite puntatori a funzione definiti all'interno di strutture dati C, che in particolare contengono puntatori a funzioni che consentono l'IPC tramite Binder*.

Analizzo quindi il costruttore della classe `AudioPolicyService`. Per quanto concerne l'inizializzazione, devo far ancora riferimento al *server* principale, ovvero il `mediaserver` di cui si è già discusso il codice.

Faccio riferimento alla figura 3.6.2 a fronte per esporre in un modo più chiaro i meccanismi di chiamata di quanto non sia possibile fare a parole, col rischio di perdere l'idea del susseguirsi delle chiamate a funzione. *In particolare il meccanismo di apertura del modulo mette in luce una potenziale fragilità del sistema: sarebbe infatti sufficiente cambiare il modulo `.so` per cambiare il comportamento delle funzioni che seguono*.

Ritornando quindi per un momento all'invocazione del metodo `AudioPolicyService::getInput`, otteniamo la sequenza di invocazioni illustrata nella Figura 3.6.3 a pagina 62, le quali consentono la creazione di un nuovo thread per la registrazione, tramite l'apertura di uno *stream* audio in entrata dal driver, non mostrato per motivi di brevità e ad ogni modo osservabile all'interno del metodo `AudioFlinger::openInput()`, e comunque ottenibile tramite la funzione il cui puntatore è contenuto nell'attributo `mInput` del Thread in questione.

Posso quindi palesare con la Figura 3.6.4 a fronte come questo meccanismo di chiamata si riassuma nella chiamata ad `openInput()`, che tuttavia poteva essere già effettuata a livello di `AudioPolicyService` tramite l'utilizzo diretto del meccanismo di IPC. Si può inoltre notare dal codice sorgente di `AudioFlinger.cpp`

```
struct audio_policy_service_ops aps_ops = {
  ...
  open_input : aps_open_input
  ↳ AudioSystem::get_audio_flinger()->openInput(...)
    ↳ AudioFlinger::openInput(...)
      ↳ mRecordThreads.add(id,new RecordThread(this,...));
  ...
  open_input_on_module : aps_open_input_on_module
  ↳ AudioSystem::get_audio_flinger()->openInput(...)
    ↳ ...
} [./frameworks/av/services/audioflinger/AudioPolicyService.cpp]

AudioPolicyService::istantiate() [AudioFlinger]
↳ AudioPolicyService::AudioPolicyService()
  ↳ {
      hw_get_module(AUDIO_POLICY_HARDWARE_MODULE_ID,&module);
      ↳ AUDIO_POLICY_HARDWARE_MODULE_ID = audio_policy [audio_policy.h]
        Il nome del binario dalla compilazione di hardware/libhardware_legacy/
            audio
        ↳ hw_get_module [hardware.c]
          ↳ hw_get_module_by_class
            ↳ Caricamento dei moduli di nome AUDIO_POLICY_HARDWARE_MODULE_ID
                tramite dlopen

      audio_policy_dev_open(module,&mpAudioPolicyDev) [definito in
          audio_policy.h]
      ↳ module->methods->open(module,"policy",...)
        dove open = default_ap_dev_open [audio_policy.c]

      mpAudioPolicyDev->create_audio_policy(mpAudioPolicyDev,&aps_ops,this
          ,...);
        ↳ create_legacy_ap(mpAudioPolicyDev,&aps_ops,this,..) [
            audio_policy_hal.cpp]
        {
          lap->service_client = new AudioPolicyCompatClient(aps_ops, service)
              [AudioPolicyCompatClient.cpp]
          ↳ mServiceOps:=&aps_ops mService:=this
          lap->apm = createAudioPolicyManager(lap->service_client)
          ↳ AudioPolicyManager(lap->service_client) [AudioPolicyManager.cpp]
              sottoclasse di:
            ↳ AudioPolicyManagerBase::AudioPolicyManagerBase(lap->
                service_client)
              ↳ mpClientInterface = lap->service_client
        }
  }
```

FIGURA 3.6.2. *Inizializzazione delle strutture dati C di collegamento.*

```
AudioPolicyService::getInput()
↳ mpAudioPolicy->get_input()
  ↳ lap->apm->getInput() [audio_policy_hal.cpp] (ovvero AudioPolicyManagerBase)
    ↳ AudioPolicyManagerBase::getInput()
      ↳ mpClientInterface->openInput() [AudioPolicyManagerBase.cpp]
        ↳ AudioPolicyCompatClient::openInput()
          ↳ mServiceOps->open_input_on_module() [AudioPolicyCompatClient.cpp]
            ↳ aps_open_input_on_module() [AudioPolicyService.cpp]
              ↳ AudioFlinger::openInput()
                ↳ mRecordThreads.add(id,new RecordThread(this,...))
```

FIGURA 3.6.3. *Inizializzazione nascosta del thread di registrazione.*

come la creazione di tale *stream* sia assolutamente indipendente dal chiamante, in quanto non viene conservata alcuna informazione in merito.

In questo modo ho finito di dettagliare la procedura di inizializzazione, e di come si utilizzino librerie di basso livello per celare il collegamento tra di esse tramite Binder.

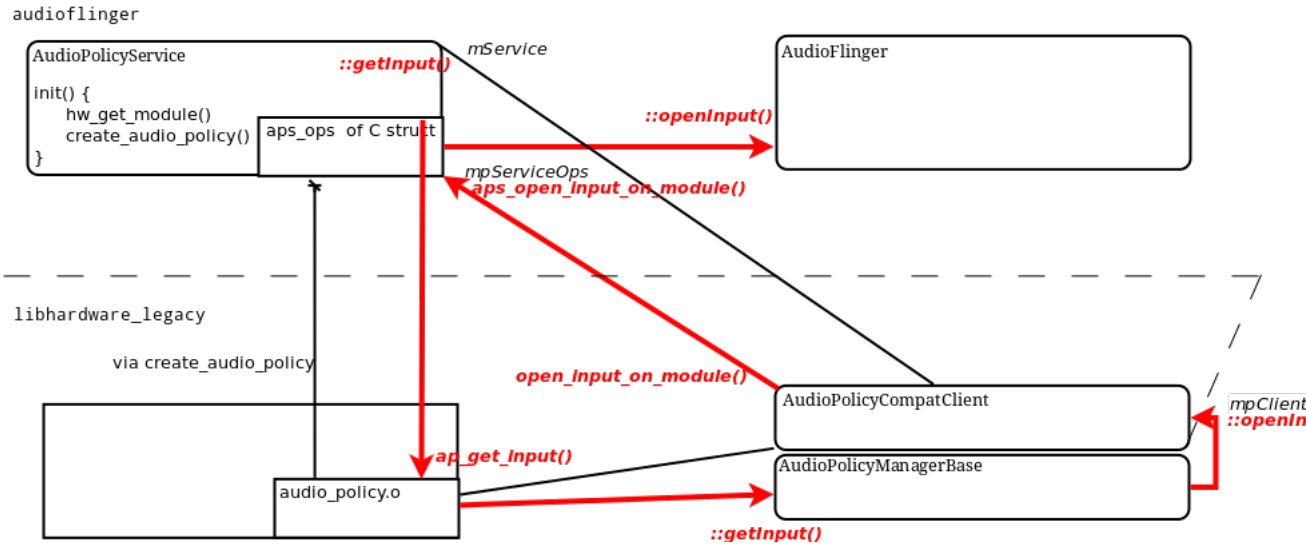


FIGURA 3.6.4. *Inizializzazione nascosta del thread di registrazione.*

Ritornando all'invocazione del metodo `AudioFlinger::openRecord`, possiamo ora vedere come il thread ottenuto dalla funzione `checkRecordThread_l` sia in particolare il risultato delle complesse interazioni mostrate sopra.

Tramite l'invocazione del metodo `registerPid_l` si provoca, se non già esistente, un istanza di un oggetto `Client`, il quale tramite l'istanziazione di un `MemoryDealer`, ottiene della memoria allocata dal servizio *ashmem* in modo che questa possa essere condivisa tra processi differenti.

Eseguendo quindi sul suddetto thread il metodo `createRecordTrack` che, dopo una serie di invocazione di metodi, effettua l'inizializzazione della memoria precedentemente allocata: l'oggetto ottenuto come risultato di questa invocazione sarà utilizzato come argomento del costruttore dell'oggetto di tipo `RecordHandle`, che si preoccuperà in seguito della lettura da interfaccia.

Concludendo quindi la procedura di inizializzazione tramite invocazione del metodo `set()`, dopo aver invocato `AudioSystem::getInput()` e `openRecord_l` nella classe `AudioRecord`, si procede alla creazione di un'istanza di oggetto `ClientRecordThread`, associata alla variabile `mClientRecordThread`.

Proseguo ora l'analisi con l'invocazione del metodo `start()` sull'oggetto `mAudioRecord` a livello di `wilhelm`, invocato all'interno della funzione `SetRecordState`; come posso notare dal codice di `AudioRecord.cpp`, in questo metodo vengono iniziati sia il thread di lettura del client, sia quello in lettura dal driver audio: si può quindi notare da codice come questi comunichino tramite l'utilizzo dell'area di memoria allocata tramite il servizio *ashmem*. Il meccanismo di Figura (a) 3.6.6 a pagina 65 delinea infine l'interazione tra il RecordThread che ottiene i dati dalla periferica ed il ClientThread che, tramite la callback, fornisce i dati ottenuti alla libreria Wilhelm.

Ho messo in questo modo in evidenza come, il codice sopra definito, non consenta minimamente una politica di caching per il livello di applicazioni native, allo scopo di far attingere ai dati i registratori. Ad ulteriore riprova è il messaggio di errore fornito dall'esecuzione di due client `pjsua` e riscontrabile all'interno del LogCat:

```
W/AudioRecord(  963): obtainBuffer timed out (is the CPU pegged?) user
    =00001080, server=00001080
W/AudioTrack(  963): obtainBuffer timed out (is the CPU pegged?) 0x15e16d8 name
    =0x2user=000011ff, server=000009bf
W/AudioTrack(  963): obtainBuffer timed out (is the CPU pegged?) 0x15e16d8 name
    =0x2user=000011ff, server=000009bf
```

Questo messaggio è causato dalla mancanza di frame in lettura, come riscontrabile all'interno dalla `while (framesReady==0)` all'interno del metodo `AudioRecord::obtainBuffer` usato dal servizio in lettura sull'area di memoria condivisa con l'altro Thread, dopo un prefissato tempo d'attesa. Ciò implica che uno dei due thread non farà mai in tempo ad accedere alle informazioni che pervengono dal microfono del dispositivo, rimanendo sempre quindi in attesa di ricevere ulteriori informazioni.

Posso di fatti ampliare la gerarchia di Android con la Figura (b) 3.6.6 a fronte, mostrando anche l'interazione tra *service* nativi ed applicazioni native.

Posso inoltre riassumere con la Figura 3.6.5 l'interazione che avviene tra *client* e *service* contenuto all'interno di un processo *server*.

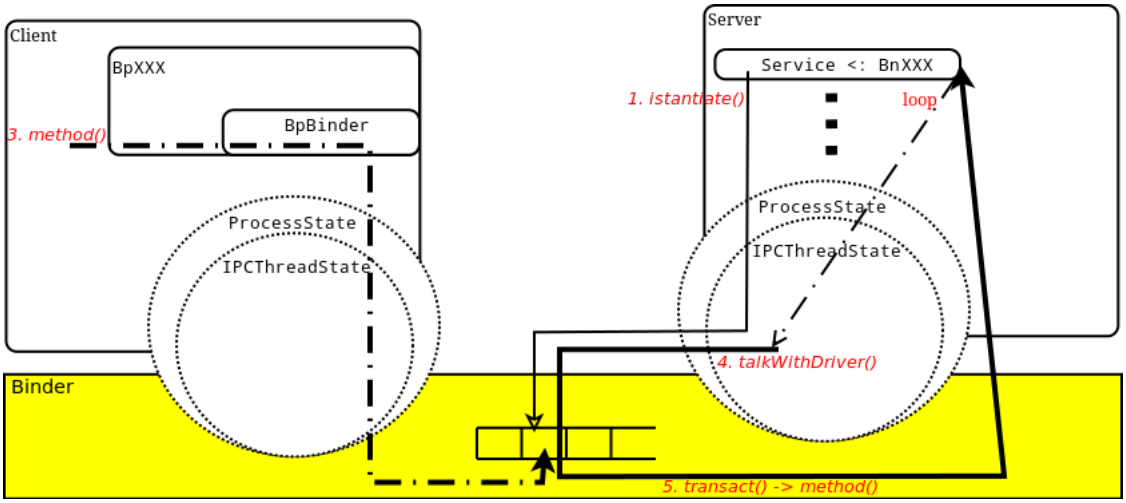
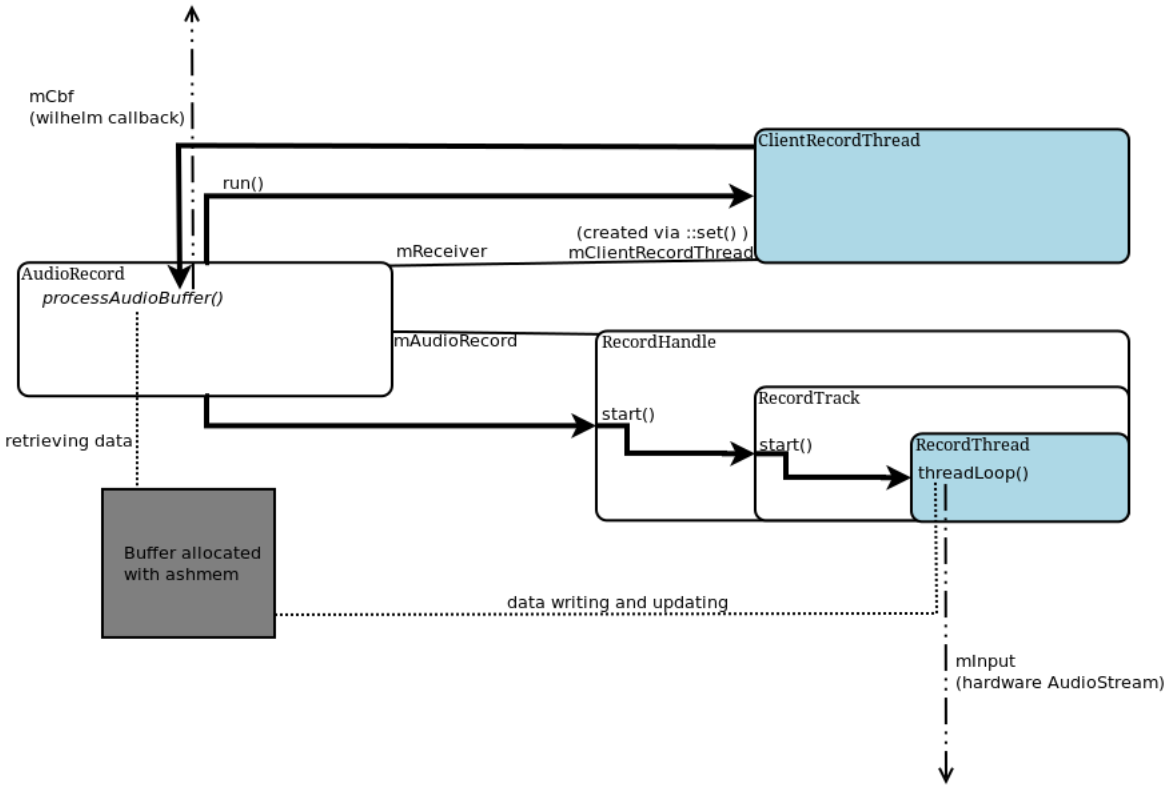
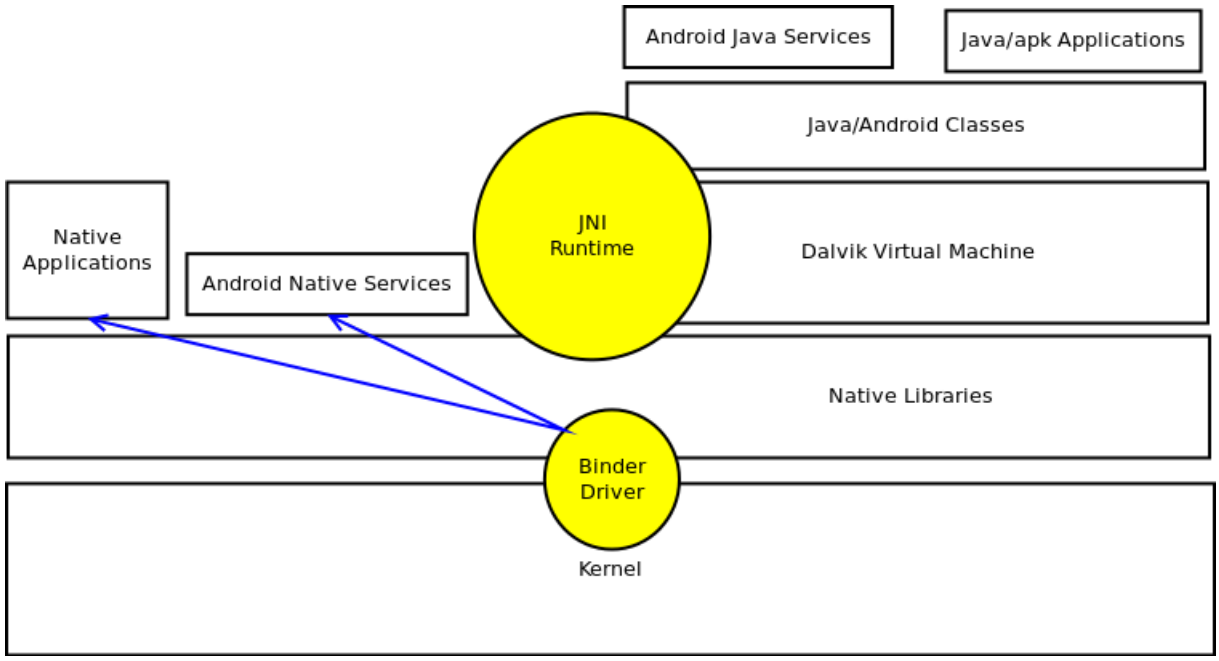


FIGURA 3.6.5. *Visione d'insieme di interazione tra client e server.*



(A) *Interazione tra i thread di lettura dal driver e di callback verso la libreria wilhelm.*



(B) *Ampliamento dell'architettura Android con i Native Service.*

FIGURA 3.6.6. Conclusioni sui metodi di interazione.

CAPITOLO 4

Utilizzo dei tool di Android

Indice

4.1. Primi passi con Android SDK ed installazione dell'emulatore

Per iniziare, è opportuno scaricare e decomprimere l'SDK e l'NDK di Android `http://developer.android.com/sdk/index.html`. In particolare l'SDK fornisce le API per quanto riguarda le applicazioni Java ed i tool di sviluppo in genere per gestire le macchine reali o gli emulatori gestiti dal server `adb` (ANDROID DEBUG BRIDGE). Per l'installazione di tali API, comunque necessarie per l'utilizzo dell'emulatore, si fa riferimento all'applicazione:

```
android-sdk-linux/tools/android sdk
```

verrà visualizzata un'interfaccia grafica dalla quale scegliere quali API di sistema installare. D'ora in poi ci riferiremo ad `android-sdk-linux` come ad `$SDK`.

Una volta ultimata questa procedura, si può passare all'installazione dell'emulatore. Per prima cosa è necessario creare una sdcard virtuale tramite l'ausilio del programma `mksdcard`:

```
$SDK/tools/mksdcard size outfile
```

Su questa verrà montato il *filesystem* FAT32. Se questa non verrà creata o comunque non verrà utilizzata per la macchina virtuale, si renderà il *filesystem* della macchina emulata di sola lettura, per quanto riguarda la creazione di nuovi files su di essa.

Per la creazione di una macchina virtuale (ANDROID VIRTUAL DEVICE), si utilizza invece il seguente binario:

```
$SDK/tools/android create avd -n new_machine_name -t api -sdcard file
```

dove il parametro `api` serve per specificare quale versione di Android emulare all'interno della macchina. La procedura di creazione dell'`avd` può essere effettuata anche tramite interfaccia grafica, accessibile tramite il comando `$SDK/tools/android avd`. Per ottenere invece l'elenco di tutte le macchine virtuali che sono state create, è sufficiente lanciare l'applicazione precedente con argomento `list avd`, mentre per visualizzare i device collegati ed attivi è necessario eseguire il seguente programma:

```
$SDK/platform-tools/adb devices
```

È inoltre possibile rendere visibile un device Android non emulato ma reale all'interno della lista degli eventuali dispositivi in esecuzione, sempre ottenibile tramite il comando fornito in precedenza. Con l'`adb` è inoltre possibile, tramite il comando `logcat`, accedere alle informazioni di logging fornite dagli applicativi in genere, ed dei *service* di sistema in particolare.

4.2. Interazione con i device Android

L'interazione con i dispositivi Android è mediata dall'`adb`. Questo programma, che abbiamo già visto occuparsi in precedenza della lista dei dispositivi attivi, può anche consentire il trasferimento dei files dal dispositivo dove è in esecuzione l'SDK all'emulatore (istruzione `push filereal fileemu`) o viceversa (istruzione `pull fileemu filereal`), dell'interazione con la shell del dispositivo (`shell`: a questo comando si possono far eventualmente seguire le istruzioni da effettuare) o la visualizzazione del file di logging (`logcat`). Tutto ciò è possibile previa definizione del nome del dispositivo con il quale interagire tramite il flag `-s`.

Per effettuare queste operazioni più agilmente, ho ideato alcuni semplici script, che sono comunque riportati in Appendice nella Sezione B.1 a pagina 117.

4.2.1. Comunicazione tra due emulatori Android, all'interno della stessa macchia. *L'analisi dei metodi di interazione degli emulatori è stata necessaria in quanto, in un primo tempo, si credeva di poterli utilizzare per il testing del driver audio.*

L'emulatore predispone, all'interno dell'ambiente di sviluppo da esso fornito, un'interfaccia di default per l'interazione con l'ambiente esterno, compreso l'accesso alla rete, con l'indirizzo di rete `10.0.2.15`; in particolare questa è un'interfaccia di rete 3G. Oltre all'indirizzo di loopback standard (`127.0.0.1`), l'Emulatore prevede anche la possibilità di accedere al loopback della macchina ospite tramite l'indirizzo `10.0.2.2`.

Ogni istanza dell'emulatore predispone due porte all'intero del localhost della macchina ospite, una con una cifra pari (utilizzata per la comunicazione Telnet dalla macchina ospite: quest'ultima verrà utilizzata di seguito per interagire con i processi di rete localizzati all'interno dell'emulatore), ed un'altra con

cifra dispari per la comunicazione con il server `adb` che gestisce l'interazione tra i dispositivi[1].

Come programma Client/server per il testing della comunicazione, si è proceduto ad utilizzare un semplice programma Client/Server, il cui codice è riportato all'interno dell'Appendice nella Sezione B.3 a pagina 122.

Utilizzando la comunicazione *telnet* sulla porta pari dell'emulatore detta sopra (nel caso d'esempio `5554`) tramite il comando:

```
telnet localhost 5554
```

possiamo ottenere, tramite il comando `help`, quali comandi possiamo utilizzare per interagire con il dispositivo. Li riporto di seguito:

```
Android console command help:

    help|h|?         print a list of commands
    event            simulate hardware events
    geo              Geo-location commands
    gsm              GSM related commands
    cdma             CDMA related commands
    kill             kill the emulator instance
    network          manage network settings
    power            power related commands
    quit|exit        quit control session
    redir            manage port redirections
    sms              SMS related commands
    avd              control virtual device execution
    window           manage emulator window
    qemu             QEMU-specific commands
    sensor           manage emulator sensors
```

Possiamo quindi notare come, tramite il comando `redir`, possiamo effettuare la redirezione delle porte del dispositivo su quelle della macchina ospite. In particolare eseguendo il comando:

```
redir add tcp:12345:12345
```

un client in esecuzione sulla macchina reale può interagire con un server in esecuzione sull'emulatore, oppure un client, all'interno di un emulatore, può interagire con un altro server su di un altro emulatore, poiché può accedere alla porta in questo modo rediretta sul localhost della macchina ospite.

Riporto di seguito le considerazioni da me effettuate prima di ottenere il risultato di cui sopra:

- Avviando un server all'interno della macchina ospite in ascolto sulla porta 12345 ed un client sull'emulatore che effettua una richiesta al server 10.0.2.2:12345, ho notato come le richieste arrivino al server come se il processo provenisse dalla macchina reale (in particolare, nella prova

[1] `http://developer.android.com/tools/help/adb.html`.

effettuata, provenivano dall'indirizzo/porta `127.0.0.1:45430`). Nelle prove effettuate, questa porta non coincideva con le due porte di default predisposte per l'emulatore sopra descritte.

- Avviando un server all'interno della macchina emulata in ascolto sulla porta 12345 ed un client sul quella ospite, se si prova ad accedere da quest'ultima al server con indirizzo `127.0.0.1:12345`, si ottiene invece un Connection Refused. Utilizzando infatti il comando `lsof -i :12345` per identificare se, all'interno della macchina ospite, sia presente un processo con quella porta, non si trova alcun `pid` che lo possegga.

  Accedendo invece alle altre due porte di default, la connessione viene accettata ma non arriva a destinazione sul server.
- Se si disabilitano le interfacce di rete tramite la "Modalità Aereo", è sempre e comunque possibile accedere all'indirizzo di loopback all'interno del dispositivo: se così non fosse, non si renderebbe possibile la comunicazione tra servizi che, in parte, avviene tramite socket locali.

4.3. NDK: Tool di cross-compilazione Android

Tratteremo in questa sezione di cross-compilazione, ed in particolare dei tool forniti da Android. In precedenza veniva utilizzata la toolchain SOURCERY CODEBENCH LITE FOR ARM EABI, che tuttavia non contiene di di default né le librerie presenti all'interno dei dispositivi Android, né i file di inclusione deducibili dal suo Kernel.

In questa sede tuttavia discorreremo del tool di crosscompilazione NDK fornito dalla stessa Google: il vantaggio principale di questa collezione di tool è quello di fornire sia le librerie, sia i file di include necessari alla cross-compilazione, senza avere la necessità di cross-compilarle o di ottenerle da un altro dispositivo.

Questa tesi utilizza la versione di "r8b" di questi tool, che sono comunque ottenibili dal seguente indirizzo internet:

`http://dl.google.com/android/ndk/android-ndk-r8b-linux-x86.tar.bz2`

Si deve sottolineare come solo alla versione 4.6 dei tool hanno provveduto a fornire il riconoscimento dell'entry point `_start` anche se il linking della sua definizione all'interno del binario `crtbegin`, causa problemi nel riconoscimento dei simboli EABI. In questa versione è tuttavia definito il simbolo `__libc_init`, tramite il quale consentire l'inizializzazione della libreria. Si è ritenuto necessario definire tale entry point in linguaggio Assembly, descritto all'interno dell'Appendice nella Sezione B.4 a pagina 127.

Anche se mostrerò come nel caso specifico dei device utilizzati non sia necessario specificare dei flag di compilazione, questi si rivelano necessari per

l'esecuzione dei binari sull'emulatore, la cui architettura è ARMv7-a. Per dispositivi con processori Cortex-A9 o similari quale il Galaxy Nexus, è possibile specifiare in fase di compilazione il flag `-mcpu=cortex-a9 -mtune=cortex-a9`.

Per architetture ARMv5[2], possono essere utilizzati i seguenti flag [Jol09]:

```
export CFLAGS += -march=armv5te -mtune=xscale -msoft-float\
                 -fpic -ffunction-sections -funwind-tables -fstack-protector \
                 -fno-exceptions -D__ARM_ARCH_5__ -D__ARM_ARCH_5T__ \
                 -D__ARM_ARCH_5E__ -D__ARM_ARCH_5TE__  -Wno-psabi -mthumb -Os \
                 -fomit-frame-pointer -fno-strict-aliasing -finline-limit=64 \
                 -DANDROID  -Wa,--noexecstack -O2 -mfpu=vfpv3-d16 -DNDEBUG -g
```

I flag per effettuare la compilazione verso architetture ARMv4 sono invece i seguenti:

```
GLOBAL_CFLAGS += \
        -march=armv4t -mcpu=arm920t -mtune=xscale \
        -msoft-float -fpic \
        -mthumb-interwork \
        -ffunction-sections \
        -funwind-tables \
        -fstack-protector \
        -fno-short-enums \
        -D__ARM_ARCH_4__ -D__ARM_ARCH_4T__ \
        -D__ARM_ARCH_5E__ -D__ARM_ARCH_5TE__
```

4.3.1. Utilizzo di altri tool di crosscompilazione.

4.3.1.1. *Crosstool-NG.* Questo tool potrebbe essere veramente utile allo scopo di generare binari che non dipendano dalle librerie e dal linker disponibili all'interno del dispositivo Android voluto: di fatti esso richiede unicamente di configurare quali versione dei tool utilizzare, quali il compilatore, e quale versione del Kernel Linux utilizzare. Faccio inoltre notare come sia consigliabile scegliere una delle configurazioni presenti all'interno del folder `$crosstoolng-path/config`, in quanto versioni differenti dei tool da quelle riportate potrebbero causare problemi di compilazione dei binari.

Si potrebbe tuttavia utilizzare questo strumento allo scopo di compilare binari per una versione Android *off-the-shelf*, ma a questo punto si rivela necessario specificare in quale percorso sia situato il linker dinamico ed in quale le librerie di sistema.

Bisogna inoltre sottolineare come tale *crosstool* non generi automaticamente il linker da utilizzare all'interno del dispositivo, che quindi necesiterebbe anch'esso di essere cross-compilato.

[2]Su alcuni dispositivi ARMv7 può essere utilizzato l'NDK senza argomenti aggiuntivi.

4.3.1.2. *ndk-build.* Lo NDK fornisce uno script, detto `ndk-build`, il quale può automatizzare le procedure di compilazione del codice nativo, del codice Java e predisporre la compilazione delle AIDL. Questo è presente all'interno del percorso `$NDK`, e deve essere eseguito all'interno della cartella dove sono presenti i sorgenti. Un esempio di strutturazione dei sorgenti per il progetto di NDK è fornito all'interno del percorso `$NDK/samples`.

Bisogna inoltre sottolineare come questo script venga in genere utilizzato per compilare applicazioni Native con JNI e librerie di sistema, che quindi non rendono necessario l'utilizzo dell'entry point `_start` necessario alle applicazioni native scritte in C, in quanto l'esecuzione dell'applicazione inizia dal codice Java e non da quello nativo. A riprova di quanto detto, si può osservare come negli esempi forniti siano presenti esempi di compilazioni di librerie, ma non di applicazioni "puramente native": è per questo motivo che, molto probabilmente, non si è risolto completamente il problema della compilazione di quest'ultimo tipo di applicazioni.

La compilazione della singola interfaccia AIDL in linguaggio Java è possibile tramite il tool omonimo fornito dallo SDK in:

`$SDK/platform-tools`

CAPITOLO 5

Preparazione dei dispositivi Android

Indice

5.1. Premesse: riconoscimento del dispositivo Android all'interno dell'ambiente GNU/Linux

Per quanto riguarda il riconoscimento del dispositivo Android all'interno di un sistema operativo GNU/Linux, ed in particolare per il riconoscimento dello stesso da parte dell'SDK Android, è necessario eseguire il seguente script una volta aver collegato il dispositivo al computer ed aver ottenuto, tramite il comando `lsusb`, l'identificativo del venditore del dispositivo.

```
#!/bin/bash
# Giacomo Bergami (c) 2012

echo Insert the idvendor from lsusb
#In particolare per il GalaxyNexus il valore è 18d1, mentre per l'Olipad è 04f2
read idd
echo Insert the username of the system
#Inserire il nome utente del sistema Linux sul quale si vuole riconoscere il
#dispositivo
read user
echo Insert the folder where to mount the drive
#Inserire il nome di una cartella d'esempio: tablet. In questo modo il
#device verrà montato sul percorso /media/tablet
read path

apt-get install mtpfs
apt-get install mtp-tools
#apt-get install python-pymtp
```

```
echo SUBSYSTEM==\"usb\", ATTR{idVendor}==\"$idd\", MODE=\"0666\" >> /etc/udev/
    rules.d/51-android.rules
mkdir /media/$path
chown $user:$user /media/$path
echo mtpfs     /media/$path     fuse     user,noauto,allow_other      0      0
    >> /etc/fstab
echo user_allow_other >> /etc/fuse.conf
adduser $user fuse
echo The System will now reboot. Press ENTER to continue.
read
reboot
```

In particolare, le operazioni di modifica di `fuse.conf` e l'aggiunta dell'utente all'interno del sistema, sono necessarie unicamente alla prima esecuzione dello stesso. Tramite questa operazione è già possibile interagire col dispositivo tramite i tool di Android come descritto nella Sezione 4.2 a pagina 68.

In questo modo tuttavia non è presente l'applicazione `su` (o comunque essa non è utilizzabile), né è presente l'applicazione apk di Superuser per gestire i permessi di superutente nelle applicazioni che si basano su Java.

5.2. Rooting del dispositivo

Premessa: se si intende compilare una versione dell'Android Kernel è opportuno, prima di effettuare il *rooting* sul dispositivo, ottenere il file `/proc/config.gz` tramite `adb`.

Il rooting del dispositivo si rivela necessario qualora si voglia rendere possibile l'accesso ai privilegi di superutente ad un'applicazione: questo ad esempio è necessario per quelle applicazioni native che devono poter accedere a servizi forniti dal sistema operativo che richiederebbero la descrizione dei permessi all'interno di un Manifest File, per i motivi già illustrati nella Sezione 3.6 a pagina 54. Questo è inoltre provato dal tentativo di esecuzione dell'applicazione `pjsua` all'interno di una shell dove non sono stati richiesti i permessi di superutente; l'applicazione termina con l'errore che segue:

```
23:34:31.991    pjsua_aud.c  ....Opening sound device PCM@16000/1/20ms
23:34:31.992   opensl_dev.c  .....Creating OpenSL stream
23:34:31.996   opensl_dev.c  .....Recording stream type 4, SDK : 16
23:34:31.999   opensl_dev.c  .....Cannot realize recorder : 9
23:34:31.999   opensl_dev.c  .....Stopping stream
23:34:31.999   opensl_dev.c  .....OpenSL stream stopped
23:34:32.000   opensl_dev.c  .....OpenSL stream destroyed
```

In particolare il numero d'errore 9 corrisponde al valore di SLResult `SL_RESULT_CONTENT_UNSUPPORTED`. Tuttavia questo output è poco informativo se non si considera anche quello fornito dal LogCat durante l'esecuzione dell'applicazione in questione:

```
W/ServiceManager(  106): Permission failure: android.permission.RECORD_AUDIO
    from uid=10043 pid=2978
E/AudioFlinger(  106): Request requires android.permission.RECORD_AUDIO
E/AudioRecord( 2978): AudioFlinger could not create record track, status: -1
E/libOpenSLES( 2978): android_audioRecorder_realize(0x2212d0) error creating
    AudioRecord object
W/libOpenSLES( 2978): Leaving Object::Realize (SL_RESULT_CONTENT_UNSUPPORTED)
```

Possiamo infatti notare che il Service Manager non rende fruibile all'applicazione l'`AndroidRecord`, in quanto l'applicazione eseguita in modalità utente non detiene il permesso `android.permission.RECORD_AUDIO` richiesto, che tuttavia risulta essere inglobato nei permessi del superutente, in quanto viene permessa l'esecuzione di tale operazione.

Posso inoltre notare come Android consenta che la connessione ad Internet venga effettuata tramite le funzioni primitive della *libc*, anche se si ritiene che Google abbia implementato un meccanismo di Networking "paranoico" ([vedi Yag11, p. 37]) che consenta solamente a certi processi, con peculiari *capability*, di accedere a tali servizi. Ricordo inoltre che, all'interno di Applicazioni Java o di Applicazioni Native con JNI, è necessario specificare all'interno del manifest l'intenzione di accedere ai servizi di rete.

Fornisco quindi di seguito alcune procedure di Rooting che sono state effettuate allo scopo di valutare il porting di Pjproject.

5.2.1. Rooting dell'emulatore. Per poter modificare la cartella `/system` che è montata in modalità di sola lettura, è necessario effettuarne il remounting nel modo seguente:

```
  ... mount -o rw,remount -t yaffs2 /dev/block/mtdblock0 /system
```

Per installare l'applicazione, è sufficiente copiare il binario "`su`" (del quale sono presenti differenti versioni in rete) all'interno di `/system/xbin` tramite `adb`, ed installare l'apk *Superuser* tramite il comando `install` di `adb`.

Alla fine della procedura possiamo rimontare `/system` come *read-only* (ro); tuttavia possiamo mantenere abilitata la scrittura per poter effettuare delle ulteriori modifiche fino al prossimo riavvio, quando tale cartella verrà rimontata con soli permessi di lettura.

5.2.2. Rooting di Samsung Galaxy Nexus. Al contrario dell'emultatore, i dispositivi Android hanno in genere il bootloader bloccato (*locked*), che è sempre necessario sbloccare anche se, in questo modo, si infrange la garanzia. Allo scopo di effettuare l'*unlocking*, è necessario entrare con il dispositivo nella modalità *bootloader*. Per accedere in questa modalità o si invoca il comando `adb reboot bootloader` o, nel caso particolare del dispositivo Galaxy Nexus, spegnere il dispositivo, prima premere i tasti di aumento e diminuzione del volume

contemporaneamente, e poi tenere premuto anche il tasto di accensione, finché non sarà visibile il bootloader. Il comando per l'*unlocking* è il seguente:

```
fastboot oem unlock
```

È sempre possibile ripristinare il bloccaggio (locking) in un secondo momento tramite il comando:

```
fastboot oem unlock
```

Bisogna ricordare che, con il primo comando di cui sopra, si compromettono i dati contenuti all'intero di `/sdcard`, le configurazioni dell'utente ed i programmi installati. Per ovviare a questo problema, è sempre possibile eseguire il seguente comando, allo scopo di effettuare un backup del dispositivo:

```
adb backup [-f <file>] [-apk|-noapk] [-shared|-noshared] [-all] [-system|-
    nosystem] [<packages...>]
```

È sempre possibile ripristinare tale backup in qualunque momento tramite l'altro comando che segue:

```
adb restore <file>
```

Con questo comando si mantengono tutte le eventuali procedure di rooting apportate, ma non si recupereranno le modifiche effettuate sul file system del dispositivo, come la creazione di nuove cartelle o l'aggiunta di nuovi binari. A questo punto sarà possibile caricare un'immagine per poter effettuare in ogni momento la recovery; adoperando ad esempio Recovery Clockwork Touch, si può invocare il comando :

```
fastboot flash recovery recovery-clockwork-touch-x.y.z.t-maguro.img
```

dove x.y.z.t è la versione a disposizione dell'immagine in questione. È possibile comunque ottenere le immagini accedendo all'indirizzo `http://www.clockworkmod.com/rommanager`.

In seguito è sufficiente fornire il file zippato per il rooting, che dovrà essere inserito all'interno della `/sdcard`, in modo da poterlo fornire alla `recovery-utility`. Sarà sempre possibile inoltre recuperare l'immagine originaria della recovery tramite l'indirizzo:

`https://dl.google.com/dl/android/aosp/yakju-jro03c-factory-3174c1e5.tgz`

È quindi sempre possibile ripristinare il dispositivo allo "stato di fabbrica" tramite il pacchetto fornito come sopra.

5.2.3. Rooting di Olivetti Olipad. La procedura di Rooting dell'Olivetti Olipad è non standard, in quanto consente immediatamente di vedere quali sono le modifiche da effettuare al Ramdisk per consentire la modifica del filesystem `/system/xbin`; altro fattore non standard è l'utilizzo del tool `nvflash`, il quale è utilizzato per effettuare il flashing di dispositivi con processore Tegra 2.

In quanto questa procedura non è di particolare interesse ai fini dell'illustrazione dei tool messi a disposizione da Android, il codice sorgente dello script è messo a disposizione nella Sezione B.5 a pagina 129.

CAPITOLO 6

Pjproject

Tue Aug 14 04:07:02 EDT 2012.

I just want to say for everyone, that the Android setup that we support is what is in https://trac.pjsip.org/repos/wiki/Getting-Started/Android. Even this setup is experimental as it's currently under development, so expect some problems.

If you decide to use different setup, such as to use the trunk, or to use pjsua, then YOU ARE ON YOUR OWN. That means you know what you're doing and you know how to fix problems yourself. Please don't report any errors or crashes here as that will only confuse people.

Best regards, Benny

Benny Prijono, Pjproject main developer.
`http://lists.pjsip.org/pipermail/pjsip_lists.pjsip.org/2012-August/015136.html`

Indice

6.1. Pjproject: descrizione

Pjproject* è una libreria scritta in linguaggio C adoperata allo scopo di implementare il protocollo di comunicazione SIP. La struttura di questo progetto è inoltre mirata a garantire un'estrema configurabilità ed indipendenza dal sistema operativo nel quale tale libreria verrà utilizzata. Allo scopo viene predisposto un primo strato, definito dalla sottolibreria `pj`, nel quale vengono fornite le implementazione delle funzioni basilari che verranno utilizzate, più o meno direttamente, da tutti gli altri strati rappresentati dalle sottolibrerie fornite dal progetto di Pjproject. Oltre alla libreria `pjmedia` di cui parlerò per la gestione dei file(s) multimediali, utilizzerò nella seguente trattazione la libreria `pjsua`, che implementa un livello di astrazione, allo scopo di realizzare applicazioni utente di alto livello. Questa consente di [Pri06]:

- Effettuare la registrazione di utenti e loro gestione.
- Supportare chiamate concorrenti.
- Gestione dello stato delle chiamate.
- Supporto wideband delle informazioni multimediali, supporto di codec audio e di riproduzione di file multimediali.

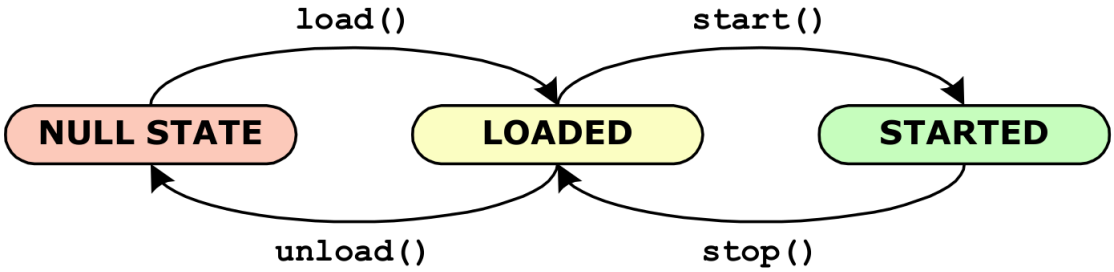


FIGURA 6.1.1. *Diagramma degli stati di un modulo.*[Pri06]

Questa libreria è inoltre strutturata sulla gestione di moduli, i quali hanno lo scopo di distribuire i messaggi SIP tra i vari strati, e di implementare un'interfaccia di astrazione. Questi moduli, come illustrato in Figura 6.1.1, sono caratterizzati da stati che mostrano lo stato di attività degli stessi. Questi sono gestiti da un `pjsip_endpoint`, il quale si preoccupa di far transitare i messaggi tra i moduli o di attivarne i callback in base al loro grado di priorità, definito in base alla loro posizione all'interno dello stack dei moduli e delle corrispondenti librerie.

Descriverò ora quale sia la struttura di un device audio all'interno di Pjproject: faccio riferimento allo stesso sorgente di `pjsua`, ottenibile all'interno

*Chiamerò in questa trattazione il progetto PJSIP (`http://www.pjsip.org/`) come Pjproject (come tra l'altro indicato dal nome del *trunk* del sorgente (`http://svn.pjsip.org/repos/`), per non confonderlo confonderlo con la stessa libreria `pjsip` presente all'interno dello stesso progetto.

del sorgente del *branch* per Android[1]. In quanto non è disponibile una spiegazione maggiormente dettagliata del codice sorgente stesso, per spiegare come avvenga questo meccanismo, debbo far riferimento al codice presente in

`$PJA/pjsip-apps/src/pjsua`

dove con `$PJA` indico il percorso dove sono situati i sorgenti di tale *branch*.

Ogni modulo della libreria fornisce un accesso compatto alle funzioni definite all'interno del file sorgente: questo consente di fornire un'unica interfaccia a tutti i moduli di una stessa tipologia, che possono quindi essere implementati differentemente in base ai singoli obiettivi di realizzazione. Questo discorso è estendibile a tutti i device implementati all'interno del percorso:

`$PJA/pjmedia/src/pjmedia-audiodev`

Soffermandomi in particolare sul file `opensl_dev.c` dov'è presente l'implementazione dell'interazione con la scheda audio tramite libreria `wilhelm`, posso notare come sia predisposta una factory allo scopo di fornire una pool per l'allocazione dei dati, il riferimento all'*engine* OpenSL ES ed una serie di funzioni per gestire lo stream di dati.

Quando farò riferimento alla versione *trunk*, mi riferirò alla versione 2.0, ottenibile al seguente indirizzo: `http://www.pjsip.org/release/2.0/pjproject-2.0.tar.bz2`

6.2. Premesse al porting di Pjproject

6.2.1. Definizione di `user.mak`. Questo è il file preposto dagli sviluppatori di Pjproject allo scopo di definire dei flag necessari allo sviluppatore. Ne forniamo di seguito l'implementazione:

Listato 6.1. user.mak

```
export INSANITY_OBJECT = /path/to/crt0.o
export CC_LDFLAGS += $(INSANITY_OBJECT)
export APP_LDLIBS += $(INSANITY_OBJECT)
export PJ_LDLIBS += $(INSANITY_OBJECT)
export PJ_CFLAGS += $(INSANITY_OBJECT)
export _CFLAGS += $(INSANITY_OBJECT)
```

In particolare si è ritenuto necessario specificare l'oggetto `/path/to/crt0.o`, dove il percorso indicato è quello del binario contenuto all'interno dell'*host* per

[1]Nota: questo branch è ritenuto, al momento di redazione della tesi, ancora instabile. Tuttavia non sono stati in questo caso risolti i problemi posti in precedenza, tra cui i flag di compilazione. Sorgente: `http://svn.pjsip.org/repos/pjproject/branches/projects/android/`.

il crosscompiling, in quanto si riscontrano problemi che possono insorgere dalla sua mancata definizione e quindi dalla mancata invocazione dell'inizializzazione della libreria *Bionic*. In particolare in fase di compilazione è possibile osservare il seguente avvertimento:

```
: warning: cannot find entry symbol _start; defaulting to 00008280
```

In mancanza della definizione di `_start` verrà utilizzata, come funzione sostituta, una presente nel programma che si sta crosscompilando. Nel caso di `pjsua` la prima funzione coincideva con quella di main e quindi, in completa assenza di tale binario, il programma partiva (apparentemente) correttamente, ma visualizzando il seguente output.

```
21:37:35.213 os_core_unix.c !pjlib 2.0 for POSIX initialized
21:37:35.229 sip_endpoint.c  .Creating endpoint instance...
21:37:35.233          pjlib  .select() I/O Queue created (0x168c7c)
21:37:35.236 sip_endpoint.c  .Module "mod-msg-print" registered
21:37:35.236 sip_transport.  .Transport manager created.
21:37:35.236   pjsua_core.c  .PJSUA state changed: NULL --> CREATED
Segmentation Fault
```

In particolare il LogCat evidenzia l'evoluzione delle chiamate a funzione tramite il tracciamento dei valori assunti dallo stackpointer:

```
I/DEBUG   (   96):          #00  pc 00093ea8  /data/local/bin/pjsua (
    _getopt_internal.constprop.1)
I/DEBUG   (   96):          #01  pc 0001282c  /data/local/bin/pjsua (app_init)
I/DEBUG   (   96):          #02  pc 0000bbbc  /data/local/bin/pjsua (main_func)
I/DEBUG   (   96):          #03  pc 000b4b5c  /data/local/bin/pjsua (pj_run_app
    )
```

Nella prima funzione avviene il parsing dei parametri, che termina con un *Segmentation Fault* in quanto si tenta di raggiungere un argomento che ha valore `NULL`. La definizione della funzione `_start` e la conseguente chiamata all'inizializzazione di suddetta libreria porta al superamento dell'ostacolo sopra descritto.

6.2.2. Definizione di `configure-android` e `myConf`. Questo primo script è stato proposto all'interno del branch Android di Pjproject, che riporto all'interno del Listato B.3 a pagina 121. In particolare utilizzo la versione 4.6 della toolchain di cross-compilazione, e specifico in `LDFLAGS` il parametro `-nostdlib` per forzare l'utilizzo delle librerie incluse nel percorso indicato dal flag `-L`.

Lo script precedente viene lanciato dal seguente:

Listato 6.3. myConf

```
#!/bin/bash
export ANDROID_NDK=/home/jack/android-ndk-r8b
export API_LEVEL=14
export PATH=$PATH:$ANDROID_NDK
```

Listato 6.2configure-android

```
#!/bin/sh
#...Omissis
TARGET_HOST="arm-linux-androideabi"
TC_DIR=${TARGET_HOST}
BUILD_MACHINE="linux-x86"
#...Omissis

if test "x$API_LEVEL" = "x"; then
  API_LEVEL=‘ls ${ANDROID_NDK}/platforms/ | sed ’s/android-//’ | sort -gr |
      head -1‘
  echo "$F: API_LEVEL not specified, using android-${API_LEVEL}"
fi

ANDROID_TC="${ANDROID_NDK}/toolchains/${TC_DIR}-4.6/prebuilt/${BUILD_MACHINE}"
if test ! -d ${ANDROID_TC}; then
  echo "$F error: unable to find directory ${ANDROID_TC} in Android NDK"
  exit 1
fi

export ANDROID_SYSROOT="${ANDROID_NDK}/platforms/android-${API_LEVEL}/arch-arm"
if test ! -d ${ANDROID_SYSROOT}; then
  echo "$F error: unable to find sysroot dir ${ANDROID_SYSROOT} in Android NDK"
  exit 1
fi

export CC="${ANDROID_TC}/bin/${TARGET_HOST}-gcc"
export CXX="${ANDROID_TC}/bin/${TARGET_HOST}-g++"

export LDFLAGS=" -nostdlib -L${ANDROID_SYSROOT}/usr/lib/"
export LIBS=" -lc -lgcc -lm"
export CFLAGS=" -Wl,-Bstatic -lm -I${ANDROID_SYSROOT}/usr/include"
export CPPFLAGS="${CFLAGS}"
export CXXFLAGS=" -Wl,-Bstatic -lm --sysroot=${ANDROID_SYSROOT}"

#Omissis

./configure --host=${TARGET_HOST} --disable-video
```

```
#Eventuale riconfigurazione degli script
cp -av /usr/share/misc/config.guess ./
cp -av /usr/share/misc/config.sub  ./

autoconf aconfigure.ac > aconfigure
chmod +x aconfigure
```

```
chmod +x configure

echo "#define PJ_CONFIG_ANDROID 1
#include <pj/config_site_sample.h>" > pjlib/include/pj/config_site.h

./configure-android --disable-floating-point --disable-large-filter $*
```

in questo caso definisco le variabili d'ambiente per il percorso base dell'NDK, il numero dell'API Android per la quale effettuare la compilazione, la riconfigurazione dell'`aconfigure.ac` in seguito al patching, la definizione del file

`pjlib/include/pj/config_site.h`

per la definizione personalizzata dei flag di inclusione, e la disabilitazione del linking dei moduli di pjsua per i codec non supportati da Android.

In particolare posso effettuare le seguenti osservazioni:

- Oltre all'inclusione della libreria `-lc`, è necessario includere anche `-lgcc` in quanto altrimenti non si troverà `_start`
- Per effettuare il linking tra codice C e C++ è necessario specificare l'opzione `-fno-exceptions`, in quanto le eccezioni non si devono propagare da C++ a C.
- Per poter effettuare il linking con la libreria audio di Android, si specifica l'opzione `-lOpenSLES`.

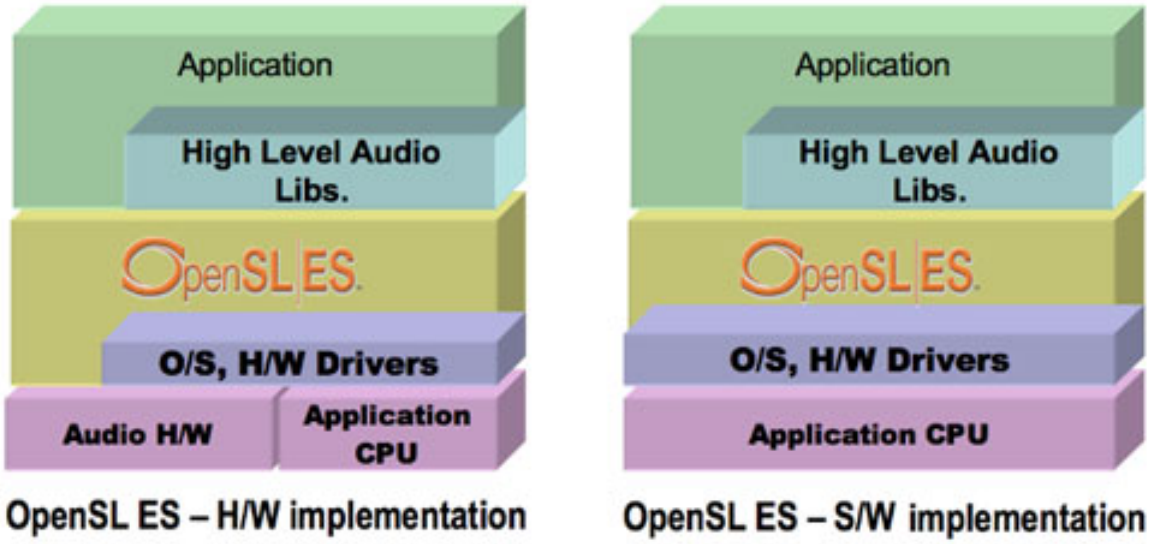
6.2.3. Pjproject e i driver audio: OpenSL-ES. *Per la descrizione della particolare implementazione delle API OpenSL ES da parte di Android detta `wilhelm`, si veda la Sottosezione 3.6.2 a pagina 59.*

OpenSL-ES è, come dicono gli stessi sviluppatori [Inc09], un'API realizzata per sistemi *embedded* e *mobili* con il supporto per la multimedialità, che fornisce un'interfaccia audio indipendente dal device ed il più possibile *cross-platform*. È inoltre sviluppata tenendo conto delle ristrette risorse disponibili di questi dispositivi ed utilizzando API *object-oriented*, che fanno assomigliare il codice C a quello di linguaggi quale il C++, con un approccio simile a quello già descritto in Pjproject.

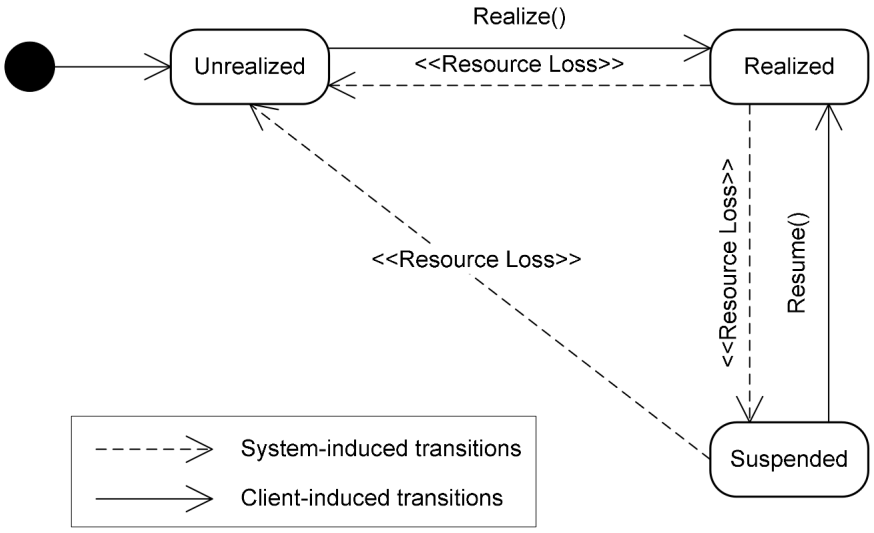
Posso mostrare brevemente quale sia l'interfaccia utilizzata per interagire con gli oggetti descrivendo l'interfaccia base per tutti gli oggetti, ovvero quella fornita da `SLObjectItf`.

Come possiamo notare dalla Figura (b) 6.2.1 nella pagina successiva, all'atto della creazione dell'oggetto questo è *unrealized*: ciò vuol dire che le sue risorse non sono state ancora allocate e non è inoltre possibile utilizzare i metodi esposti dalle sue interfacce. Per effettuare la transizione allo stato *realized*, è necessario invocare l'omonimo metodo (`Realize`) sull'oggetto in questione.

Tuttavia l'interazione con i suddetti oggetti è possibile unicamente previa istanziazione dell'*Engine Object*, tramite la funzione globale `slCreateObject()`,



(A) *Posizione della libreria OpenSL ES all'interno di un sistema operativo: l'implementazione Wilhelm è del secondo tipo, in quanto interagisce direttamente con le librerie di sistema.* `http://www.khronos.org/opensles/`



(B) *Diagramma degli oggetti mostrante l'interazione tra oggetti nei vari livelli del codice. [Inc09]*

FIGURA 6.2.1. OpenSL ES.

la quale fornirà appunto un'istanza del sistema, per mezzo della quale è possibile ottenere nuove istanze di oggetti, le cui interfacce sono già descritte all'interno del sistema. Un utilizzo dell'implementazione di tale API è fornita

all'interno del file `opensl_dev.c` della libreria Pjproject nel branch Android, che ho quindi deciso di adottare in quanto questa versione aveva nel frattempo implementato i device per la gestione dell'audio.

In prima analisi, noto come `CreateAudioInterface` sia un puntatore a funzione della `struct SLEngineItf_` definita all'interno di *OpenSLES.h*, che è a sua volta il tipo della variabile `engineEngine` all'interno della definizione del driver audio nel file `opensl_dev.c`. In pratica posso osservare come l'implementazione di `src/itf/IEngine.c` associ a tale puntatore la funzione `IEngine_CreateAudioPlayer`.

6.2.3.1. *Patch per il device audio (`android_sles_dev.c`) e branch Android.* Utilizzando la versione *trunk* di `pjsip` e crosscompilandola con la patch trovata all'interno della mailing-list di Android, all'atto dell'instaurazione della chiamata, riscontro il seguente errore:

```
23:10:04.169   pjsua_call.c !Making call with acc #1 to sip:192.168.0.9
23:10:04.169    pjsua_aud.c  .Set sound device: capture=-1, playback=-2
23:10:04.169    pjsua_aud.c  ..Error retrieving default audio device parameters
    : Unable to find default audio device (PJMEDIA_EAUD_NODEFDEV) [status
    =420006]
```

A causa di questo problema, ho optato per l'utilizzo della versione *branch* per Android, anche perché questa versione è stata corredata per i codec wav, che sembrano mancanti nella versione *trunk*, ed è da ritenersi maggiormente completa. Di fatti a quanto pare il patch indicato non sembra effettuare correttamente l'ottenimento dei parametri di default, e nemmeno è in grado di ottenere il device audio predefinito.

6.3. Gestione dei file WAVE

La gestione dei file audio è affidata alla definizione di moduli detti *porte*, allo scopo di creare un livello di astrazione sulle informazioni multimediali contenute al livello di File System. In questo modo le informazioni multimediali ottenute possono essere agevolmente processate verso il *conference bridge* che veicola le informazioni multimediali in entrata od in uscita alla chiamata VoIP che si verrà ad instaurare. Ogni porta contiene inoltre le informazioni dello stream multimediale in entrata ed in uscita dalla stessa.

Guardando ora alla funzione `app_init` definita all'interno del file `pjsua_app.c`, dalla quale si inizia l'esecuzione effettiva dell'applicazione, otteniamo come venga predisposta la *conference*:

```
app_init() [pjsua_app.c]
↳ pjsua_init() [pjsua_core.c]
  ↳ pjsua_media_subsys_init() [pjsua_media.c]
    ↳ pjsua_aud_subsys_init [pjsua_aud.c]
      ↳ pjmedia_conf_create() [conference.c]
```

Questa procedura tuttavia si differenzia dall'inizializzazione della porta della conferenza, che avviene all'apertura del file:

```
app_init() [pjsua_app.c]
↳ pjsua_player_create() [pjsua_aud.c]
  ↳ pjmedia_conf_add_port() [conference.c]
    ↳ create_conf_port()
```

In questo punto avviene l'effettiva inizializzazione della porta di conferenza, grazie alle informazioni ottenibili dalla porta audio precedentemente aperta sul file audio, e l'inizializzazione dei buffer per la ricezione dei dati dalla porta di conferenza, o in uscita verso la stessa. Tuttavia nemmeno qui si ottengono le informazioni sulle caratteristiche del file audio.

L'inizializzazione delle informazioni dalla porta audio avviene nella funzione `pjmedia_wav_player_port_create`, richiamata direttamente dalla funzione `pjsua_player_create()`, prima dell'aggiunta della porta ottenuta alla porta di conferenza in `pjmedia_conf_add_port()`. All'interno della funzione suddetta, dopo aver aperto una porta sul file audio ed aver acceduto al file in lettura, si ottengono le informazioni dallo header del file WAVE, che vengono utilizzati nella funzione `pjmedia_port_info_init` per impostare i valori di default della porta.

6.3.1. Approfondimento: struttura di un file WAVE. Presento qui un'introduzione sulla struttura di un file WAVE (contrazione di *WAVEform audio file format*): tale formato audio è costituito principalmente da vari *chunk* caratterizzati da uno *header* iniziale, e da dati che seguono. La struttura standard di un file WAVE è presentata in Figura 6.3.1 nella pagina seguente, dalla quale si può evincere come principalmente possano esistere 3 *chunk* chiamati *RIFF*, *format* e *data*:

RIFF: Questo primo *chunk* contiene uno header che contraddistingue tale file come WAVE: in particolare il primo ed il terzo campo assumono il valore rispettivamente di `RIFF` e `WAVE`, mentre il secondo indica la dimensione delle informazioni inglobate dopo il terzo campo.

Format: Questo secondo *chunk* e accomunato a tutti gli altri, escluso il precedente, dai campi `ChunkID` e `ChunkDataSize`. I campi rimanenti caratterizzano il tipo di compressione (es. `PCM` - ovvero non compresso, *GSM*, *MPEG*), i canali audio utilizzati, il numero di campionamenti al secondo (valore indipendente dal numero dei canali), quanti *bytes* devono essere riprodotti al secondo per riprodurre l'audio contenuto ($ByteRate = SampleRate \cdot BlockAlign$), la dimensione di ciascun campionamento ($BlockAlign = \frac{bpsample}{8} \cdot NumChannels$), il numero dei *byte* utilizzato per definire ciascun campionamento e il numero di *byte* addizionali che segue.

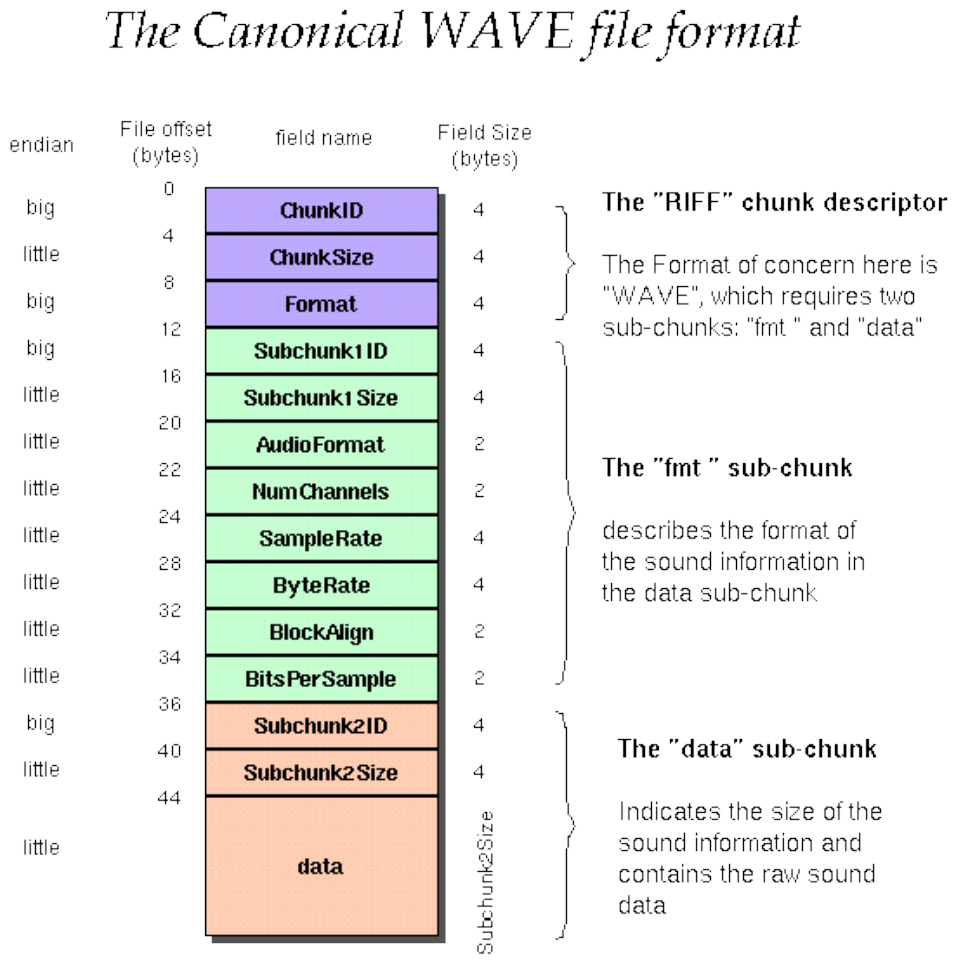


FIGURA 6.3.1. Struttura di un file WAVE. `http://math2033.uark.edu/wiki/images/4/45/Wav-sound-format.gif`

Data: Questo terzo *chunk* contiene le informazioni digitalizzate dell'audio.

Possiamo osservare un esempio di come ottenere queste informazioni tramite il file fornito nella Sottosezione C.1.1 a pagina 133, che è stato utile ai fini di individuare, all'interno del codice di *pjmedia*, l'ottenimento delle informazioni dal file WAVE.

CAPITOLO 7

Tentativi di porting e considerazioni effettuate

Indice

Esporrò di seguito per punti le procedure effettuate allo scopo di realizzare il porting dell'applicazione, ed a quali conclusioni sia giunto ogni volta. Elenco di seguito ulteriori considerazioni effettuate, non descritte all'interno di questo capitolo ma presenti all'interno della Tesi:

- ◇ Interazione tra Emulatore e Macchina Ospite: v. Sottosezione 4.2.1 a pagina 68.
- ◇ Tentativi di configurazione per la crosscompilazione: v. Sezione B.2 a pagina 118.
- ◇ Analisi del sorgente e modifica nel supporto dei file WAVE: v. Sezione 7.2 a pagina 94.
- ◇ Modifica nel sorgente dell'AOSP Source 7.3 a pagina 98.

7.1. Considerazioni sulla crosscompilazione

Una breve descrizione dei tentativi effettuati per la crosscompilazione è fornita nell'Appendice alla Sezione B.2 a pagina 118, tramite il particolareggiamento sul cambiamento delle configurazioni.

Proseguo ora nella descrizione dei tentativi di porting:

(1) Come prima cosa, ho provato a compilare `pjsua` specificando unicamente in quale percorso trovare il binario di cross-compilazione, effettuando l'export dell'unica variabile CC.

Proprio a questo credo sia dovuto il primo errore di compilazione: di fatti lo stack del `pc` deducibile dal `logcat` forniva come ultimi binari `/system/bin/linker` e `/system/lib/libc.so`.

Questo mostrava che la procedura di linking dinamica funzionava a dovere, che effettivamente il linker aveva utilizzato la libreria richiesta, ma l'esecuzione di `__set_errno` era errata. Con il senno di poi, e conoscendo l'istruzione `nm` per andare ad indagare i simboli contenuti all'interno dei file binari, ho visto che `__set_syscall_errno` era posizionata proprio all'inizio del file, e con ogni probabilità è stata scelta come funzione *entry point*. Non era stato quindi compilato correttamente per l'architettura ARM.

(2) In seguito ho riscontrato lo stesso errore anche compilando la libreria *pjlib* nel branch di Android, in quanto (con ogni probabilità) credevo che il problema consistesse nel sorgente, e non nel metodo da me adottato per effettuare cross compiling.

(3) Poi ho trovato i flag di compilazione per architettura ARMv5, ed ho provato ad applicare quelli allo scopo di effettuare cross compilation. Ho provveduto quindi a modificare direttamente i flag di linking e compilazione in `configure-android`, senza ancora utilizzare il file .mak

Con ogni probabilità anche l'esecuzione della syscall con i seguenti parametri:

```
sigaction(48472, 0xb00144c4, [], SA_RESTART, 0xb00144c4, [],
                 SA_RESTART, 0) = -1 EINVAL
```

(che portava ad un SIGFAULT in quanto non era definito da nessuna parte un segnale con codice 48472) indicava un errato modo di effettuare cross-compilazione, o per i flag che indicano le caratteristiche dell'EABI del processore, o ancora per una non corretta endianess riconosciuta in fase di configurazione.

(4) Osservando che queste syscall non vengono eseguite correttamente, suppongo che questo sia dovuto ad una carenza di permessi dell'utente che esegue le istruzioni.

Come ho avuto modo di dimostrare precedentemente, ritengo col senno di poi che questo fosse imputabile ad una non corretta cross-compilazione, anche se il problema dei permessi sarebbe comunque prima o poi emerso.

Ho osservato, leggendo la documentazione di Google, che ogni applicazione viene eseguita all'interno di una "sandbox", per uscire parzialmente dalla quale è necessario esibire al sistema la richiesta di speciali permessi di esecuzione. Avendo prima d'allora effettuato programmazione Android unicamente in Java, ho fornito tali permessi unicamente tramite un file, detto "Manifest", all'interno del quale esplicitavo la necessità di connettermi alla rete Internet, piuttosto di scrivere all'interno del filesystem. Da quanto mi risultava inoltre da procedure di rooting, con i permessi di superutente si sarebbe potuto uscire dalla

sandbox. Questo è peraltro confermato da informazioni che ho acquisito successivamente e che mostra come il controllo dei permessi avvenga lato *service* Java tramite l'utilizzo delle librerie fornite dal sistema.

(5) In quanto ritengo "unsafe" in termini di garanzia del dispositivo reale effettuare rooting sul mio primo dispositivo (ovvero l'Olivetti Olitab, che tra l'altro è scarsamente supportato in ambito cracking rispetto ad altri più mainstream), effettuo una prima procedura di rooting sull'emulatore; questa è già stata discussa nella Sottosezione 5.2.1 a pagina 75.

A riprova del fatto che la causa del mancato funzionamento del binario non era da addebitare al mancato rooting del dispositivo, neanche in questo modo nell'emulatore si risolve l'`EINVAL` di cui sopra. A questo punto, imputando il problema ad una limitazione effettiva di tale dispositivo (cosa plausibile visti i miei trascorsi con la programmazione Java), mi decido ad effettuare la procedura di rooting anche sul mio dispositivo personale.

Ad ulteriore riprova di ciò che sto più volte ribadendo, anche utilizzando il tablet l'applicazione non sembra partire. Inoltre, non essendo implementato in questo dispositivo il binario `strace`[1], ciò rende difficoltosa l'interpretazione dei motivi dell'errore.

Effettuo quindi la comparazione tra gli output del LogCat (più verboso ma meno informativo della `strace`) dell'emulatore e quello del dispositivo personale in questione. Noto solo così per la prima volta la discrepanza tra i due output. Inizio a pensare per la prima volta che l'errore di esecuzione sia dovuto ad una disparità tra la costruzione architetturale ed i flag di compilazione.

(6) Riguardando in una fase successiva la definizione di `SIGILL`, noto che questo segnale è effettivamente lanciato al processo che tenta di eseguire un'istruzione malformata, sconosciuta o che richiede particolari privilegi.

Esclusa l'ipotesi della necessità di acquisire particolari permessi, in quanto abbiamo supposto sin dall'inizio che la procedura di rooting garantisca l'accesso al superuser e quindi potenzialmente a tutte le istruzioni fornite dal kernel (o almeno alla maggior parte), è evidente che l'istruzione debba essere malformata, e conseguentemente sconosciuta, proprio a causa di una differente tipologia tra hardware reale e hardware di compilazione. L'errore non era localizzabile nella libreria in sé, in quanto caricata dinamicamente, ma nell'istruzione richiesta dal binario da me compilato.

[1] Provai ad utilizzare un binario trovato in Rete: tuttavia ancora adesso, su qualsiasi binario tracciato (compreso l'`ls` effettivamente disponibile all'interno del dispositivo personale in questione). effettua unicamente il tracciamento della syscall `exec*`.

A questo punto arrivo a concludere che il problema possa essere insito nei flag di crosscompilazione: da ciò segue che le direttive al compilatore fornite dal branch di Android per effettuare la crosscompilazione, al quale mi appoggiavo, non erano sufficienti al fine di effettuare una corretta crosscompilazione. Di fatti, dispositivi differenti dovrebbero necessitare di differenti configurazioni in fase di compilazione.

(7) In riferimento al tracciamento dei processi figli, che sembrano non essere tracciabili da strace in Android, ritengo opportuno spostare la fase di testing su applicazioni che non prevedano inizializzazioni o creazioni di thread come *pjsua*, almeno per una fase iniziale, visto l'esiguo numero di informazioni che sono ottenibili dal LogCat (Non viene di fatti indicato quale syscall è stata chiamata, ma solo l'avanzare del *pc* nel corso dell'esecuzione, di per se non molto significativo, se non a patto di ricostruirlo analizzando il binario generato). Proseguo quindi il testing su `test-pjsip`.

Documentandomi meglio, tendo poi ad escludere che Android non permetta la creazione di processi figli, in quanto la libreria *pthread* è nativamente supportata all'interno di *Bionic*.

(8) Riporto le osservazioni che avevo effettuato sull'esecuzione di `test-pjsip`: l'esecuzione dell'istruzione `usage` al posto di `main` era proprio da imputare alla mancanza dell'entry-point `_start`. Non conscio del fatto che questo fosse già implementato all'interno di `crtbegin`, in quanto convinto che questo dovesse comunque essere implementato all'interno della `libc`, decido di utilizzare l'implementazione Assembly di tale entry point, documentandomi su quale fosse la procedura di inizializzazione di libc, e provando ad effettuare il disassembly di alcuni binari di prova per Android. Trovo poi conferma di questa procedura documentandomi in rete, ed appoggiandomi a quel risultato maggiormente raffinato rispetto a quello prodotto da me inizialmente.

Solo in seguito alla redazione di una prima bozza di Tesi ed alla visione dei sorgenti di *Bionic*, scopro che tale implementazione era contenuta all'intero di `crtbegin`, unicamente tramite la somiglianza tra il sorgente da me prodotto, e quello fornito da Google. In seguito questa supposizione trae conferma dall'utilizzo di `nm` sull'oggetto binario in questione.

(9) Avevo precedentemente imputato l'ulteriore interruzione dell'esecuzione di `pjsua` all'errata esecuzione della syscall uname: tuttavia, a riprova che questa è effettivamente disponibile, il file da me scritto in C e crosscompilato allo scopo di ottenere la versione correntemente utilizzata nel mio dispositivo, ha ribadito (come tra l'altro evidenziato da un'analisi più accorta dell'output della strace) come questa sia effettivamente accessibile.

Come evidenziato, il problema dovrebbe essere costituito dall'accesso all'indirizzo NULL di memoria (Compare di fatti da Logcat un "Segmentation Fault at 0x0"). Ciò tuttavia mi lascia alquanto perplesso, in quanto pjsua su adm64 non ha mai causato quel genere di problemi, ma veniva eseguito correttamente. Da qui nasce appunto l'esigenza di proseguire il testing su di un sorgente meno "intricato" dal punto di vista del debugging.

(10) Come tra l'altro appena ribadito, il mio dubbio risiede nel fatto che, una volta liberato dai vari permessi, l'applicazione sembra ancora non proseguire correttamente nell'esecuzione. In particolare, dopo l'ultima stampa a video, noto un palese delay tra questa e la interruzione dell'operazione.

(11) Mi riservo un'ultima considerazione di mero carattere speculativo in merito all'architettura Android, e non correlata con il sorgente specifico se non per lo scopo che mi prefiggo, ovvero quello di effettuare il porting dell'applicazione per Android, circa la lettura di "Embedded Android" di Karim Yaghmour.

L'autore tuttavia sostiene che le applicazioni "native" non dovrebbero essere soggette a restrizioni, anche se poi asserisce che "tutto ciò che avviene di interessante è gestito da 50 servizi (tra cui cito `netstat` e `permission`) di sistema, dove l'interazione è gestita da un Binder (`/dev/binder`).

Inizialmente ritengo che siano necessarie tecniche di "hookup"[2] per evitare che le *system call* vengano intercettate dal sistema operativo: tuttavia a questo punto non ero ancora consapevole dei meccanismi di IPC che collegano tramite *upsyscall* i binari ai servizi di sistema, in quanto confinavo il Binder ad un ambito strettamente legato alla DVM ed a Java.

(12) Seguono i tentativi di configurazione degli script di compilazione descritti nella Sezione B.2 a pagina 118.

(13) Dopo un'attenta analisi dei warning in fase di compilazione, mi accorgo di come rimangano ulteriori messaggi segnalanti la mancata definizione di `_start`. Ciò era dovuto all'indicazione del binario `crt0.o` unicamente all'interno della variabile `PJ_LDLIBS`, che credevo contenuta all'interno di tutte le variabili per il linking. In seguito alle configurazioni definitive già mostrate nel Capitolo 6 a pagina 79, ottengo per la prima volta l'esecuzione completa di `pjsua`.

[2] I meccanismi più diffusi tramite i quali si effettua lo hacking del dispositivo sono il "Privilege Escalation Attack" e il cosiddetto "hooking", anche se la procedura di rooting varia in base alla versione di Android ed alle caratteristiche del dispositivo.

7.2. Considerazioni sulla riproduzione dei file WAVE

(1) Come descritto nella Sottosezione 7.4.1 a pagina 101, era impossibile da emulatore controllare l'effettiva riproduzione dei files, in quanto questo non supporta l'emulazione delle periferiche audio. Dopo aver notato, prima sull'Olipad e poi anche sul Galaxy Nexus, che l'esecuzione contemporanea di due istanze di `pjsua` provoca dei problemi nell'esecuzione del programma, in questa prima fase mi concentro sul testing dei due dispositivi, utilizzandone uno per la riproduzione dell'audio, e l'altro per la trasmissione dello stesso all'interno del canale di comunicazione.

Questi due dispositivi potranno interagire grazie all'utilizzo di un *router* domestico, che fornirà loro gli indirizzi IP per poter comunicare all'interno della rete locale.

Per l'analisi del problema relativo all'esecuzione contemporanea di due istanza di `pjsua`, si veda la Sezione 7.3 a pagina 98.

(2) Prima di effettuare l'analisi approfondita dell'errore, provo ad attivare o disattivare alcuni codec in fase di compilazione. Nel caso specifico, come primo tentativo aggiungo i seguenti:

```
--enable-ext-sound --disable-speex-codec --disable-speex-aec
```

Effettuo questa scelta in quanto l'applicazione mostra, con la configurazione di default, di utilizzare il codec `speex`:

```
>15:24:37.407  pjsua_media.c  ......Audio updated, stream #0: speex (
    sendrecv)
```

Una volta ricompilato il sorgente, provo ad eseguire un'istanza di `pjsua` come registratore sul Tablet Olipad, mentre scelgo di far riprodurre la traccia audio all'interno del Galaxy Nexus. In questo caso riscontro quindi un problema con la deallocazione del codec GSM, ed in particolare l'applicazione termina lato riproduttore con un Segmentation Fault. In particolare il LogCat palesa l'errore all'interno della funzione `gsm_dealloc_codec`.

In particolare noto che, dopo una prima esecuzione del programma con errore, facendolo ripartire in un secondo momento sulla stessa porta, si riscontra un errore all'atto del *binding* della porta, come mostrato dallo stesso output dell'applicazione:

```
22:57:24.663   pjsua_core.c  bind() error: Address already in use [
    status=120098]
```

Per evitare l'errore nella gestione di `gsm_dealloc_codec`, provo a disabilitare anche quella libreria in fase di configurazione con `-disable-gsm-codec`. Provando quindi ad eseguire nuovamente il binario sui due dispositivi, come già illustrato sopra: ottengo ora ancora una volta nel dispositivo

al quale è preposta la registrazione l'errore causato dall'asserzione già osservato, ovvero:

```
assertion "cport->rx_buf_count <= cport->rx_buf_cap" failed: file "../
    src/pjmedia/conference.c", line 1498, function "read_port"
```

Ciò evidenzia come il problema sia indipendente dal codec utilizzato, in quanto è comune sia alla prima configurazione e sia all'ultima.

(3) Proseguendo ora nell'analisi dell'errore, effettuo i seguenti controlli nel codice sorgente:
- Controllo che effettivamente si ottengano le informazioni corrette dal file: questo è verificato dal fatto che si ottengono le informazioni dallo header del file WAVE.
- Controllo se esista un formato audio riproducibile e che non causi errori. Mentre utilizzando il file `http://www.nch.com.au/acm/11k16bitpcm.wav` si continua a verificare il problema di asserzione, così come tutti gli altri files ottenibili dal sito, non riscontro alcun problema con il file d'esempio:

  `$PJA/tests/pjsua/wavs/input.8.wav`

  Adduco a motivazione il fatto che, il file correntemente aperto, abbia le stesse caratteristiche del bridge di comunicazione, e che per questo si ottenga il bypass del controllo di asserzione del controllo di cui sopra, in quanto in quel caso i dati vengono scritti direttamente all'interno della porta del bridge.

(4) Per verificare questa mia ultima supposizione, aggiungo delle stampe di controllo all'interno della funzione `read_port`: da queste scopro che anche nel caso del file ottenibili dai test del sorgente si entra nella gestione dei files con configurazione differente dal bridge di comunicazione.

Posso inoltre controllare come, in questo caso, avvenga effettivamente la conversione stereo/mono o mono/stereo ed il resampling audio in base al differente sample rate, copiando poi le informazioni all'interno del buffer RX per l'uscita delle informazioni verso il bridge di comunicazione. In particolare l'output dell'esecuzione di `pjsua` file d'esempio corrente è il seguente:

```
21:21:49.299   conference.c  bufcount = 160, bufcap = 160, tmpsize=320,
    spf=160
21:21:49.308   conference.c  WARNING: EXCEEDING. bufcount = 0, bufcap =
    160, tmpsize=320, spf=160
21:21:49.308   conference.c  bufcount = 160, bufcap = 160, tmpsize=320,
    spf=160
```

Mentre l'esecuzione di un file con caratteristiche audio differenti è il seguente:

```
21:19:09.101   conference.c !WARNING: EXCEEDING. bufcount = 0, bufcap =
    429, tmpsize=438, spf=219
21:19:09.102   conference.c  bufcount = 219, bufcap = 429, tmpsize=438,
    spf=219
21:19:09.102   conference.c  WARNING: EXCEEDING. bufcount = 219, bufcap
    = 429, tmpsize=438, spf=219
21:19:09.102   conference.c  bufcount = 438, bufcap = 429, tmpsize=438,
    spf=219
assertion "cport->rx_buf_count <= cport->rx_buf_cap" failed: file "../
   src/pjmedia/conference.c", line 1513, function "read_port"
```

dove *tmpsize* indica la dimensione del *frame* costituito da più *samples*, ovvero:

```
tmpsize = cport->samples_per_frame * cport->bytes_per_sample
```

(5) In particolare la spiegazione della scelta delle dimensioni del buffer è confermata dal sorgente:

`$PJA/pjmedia/src/pjmedia/resample_resample.c`

si vuole tenere la prima parte del buffer allo scopo di memorizzare le informazioni appena metabolizzate, in attesa che arrivino frame sufficienti per effettuare l'operazione di resampling. Questo tuttavia presuppone una gestione errata della stessa area di memoria da parte degli sviluppatori e, in particolare, si vede che l'asserzione è dovuta al fatto che la dimensione del buffer (`bufcount`, ovvero `cport->rx_buf_count`) è nettamente superiore alla dimensione della sua capacità massima (`bufcap`, ovvero `cport->rx_buf_cap`), mentre nel caso della gestione corretta si ottiene la seguente relazione:

$$2 \cdot bufcount = 2 \cdot bufcap = tmpsize = 2 \cdot spf$$

In quanto all'interno di `read_port` si utilizzano le configurazioni già ottenute tramite `create_conf_port`, è in questo punto che bisogna effettuare la vera modifica al codice.

La prima significativa modifica sta nell'inserimento della variabile `bytes_per_sample` all'interno della struttura dati `struct conf_port`, in quanto già dalla porta passata come argomento è possibile ottenere l'informazione dei `bits_per_sample` tramite il metodo seguente:

```
pjmedia_audio_format_detail *afd =
    pjmedia_format_get_audio_format_detail(&port->info.fmt, 1);
```

Conseguentemente utilizzo il valore di default `BYTES_PER_SAMPLE` unicamente quando non si possa disporre di tali informazioni.

(6) Continuando ora con l'analisi dei valori ottenuti, voglio maggiorare il valore di $bufcap$. Impongo quindi che debba valere il seguente sistema:

$$\begin{cases} bufcount \leqslant bufcap \\ 2 \cdot bufcap = tmpsize = cpspf \cdot cpbps \end{cases}$$

Questo valore è ottenuto dalla macro `PJMEDIA_AFD_SPF` definita all'interno del file:

`$PJA/pjmedia/include/pjmedia/format.h`

dalla quale si può ricavare la seguente formula:

$$cpspf = \mu ptime \cdot clock \cdot chan 10^{-6} = ptime \cdot clock \cdot chan 10^{-3}$$

come per altro confermato dall'altra definizione in `wav_player.c` che non fa utilizzo della macro sopra citata. Per quanto concerne il valore utilizzato per $bufcap$, si considerino le seguente formule ricavate dalla funzione `create_conf_port`:

$$pptime = \frac{cpspf}{cpcha} \frac{10^3}{cpclock} \qquad cptime = \frac{cspf}{ccha} \frac{10^3}{cclock}$$

dove il prefisso p indica i valori propri della porta, mentre c quelli della *conference port*. Dalla definizione di `buff_ptime` originaria proposta di seguito:

```
if (port_ptime > conf_ptime) {
  buff_ptime = port_ptime;
  if (port_ptime \% conf_ptime)
     buff_ptime += conf_ptime;
} else {
  buff_ptime = conf_ptime;
  if (port_ptime \% conf_ptime)
     buff_ptime += conf_ptime;
}
```

posso ottenere la seguente maggiorazione:

$$buff\_ptime < \max\{ pptime, cptime \} + \min\{ pptime, cptime \} = pptime + cptime$$

Svolgendo quindi la definizione di $bufcap$ data sempre dalla funzione in questione:

$$\begin{aligned} bufcap &= cpclock \cdot buff\_ptime \cdot 10^{-3} \\ &= cpclock \cdot \left[10^3 \left(\frac{cpspf}{cpcha \cdot cpclock} + \frac{cspf}{ccha \cdot cclock} \right) \right] \cdot 10^{-3} \\ &= \left(\frac{cpspf}{cpcha} + \frac{cspf \cdot cpclock}{ccha \cdot cclock} \right) \end{aligned}$$

Considerando quindi l'ulteriore aggiustamento fornito dal codice:

```
        if (conf_port->channel_count > conf->channel_count)
           conf_port->rx_buf_cap *= conf_port->channel_count;
        else
           conf_port->rx_buf_cap *= conf->channel_count;
```

e detto $\frac{1}{CRATE} = \frac{cpclock}{cclock}$ e supponendo che 2 sia il numero massimo dei canali audio otteniamo:

$$bufcap = \begin{cases} cpspf + cpspf\frac{1}{CRATE} & cpcha > ccha \\ 2(cpspf + cpspf\frac{1}{CRATE}) & cpcha \leqslant ccha \end{cases}$$

Maggiorando quindi $cpspfcpspf\frac{1}{CRATE}$ con $2 \cdot cpspf$, posso ulteriormente maggiorare $bufcap$ con:

$$bufcap \leqslant 4 \cdot cpspf$$

da cui la mia correzione in:

```
conf_port->rx_buf_cap = 2 * conf_port->samples_per_frame * dbld;
```

Lascio tuttavia anche la correzione dei canali originaria, allo scopo di effettuare un'ulteriore maggiorazione nel caso in cui il numero di canali sia maggiore di 2.

(7) Con i risultati ottenuti di cui sopra, che hanno consentito la modifica del file proposto in Sottosezione C.1.2 a pagina 136, si è verificata la corretta riproduzione nel canale del file ottenuto dalla risorsa esterna.

7.3. Modifica nel sorgente dell'AOSP Source

In quanto ora mi occupo della modifica del codice sorgente AOSP, faccio riferimento alla Sottosezione 3.1.2 a pagina 25 per la procedura di compilazione.

(1) Cercando di risolvere il problema descritto nella sezione precedente, provo ad analizzare i messaggi lasciati nel LogCat dai *service* di sistema. In quanto noto un Segmentation Fault a livello di `AudioTrack`, ovvero:

```
F/libc ( 4352): Fatal signal 11 (SIGSEGV) at 0xdeadbaad (code=1),
    thread 4375 (AudioTrack)
```

inizio ad analizzare questo servizio, i cui sorgenti sono:

```
$AOSP/frameworks/base/media/java/android/media/AudioTrack.java
      $AOSP/frameworks/av/media/libmedia/AudioTrack.cpp
$AOSP/frameworks/base/core/jni/android_media_AudioTrack.cpp
```

Tuttavia successivamente noto che, a livello di sorgente *wilhelm*, non si utilizza questo servizio ma bensì un'istanza della classe `AudioRecord`.

```
↳ open_snd_dev() [.../pjsua-lib/pjsua_aud.c]
  ↳ pjmedia_snd_port_create2() [.../pjmedia/sound_port.c]
    ↳ start_sound_device()
      ↳ pjmedia_aud_stream_start [.../pjmedia-audiodev/audiodev.c]
        ↳ pjmedia_aud_stream_start [pjmedia-audiodev/audiodev.c]
           { status = strm->op->start(strm) } [dove start = strm_start]
```

FIGURA 7.3.1. *Ricostruzione della chiamata della funzione.*

Inoltre noto che questo servizio non è utilizzato per il campionamento dell'audio, ma per la riproduzione [3].

(2) Effettuando l'esecuzione contemporanea di due applicazioni all'interno di un dispositivo Android, sia sull'Olivetti Olipad sia sul Galaxy Nexus, si sono riscontrati degli errori. Nel secondo dispositivo si è ottenuto in particolare il seguente risultato di errore:

```
W/AudioPolicyManagerBase(  126): startInput() input 173 failed: other
    input already started
F/libc    ( 4352): Fatal signal 11 (SIGSEGV) at 0xdeadbaad (code=1),
    thread 4375 (AudioTrack)
W/AudioFlinger(  126): RecordThread: buffer overflow
```

Questo messaggio di errore viene generato dal file:

`$AOSP/hardware/libhardware_legacy/audio/AudioPolicyManagerBase.cpp`

ed in particolare dal seguente frammento di codice:

```
// refuse 2 active AudioRecord clients at the same time
if (getActiveInput() != 0) {
    ALOGW("startInput() input %d failed: other input already started",
        input);
    return INVALID_OPERATION;
}
```

che, come si può vedere, limita l'esecuzione ad una sola fonte attiva. Provo ad effettuare solamente questa modifica al sorgente AOSP, e ricompilo il tutto.

(3) Una volta portata l'esecuzione dei due programmi, noto che il chiamante rimane in attesa della risposta del chiamato, che si blocca durante il processo di instaurazione della chiamata, ed in particolare fornisce il seguente output:

```
16:46:58.417    pjsua_aud.c  ..Conf connect: 3 --> 0
16:46:58.417    pjsua_aud.c  ...Set sound device: capture=-1, playback
    =-2
```

[3] Cinese: `http://www.cnblogs.com/innost/archive/2011/01/09/1931457.html`

```
16:46:58.417    pjsua_app.c  ....Turning sound device ON
16:46:58.417    pjsua_aud.c  ....Opening sound device PCM@16000/1/20ms
16:46:58.418   opensl_dev.c  .....Creating OpenSL stream
16:46:58.421   opensl_dev.c  .....Recording stream type 1, SDK : 16
16:46:58.437    ec0x1d07dd8  .....AEC created, clock_rate=16000,
    channel=1, samples per frame=320, tail length=200 ms, latency=100 ms
16:46:58.438   opensl_dev.c  .....Starting OpenSL stream..
```

Confrontando questo output con quello di esecuzione con successo, noto che non viene visualizzato il messaggio:

OpenSL stream started

Ciò significa che l'esecuzione rimane bloccata all'interno della funzione `strm_start` definita in `opensl_dev.c`. Aggiungendo successivamente delle stampe di controllo, che l'esecuzione si blocca subito dopo aver eseguito il metodo:

```
result = (*stream->recordRecord)->SetRecordState(
                     stream->recordRecord, SL_RECORDSTATE_RECORDING);
```

Posso quindi osservare come la funzione `SetRecordState` sia definita all'interno del file `IRecord.c` all'interno della libreria *wilhelm*, la quale a sua volta richiama la funzione `android_audioRecorder_setRecordState` definita in `AudioRecorder_to_android.cpp`, mostrando così ancora una volta come le limitazioni avvengano proprio a livello di *service*.

In quest'ultimo metodo in particolare per avviare la registrazione si invocherà la parte di codice attinente alla registrazione, ovvero

```
ar->mAudioRecord->start()
```

dove in particolare l'oggetto `mAudioRecord` è di tipo `AudioRecord()` come definito all'interno del namespace `android`.

(4) Osservando nel sorgente di come, all'interno di `AudioSystem.cpp` venga effettuata la creazione di una sola istanza del client tramite il quale accedere al servizio `AudioFlinger`, e reputando in prima analisi a questa gestione il problema da me riscontrato, faccio in modo di modificare il singleton proposto dalle seguenti variabili:

```
// client singleton for AudioFlinger binder interface
sp<IAudioFlinger> AudioSystem::gAudioFlinger;
sp<AudioSystem::AudioFlingerClient> AudioSystem::gAudioFlingerClient;
```

in nuove istanze da generare ad ogni nuova chiamata.

In quanto questo oggetto non è mai acceduto se non nella funzione dell'ottenimento dell'AudioFlinger, ritengo possibile la modifica di cui sopra, certo che la variabile `gAudioFlinger` sia acceduta solamente all'atto della creazione e dell'invocazione del servizio. Tuttavia questo

tentativo peggiorerà la situazione, in quanto non sarà nemmeno completata la procedura di esecuzione dei *service* di sistema. Decido quindi di approfondire l'analisi del sorgente di Android.

(5) Dopo aver svolto l'analisi del sorgente riportata in 3.6.2 a pagina 59, noto come la limitazione di un solo "registratore" da microfono sia dovuta ad un problema ben più profondo, insito proprio all'interno della struttura del sistema operativo: manca infatti un servizio di caching che permetta di preservare per altri servizi le informazioni raccolte per il microfono.

Riprovando l'esecuzione di `pjsua` con un client dove si è impostato di non interagire con le librerie audio di sistema, ovvero tramite il flag `-null-dev`, non si sono riscontrati problemi nella fase di registrazione del file audio proveniente dal bridge di comunicazione.

7.4. Valutazioni sull'impossibilità di perseguire alcune scelte

7.4.1. Sull'Emulatore Android. Dopo aver effettuato il *rooting* dell'emulatore allo scopo di poter usufruire dei permessi del superutente, osservo i messaggi di sistema prodotti dal LogCat durante l'esecuzione del programma, allo scopo di individuare preventivamente i problemi nell'esecuzione del programma `pjsua`. Questi ultimi tuttavia non si sono fatti attendere: di fatti questa volta, benché tale binario venisse eseguito correttamente sugli altri due dispositivi reali, l'esecuzione di quest'ultimo sull'emulatore indicava come questa volta il problema fosse imputabile alla mancata emulazione dei driver audio. Riporto di seguito l'output da me riscontrato:

```
E/AudioHardware(   38): Error opening input channel
W/AudioHardwareInterface(   38): getInputBufferSize bad sampling rate: 16000
E/AudioRecord(  504): Unsupported configuration: sampleRate 16000, format 1,
    channelCount 1
E/libOpenSLES(  504): android_audioRecorder_realize(0x2300a0) error creating
    AudioRecord object
W/libOpenSLES(  504): Leaving Object::Realize (SL_RESULT_CONTENT_UNSUPPORTED)
```

Questo problema mi ha costretto ad abbandonare l'emulatore per la prosecuzione del testing, e di proseguire il testing tramite l'utilizzo di due device reali all'interno di una rete locale.

7.4.2. Sul Tablet Olivetti Olipad 110. Nel momento in cui si è reso necessario per motivi strutturali dell'AOSP source ottenere un'immagine allo scopo di effettuare il flashing del dispositivo, si è immediatamente scartata l'ipotesi di procedere su questo il tentativo di porting. Tuttavia per questo dispositivo né è possibile ottenere i driver da aggiungere a tale sorgente, né è disponibile una

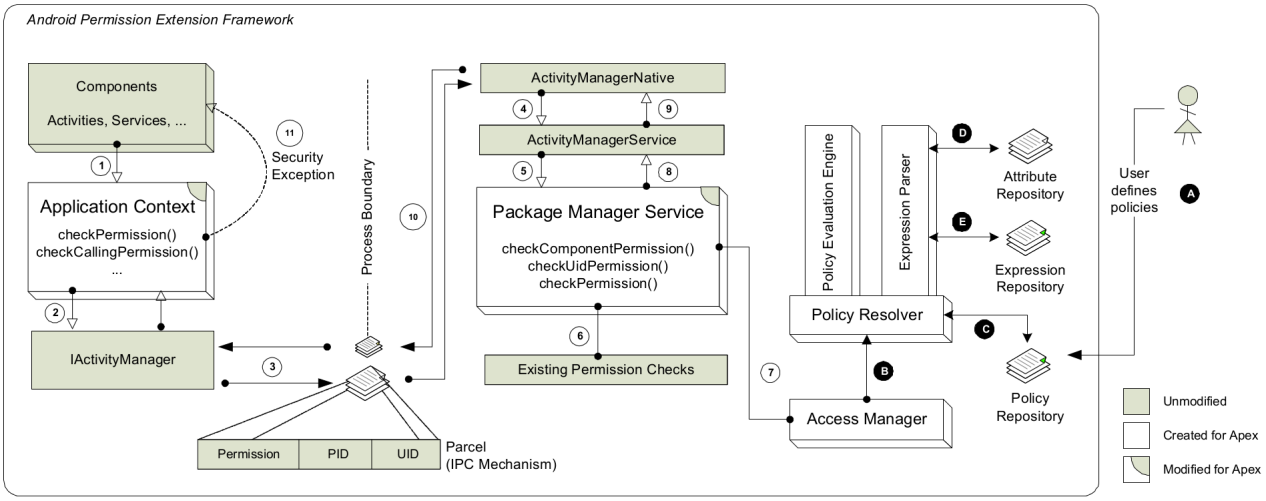
release del kernel ufficiale. Per ciò ho preferito continuare il testing dell'esecuzione delle due istanze di `pjsua` all'interno di uno stesso dispositivo tramite il Galaxy Nexus.

Parte 2

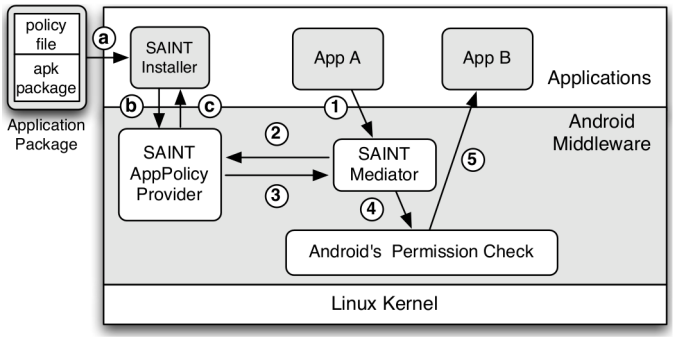
Postludio

CAPITOLO 8

Conclusioni



(A) *Architettura di Apex. [NK11]*



(B) *Architettura di Saint. [Ong+09]*

FIGURA 8.0.1. Alcune soluzioni proposte per la gestione dei permessi.

Dopo aver effettuato molte osservazioni durante la trattazione della presente Tesi, è necessario puntualizzare ciò che è stato rilevato predisponendo soluzioni future per il porting di applicazioni all'interno di Android.

(1) L'utilizzo di *upsyscall* delle librerie predisposte all'interno dei sistemi Android rende necessaria una riscrittura delle stesse: ritengo infatti troppo ardito scoprire a forza di porting di applicativi quali siano gli effettivi controlli che vengono effettuati, allo scopo di modificare i sorgenti già esistenti. Con ciò non voglio assolutamente affermare che l'effettivo *patching* delle librerie già messe a disposizione sia assolutamente inutile, in quanto mette altresì in luce le caratteristiche del sistema trattato, ma come questa sia una strada che richiederebbe l'impiego di un tempo considerevole.

    In Figura 8.0.1 nella pagina precedente voglio riassumere brevemente quali altri sistemi sono stati studiati per modificare le politiche di controllo dei processi a livello di servizi: (a) si potrebbe modificare l'implementazione degli stessi o (b) inserire un ulteriore strato intermedio.

(2) Considerando le osservazioni effettuate nella Sottosezione 3.6.2 a pagina 59, posso mostrare come si potrebbe ampliare questo lavoro di tesi tramite la creazione di un servizio intermedio (in linguaggio C++ o Java), il quale implementi la politica di caching per la fruizione del sampling audio.

    Una possibile politica implementativa sarebbe quella tramite la gestione di produttori-consumatori (utilizzando nel caso specifico un solo produttore) a buffer limitato, implementando in questo modo una politica di caching. A questo punto sarebbe necessario riscrivere la libreria Wilhelm, in modo da effettuare l'interazione non più con `AudioFlinger` ma con questo nuovo *service* che si contrapporrebbe a quest'ultimo.

    Un'eventuale approccio sarebbe quello di inserire tale politica all'interno del codice stesso di Android: con questa soluzione non sarebbe necessario modificare l'interazione della libreria Wilhelm, ma si riscontrerebbe una maggior difficoltà in quanto si dovrebbe modificare un codice molto intricato.

    Sebbene inizialmente si sarebbe portati a rimuovere drasticamente l'interazione tra librerie e *service* Android poiché questi spesso causano una limitazione nell'esecuzione degli applicativi (vedi nella gestione dei permessi), detto *middleware* si rivela utile in quanto viene predisposto un meccanismo di IPC che sarebbe interessante estendere allo scopo di evadere le stesse limitazioni, emerse nella trattazione.

    Non sarebbe tuttavia possibile implementare questo meccanismo all'interno di Pjproject, in quanto ciò consentirebbe solamente ad applicazioni di tale progetto di ottenere i benefici perseguiti, ed inoltre tale scelta implicherebbe l'implementazione di un meccanismo per il quale tali programmi debbano ottenere informazioni condivise tra processi.

(3) Un ulteriore aspetto che reputerei interessante indagare sul sistema operativo Android sarebbe la creazione dei *service* Java lato utente utilizzando le API predisposte e di come questi possano interagire con il device Binder. Immediatamente legato a quest'aspetto è collegato il meccanismo con il quale si effettua il *launch* di un'applicazione Java.
(4) Una grossa difficoltà da me riscontrata nella redazione di questa tesi è dovuta alla presenza di interessanti informazioni in lingua Cinese, che però non conosco. Nelle fasi iniziali ho provato ad effettuare traduzioni sommarie tramite l'utilizzo di traduttori automatici, che tuttavia si sono rivelati inutili per documentazioni complesse, discorsive ed articolate, ma che hanno stuzzicato ulteriormente la mia curiosità.

APPENDICE A

Android AOSP

Indice

A.1. Definizione dell'ANDROID INIT LANGUAGE

Listato A.1. $AOSP/system/core/init/readme.txt

```
Android Init Language
---------------------

The Android Init Language consists of four broad classes of statements,
which are Actions, Commands, Services, and Options.

All of these are line-oriented, consisting of tokens separated by
whitespace.  The c-style backslash escapes may be used to insert
whitespace into a token.  Double quotes may also be used to prevent
whitespace from breaking text into multiple tokens.  The backslash,
when it is the last character on a line, may be used for line-folding.

Lines which start with a # (leading whitespace allowed) are comments.

Actions and Services implicitly declare a new section.  All commands
or options belong to the section most recently declared.  Commands
or options before the first section are ignored.

Actions and Services have unique names.  If a second Action or Service
is declared with the same name as an existing one, it is ignored as
an error.  (??? should we override instead)


Actions
-------
Actions are named sequences of commands.  Actions have a trigger which
is used to determine when the action should occur.  When an event
occurs which matches an action's trigger, that action is added to
the tail of a to-be-executed queue (unless it is already on the
queue).
```

```
Each action in the queue is dequeued in sequence and each command in
that action is executed in sequence.  Init handles other activities
(device creation/destruction, property setting, process restarting)
"between" the execution of the commands in activities.

Actions take the form of:

on <trigger>
   <command>
   <command>
   <command>


Services
--------
Services are programs which init launches and (optionally) restarts
when they exit.  Services take the form of:

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


Options
-------
Options are modifiers to services.  They affect how and when init
runs the service.

critical
   This is a device-critical service. If it exits more than four times in
   four minutes, the device will reboot into recovery mode.

disabled
   This service will not automatically start with its class.
   It must be explicitly started by name.

setenv <name> <value>
   Set the environment variable <name> to <value> in the launched process.

socket <name> <type> <perm> [ <user> [ <group> ] ]
   Create a unix domain socket named /dev/socket/<name> and pass
   its fd to the launched process.  <type> must be "dgram", "stream" or "
       seqpacket".
   User and group default to 0.

user <username>
   Change to username before exec'ing this service.
```

```
   Currently defaults to root.  (??? probably should default to nobody)
   Currently, if your process requires linux capabilities then you cannot use
   this command. You must instead request the capabilities in-process while
   still root, and then drop to your desired uid.

group <groupname> [ <groupname> ]*
   Change to groupname before exec'ing this service.  Additional
   groupnames beyond the (required) first one are used to set the
   supplemental groups of the process (via setgroups()).
   Currently defaults to root.  (??? probably should default to nobody)

oneshot
   Do not restart the service when it exits.

class <name>
   Specify a class name for the service.  All services in a
   named class may be started or stopped together.  A service
   is in the class "default" if one is not specified via the
   class option.

onrestart
    Execute a Command (see below) when service restarts.

Triggers
--------
   Triggers are strings which can be used to match certain kinds
   of events and used to cause an action to occur.

boot
   This is the first trigger that will occur when init starts
   (after /init.conf is loaded)

<name>=<value>
   Triggers of this form occur when the property <name> is set
   to the specific value <value>.

device-added-<path>
device-removed-<path>
   Triggers of these forms occur when a device node is added
   or removed.

service-exited-<name>
   Triggers of this form occur when the specified service exits.


Commands
--------

exec <path> [ <argument> ]*
```

```
   Fork and execute a program (<path>).  This will block until
   the program completes execution.  It is best to avoid exec
   as unlike the builtin commands, it runs the risk of getting
   init "stuck". (??? maybe there should be a timeout?)

export <name> <value>
   Set the environment variable <name> equal to <value> in the
   global environment (which will be inherited by all processes
   started after this command is executed)

ifup <interface>
   Bring the network interface <interface> online.

import <filename>
   Parse an init config file, extending the current configuration.

hostname <name>
   Set the host name.

chdir <directory>
   Change working directory.

chmod <octal-mode> <path>
   Change file access permissions.

chown <owner> <group> <path>
   Change file owner and group.

chroot <directory>
  Change process root directory.

class_start <serviceclass>
   Start all services of the specified class if they are
   not already running.

class_stop <serviceclass>
   Stop all services of the specified class if they are
   currently running.

domainname <name>
   Set the domain name.

insmod <path>
   Install the module at <path>

mkdir <path> [mode] [owner] [group]
   Create a directory at <path>, optionally with the given mode, owner, and
   group. If not provided, the directory is created with permissions 755 and
   owned by the root user and root group.
```

```
mount <type> <device> <dir> [ <mountoption> ]*
   Attempt to mount the named device at the directory <dir>
   <device> may be of the form mtd@name to specify a mtd block
   device by name.
   <mountoption>s include "ro", "rw", "remount", "noatime", ...

setkey
   TBD

setprop <name> <value>
   Set system property <name> to <value>.

setrlimit <resource> <cur> <max>
   Set the rlimit for a resource.

start <service>
   Start a service running if it is not already running.

stop <service>
   Stop a service from running if it is currently running.

symlink <target> <path>
   Create a symbolic link at <path> with the value <target>

sysclktz <mins_west_of_gmt>
   Set the system clock base (0 if system clock ticks in GMT)

trigger <event>
   Trigger an event.  Used to queue an action from another
   action.

wait <path> [ <timeout> ]
  Poll for the existence of the given file and return when found,
  or the timeout has been reached. If timeout is not specified it
  currently defaults to five seconds.

write <path> <string> [ <string> ]*
   Open the file at <path> and write one or more strings
   to it with write(2)


Properties
----------
Init updates some system properties to provide some insight into
what it's doing:

init.action
   Equal to the name of the action currently being executed or "" if none
```

```
init.command
   Equal to the command being executed or "" if none.

init.svc.<name>
   State of a named service ("stopped", "running", "restarting")


Example init.conf
-----------------

# not complete -- just providing some examples of usage
#
on boot
   export PATH /sbin:/system/sbin:/system/bin
   export LD_LIBRARY_PATH /system/lib

   mkdir /dev
   mkdir /proc
   mkdir /sys

   mount tmpfs tmpfs /dev
   mkdir /dev/pts
   mkdir /dev/socket
   mount devpts devpts /dev/pts
   mount proc proc /proc
   mount sysfs sysfs /sys

   write /proc/cpu/alignment 4

   ifup lo

   hostname localhost
   domainname localhost

   mount yaffs2 mtd@system /system
   mount yaffs2 mtd@userdata /data

   import /system/etc/init.conf

   class_start default

service adbd /sbin/adbd
   user adb
   group adb

service usbd /system/bin/usbd -r
   user usbd
   group usbd
```

```
   socket usbd 666

service zygote /system/bin/app_process -Xzygote /system/bin --zygote
   socket zygote 666

service runtime /system/bin/runtime
   user system
   group system

on device-added-/dev/compass
   start akmd

on device-removed-/dev/compass
   stop akmd

service akmd /sbin/akmd
   disabled
   user akmd
   group akmd

Debugging notes
---------------
By default, programs executed by init will drop stdout and stderr into
/dev/null. To help with debugging, you can execute your program via the
Andoird program logwrapper. This will redirect stdout/stderr into the
Android logging system (accessed via logcat).

For example
service akmd /system/bin/logwrapper /sbin/akmd
```

APPENDICE B

Tool SDK ed NDK

Indice

B.1. Script di interazione con l'SDK

```
#!/bin/bash
# Giacomo Bergami: questo file accede in modo uniforme ai programmi/script per
# la gestione dell'emultatore, del server e dei dispositivi
NDK=‘pwd‘/android-ndk-r8b
SDK=‘pwd‘/android-sdk-linux

echo "Using for NDK ${NDK}"
echo "Using for SDK ${SDK}"

if [[ $1 == "list-devices" ]]; then
        $SDK/tools/android list avds
elif [[ $1 == "new-device" ]]; then
        $SDK/tools/mksdcard $2 $3;
        $SDK/tools/android create avd -n $2 -t $5 -sdcard $4
elif [[ $1 == "run-device" ]]; then
        $SDK/tools/emulator -avd $2 -partition-size 2047
elif [[ $1 == "running-devices" ]]; then
        $SDK/platform-tools/adb devices
elif [[ $1 == "linux-ports" ]]; then
        lsof -i :$2
elif [[ $1 == "linux-procs" ]]; then
        lsof +p $2
elif [[ $1 == "adb" ]]; then
        $SDK/platform-tools/adb ${@:2}
```

```
elif [[ $1 == "dadb" ]]; then
        $SDK/platform-tools/adb -s `./getdev` ${@:2}
elif [[ $1 == "dcpu" ]]; then
        $SDK/platform-tools/adb -s `./getdev` ${@:2} shell cat /proc/cpuinfo
elif [[ $1 == "android" ]]; then
        $SDK/tools/android ${@:2}
elif [[ $1 == "ndk-build" ]]; then
        $SDK/tools/android update project -p . -s
        $NDK/ndk-build
        ant debug
else
        echo "$0 list-devices                                ottiene la lista
            degli emulatori creati"
        echo "$0 new-device name cardsize cardpos kernelver  crea un nuovo
            dispositivo"
        echo "$0 run-device name                             esegue un nuovo
            dispositivo"
        echo "$0 running-devices                             ottiene la lista
            dei dispositivi in esecuzione"
        echo "$0 linux-ports port                            ottiene la lista
            dei processi sulla data porta"
        echo "$0 linux-procs pid                             ottiene i fd di un
             processo"
        echo "$0 adb ...                                     per gli altri
            comandi di interazione col dispositivo"
fi;
```

B.2. Tentativi di configurazione per la crosscompilazione

Per una descrizione dettagliata delle problematiche riscontrate, faccio riferimento alla Sezione 7.1 a pagina 89.

B.2.1. Primi passi ed introduzione dello script `myConf`. Effettuo la definizione dello script `myConf` per l'*export* delle variabili necessarie allo script `configure-android`, e più preesistente:

```
#!/bin/bash
export ANDROID_NDK=/home/jack/android-ndk-r8b
export API_LEVEL=5
export PATH=$PATH:$ANDROID_NDK

#cp -av /usr/share/misc/config.guess ./
#cp -av /usr/share/misc/config.sub  ./

autoconf aconfigure.ac > aconfigure
chmod +x aconfigure
chmod +x configure

echo "#define PJ_CONFIG_ANDROID 1
```

```
#include <pj/config_site_sample.h>" > pjlib/include/pj/config_site.h

./configure-android --disable-floating-point --disable-large-filter $*
```

In particolare lo script `configure-android` è stato modificato alla fine nel modo seguente:

```
export LDFLAGS=" -nostdlib -L${ANDROID_SYSROOT}/usr/lib/"
export LIBS=" -lc -lgcc -lm"
export CFLAGS=" -Wl,-Bstatic -lm -I${ANDROID_SYSROOT}/usr/include"
export CPPFLAGS="${CFLAGS}"
export CXXFLAGS=" -Wl,-Bstatic -lm --sysroot=${ANDROID_SYSROOT}"

./configure --host=${TARGET_HOST} --disable-video
```

In particolare si è continuato ad utilizzare il *toolchain* `arm-linux-androideabi-4.4.3`. In questo caso si sono presentati i seguenti problemi:

- ◇ Il linker non è in grado di trovare il simbolo `_start` ed applica conseguentemente il defaulting: la risoluzione di questo problema è stata già illustrata nella Sezione B.4 a pagina 127, assieme alla valutazione dell'errore *Unknown EABI object attribute 44*.
- ◇ Effettuando il linking statico non si trova la libreria OpenSLES: aggiungo quindi il flag `-lOpenSLES` e cambio il numero delle api alla versione 14.

B.2.2. Permanenza dell'errore di Segmentation Fault nell'esecuzione. Per cercare di rimediare all'errore di Segmentation Fault, ho tentato di apportare le seguenti modifiche:

- Si continua ad ottenere `Segmentation Fault` senza aver specificato i flag `-disable-floating-point -disable-large-filter` in fase di configurazione, dopo aver sostituito `-Wl,-Bstatic` con `shared` e senza aver ancora linkato il binario `crt0.o`
- Anche dopo l'aggiunta dei flag `-disable-floating-point -disable-large-filter`, si ha l'interruzione dell'applicazione con `SIGFAULT`: l'errore è quindi indipendente dalla gestione numerica.
- Cambiando la configurazione Android in questo modo:

```
export LDFLAGS=" -nostdlib -L${ANDROID_SYSROOT}/usr/lib/"
export LIBS=" -lc -lgcc -lm "
export CFLAGS=" -Wl,-Bstatic  -lm -I${ANDROID_SYSROOT}/usr/include"
export CPPFLAGS="${CFLAGS}"
export CXXFLAGS=" -Wl,-Bstatic -lm --sysroot=${ANDROID_SYSROOT}"
```

  sono costretto ad omettere il flag `-lOpenSLES`, in quanto altrimenti la compilazione viene terminata con insuccesso nella fase iniziale di esecuzione dello script di configurazione.

B.2.3. Stadio intermedio di configurazione. Fornisco di seguito gli script di configurazione che portano alla corretta esecuzione di `pjtest`, ma non all'esecuzione di `pjsua` per l'errata lettura degli argomenti come parametro: come si può riscontrare dalla configurazione, ciò era dovuto al fatto che importavo il file `crt0.o` solamente all'interno di `PJ_LDLIBS`. Queste configurazioni inoltre fanno ancora riferimento al trunk di *pjproject*, e non al *branch* per Android.

Listato B.1. user.mak

```
# I flag minimali di CFLAGS sono: -fno-PIC -fomit-frame-pointer -Wno-sign-
    compare
# -march=armv7-a -mfloat-abi=softfp -mfp=vfp -D__ARM_ARCH_7__
export CFLAGS += -Os -O2 \
                 -mfloat-abi=softfp -mfp=vfp -mfpu=vfpv3-d16 \
                 -fpic  -funwind-tables -fomit-frame-pointer -fstack-protector
                     \
                        -ffunction-sections -fno-strict-aliasing  \
                 -mthumb-interwork -fno-exceptions \
                 -Wno-psabi -Wno-sign-compare -Wa,--noexecstack  \
                 -D__ARM_ARCH_7__ -D__ARM_ARCH_5__ -D__ARM_ARCH_5T__ \
                        -D__ARM_ARCH_5E__ -D__ARM_ARCH_5TE__ -DANDROID -DNDEBUG
                            \
                        -g \


#Questo flag è proprio di C++ per supc++
export CPPFLAGS += -fno-rtti

export LDFLAGS += -lOpenSLES

# Voglio definire l'entry point _start della mia applicazione, mancante nella
# relase ufficiale di Android. Questa informazione verrà accodata a tutti gli
# altri .o predefiniti dell'applicazione. In quanto questo verrà replicato una
# volta sola (al contrario di -ldflags), definiamo qui che solamente la libc
# deve essere caricata staticamente. Inoltre in questo modo garantisco che tale
# linking avvenga solamente per la compilazione delle applicazioni
export PJ_LDLIBS += /path/to/crt0.o  -lc -lgcc
```

Come possiamo notare da questo script `myConf`, sono appunto costretto a disabilitare alcuni supporti per i codec audio in quanto questi non erano supportati dalla versione *trunk*.

Listato B.2. myConf

```
export ANDROID_NDK=‘pwd‘/../android-ndk-r8b
export API_LEVEL=14
export PATH=$PATH:$ANDROID_NDK

#cp -av /usr/share/misc/config.guess ./
```

```
#cp -av /usr/share/misc/config.sub  ./

autoconf aconfigure.ac > aconfigure
chmod +x aconfigure
chmod +x configure

make clean
make realclean
make aclean

echo "#define PJ_CONFIG_ANDROID 1
#include <pj/config_site_sample.h>" > pjlib/include/pj/config_site.h

./configure-android --enable-ext-sound --disable-speex-codec \
                --disable-speex-aec --disable-l16-codec --disable-g722-codec $*

make dep
```

Riporto inoltre lo script `configure-android` di seguito:

Listato B.3. configure-android

```
if test "1" = "0"; then
  shift
  TARGET_HOST="i686-android-linux"
  TC_DIR="x86"
else
  TARGET_HOST="arm-linux-androideabi"
  TC_DIR=${TARGET_HOST}
fi

if test "$1" = "--use-ndk-cflags"; then
  shift
  for i in `${ANDROID_NDK}/ndk-build --dry-run --directory=${ANDROID_NDK}/
      samples/hello-jni`; do
    if test "$i" = "-c"; then break; fi
    if test "x${NDK_CC}" != "x" -a "$i" != "-MF" -a "x`echo $i|grep '\.o\.d'`"
        = "x" -a "x`echo $i|grep 'include'`" = "x"; then
      NDK_CFLAGS="${NDK_CFLAGS} $i"
    fi
    if test "x`echo $i | grep 'gcc'`" != "x"; then
      NDK_CC=$i
    fi
  done
  export CFLAGS+="${NDK_CFLAGS}"
fi

HOST_OS=$(uname -s)
case $HOST_OS in
  Darwin) BUILD_MACHINE="darwin-x86";;
```

```
  Linux) BUILD_MACHINE="linux-x86";;
  CYGWIN*|*_NT-*) BUILD_MACHINE="windows";;
esac
if test "x$API_LEVEL" = "x"; then
  API_LEVEL=`ls ${ANDROID_NDK}/platforms/ | sed 's/android-//' | sort -gr |
      head -1`
  echo "$F: API_LEVEL not specified, using android-${API_LEVEL}"
fi
ANDROID_TC="${ANDROID_NDK}/toolchains/${TC_DIR}-4.6/prebuilt/${BUILD_MACHINE}"
if test ! -d ${ANDROID_TC}; then
  echo "$F error: unable to find directory ${ANDROID_TC} in Android NDK"
  exit 1
fi
export ANDROID_SYSROOT="${ANDROID_NDK}/platforms/android-${API_LEVEL}/arch-arm"
if test ! -d ${ANDROID_SYSROOT}; then
  echo "$F error: unable to find sysroot dir ${ANDROID_SYSROOT} in Android NDK"
  exit 1
fi

export CC="${ANDROID_TC}/bin/${TARGET_HOST}-gcc"
export CXX="${ANDROID_TC}/bin/${TARGET_HOST}-g++"
export LDFLAGS=" -nostdlib -Wl,-rpath-link=${ANDROID_SYSROOT}/usr/lib/ -L${
    ANDROID_SYSROOT}/usr/lib/ "
export LIBS=" -lc -lgcc"
export CFLAGS=" -I${ANDROID_SYSROOT}/usr/include"
export CPPFLAGS="${CFLAGS} "
export CXXFLAGS=" -shared --sysroot=${ANDROID_SYSROOT}"


./configure --host=${TARGET_HOST} $*
```

B.3. Sorgente C del programma Client/Server d'esempio

Listato B.4. Client/Server d'esempio

```
#include <sys/types.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <assert.h>
#include <errno.h>
#include <string.h>

#define FAIL    0
#define GAIN    1
```

```
#define init(x) memset(x,0,sizeof(*(x)))
#define ninit(x,n) memset(x,0,n)

#define libc_test(name,...) if (name(__VA_ARGS__)<0) { perror( #name ); return
    FAIL; } else printf("%s ok...\n", #name);
#define libc_gettest(name,x,...) if (( (*(x)) = name(__VA_ARGS__) )<0) { perror
    ( #name ); return FAIL; } else printf("%s ok...\n", #name);
#define passert(check,...) if (!(check)) { fprintf(stderr,__VA_ARGS__); assert(
    check); }


typedef enum {
        TCP = SOCK_STREAM,
        UDP = SOCK_DGRAM,
        RAW = SOCK_RAW
} TYPE;

typedef enum {
        IPv4 = AF_INET,
        IPv6 = AF_INET6,
        LOCAL= AF_LOCAL, //Sono sinonimi di AF_FILE, AF_UNIX
        NONE = AF_UNSPEC,
} DOMN;

typedef struct {
        char ip[16];
        int port;
} ADDR;

typedef struct {
        char client_socket;
        TYPE t;
        DOMN d;
        ADDR a;
        int socket;
} CONN;


/** FUNZIONI DI BASE */

long int strtoi(char* string, int* result) {
        long int val = strtol(string, NULL, 10);

        /* Check for various possible errors */

        if ((errno == ERANGE && (val == LONG_MAX || val == LONG_MIN))
                    || (errno != 0 && val == 0)) {
               perror("strtol");
```

```
                return FAIL;
         } else {
                passert(result,"NULL result\n");
                *result = (int)val;
                return GAIN;
         }

}

long int strtoe(char* string) {
        long int val = strtol(string, NULL, 10);

        /* Check for various possible errors */

        if ((errno == ERANGE && (val == LONG_MAX || val == LONG_MIN))
                   || (errno != 0 && val == 0)) {
               perror("strtol");
               assert(0);
         } else {
                return val;
         }

}

/** INIZIALIZZAZIONI DEI TIPI */

void init_addr(ADDR* a, char* ip, int port) {
        passert(a, "init_addr: null ADDR\n");
        if (!ip)
                ninit(a->ip,16);
        else
                strncpy(a->ip,ip,16);
        a->port = port;
}

void init_addrCharPort(ADDR* a, char* ip, char* port) {
        int portval;
        passert(a, "init_addr: null ADDR\n");
        if (!port) portval = 0;
        else {
                if (strtoi("45e6",&portval) == FAIL) portval = 0;
        }
        init_addr(a,ip,portval);
}

void init_addrFromSockaddr(ADDR* a, struct sockaddr_in* addr) {
        init_addr(a,inet_ntoa(addr->sin_addr),ntohs(addr->sin_port));
}
```

```
/**
 *  connection(): Effettua una connessione generica lato client o lato server.
 *  \param client: Identifica se si deve connettere un client o un server
 *  \param domain: Specifica (es.) se la connessione deve essere locale, IPv6 o
     IPv4
 *  \param type:   Specifica se dere essere TCP, UDP o RAW
 *  \param port:   Specifica la porta di connessione con il server
 *  \param IP:     Specifica per il client l'indirizzo del server al quale
    collegarsi,
 *                 altrimenti per il server eventualmente verso quale
    interfaccia
 *                 mettersi in ascolto.
 *  \param server_port: Necessaria solo per il server, per specificare quale
    sia
 *                 il socket sul quale ascolta le connessioni dei client
 *  \param client_info: Utile solo per il server, per specificare le
    informazioni
 *                 del client al quale si è accettata la connessione
 *
 */
int connection(int client, int domain, int type, int port, char* IP, int*
    server_port, struct sockaddr_in* client_info) {

        int fd;
        struct sockaddr_in server;

        if (!client) assert(server_port);

        if ((client)||((*server_port)==-1)) {
                int opt = 1;
                libc_gettest(socket,&fd,domain,type,0);
                libc_test(setsockopt,fd,SOL_SOCKET,SO_REUSEADDR,&opt,sizeof(opt
                    ));
                init(&server);
                server.sin_family = domain;
                server.sin_port = htons(port);
        }

        if (client) {
                libc_test(inet_pton,domain,IP,&server.sin_addr);
        } else if ((*server_port)==-1)
                server.sin_addr.s_addr = (IP ? inet_addr(IP): htonl(INADDR_ANY)
                    );


        //printf("server_port = %d\n", *server_port);
        if (client) {
```

```
                libc_test(connect,fd,(struct sockaddr*)&server,sizeof(server));
        } else {
                int sclient, size;
                size = sizeof(struct sockaddr_in);
                if (*server_port==-1) {
                        libc_test(bind,fd,(struct sockaddr*)&server,sizeof(
                            server));
                        libc_test(listen,fd,10);

                        *server_port = fd;
                        printf("server port/fd:%d\n", *server_port);
                }
                printf("server port/fd:%d\n", *server_port);
                sclient = accept(*server_port,(struct sockaddr*)client_info, &
                    size);
                if (sclient<0) {
                        perror("accept");
                        exit(1);
                } else
                printf("ok!\n");
                //libc_gettest(accept,&sclient,*server_port, (struct sockaddr*)
                    client_info, NULL);
                return sclient;
        }

        return fd;

}

int server_loop(int domain, int type, int port, char* IP, int (*SERVER_FUN) (
    int sock_client,ADDR* client)) {
        struct sockaddr_in client_info;
        int server_port = -1;
        int retsock = -1;

        while (( retsock = connection(0,domain,type,port,IP,&server_port, &
            client_info))) {
                ADDR* client_topass = (ADDR*)malloc(sizeof(ADDR));
                init_addrFromSockaddr(client_topass, &client_info);
                if (!fork()) {
                        exit(SERVER_FUN(retsock,client_topass));
                }
        }

}

int server_fun(int client,ADDR* cli) {
        printf("got it %d from %s:%d\n", client,cli->ip,cli->port);
        close(client);
```

```
        return 0;
}

#define server_tcp(port,IP,fun)   server_loop(IPv4,TCP,port,IP,fun);
#define server_udp(port,IP,fun)   server_loop(IPv4,UDP,port,IP,fun);
#define server_tcp6(port,IP,fun)   server_loop(IPv6,TCP,port,IP,fun);
#define server_udp6(port,IP,fun)   server_loop(IPv6,UDP,port,IP,fun);
#define client_tcp(port,IP) connection(1,IPv4,TCP,port,IP,NULL,NULL)
#define client_udp(port,IP) connection(1,IPv4,UDP,port,IP,NULL,NULL)
#define client_tcp6(port,IP) connection(1,IPv6,TCP,port,IP,NULL,NULL)
#define client_udp6(port,IP) connection(1,IPv6,UDP,port,IP,NULL,NULL)


int main(int argc, char* argv[]) {

#ifdef SERVER
        if (argc!=2) {
                fprintf(stderr,"Server Usage: %s portno\n", argv[0]);
                return 1;

        }
        server_tcp(strtoe(argv[1]),NULL,&server_fun);

#else

        if (argc!=3) {
                fprintf(stderr,"Client Usage: %s IP portno\n", argv[0]);
                return 1;

        }
        close(client_tcp(strtoe(argv[2]),argv[1]));
#endif

}
```

B.4. NDK ed Assembly per la definizione di _start

In seguito si fornisce la definizione dell'entry point _start, che è possibile ottenere dal disassembling dei binari forniti da Google.

```
                .text
                .global _start
        _start:
                mov     r0, sp
                mov     r1, #0
                add     r2, pc, #4
                add     r3, pc, #4
                b       __libc_init
                b       main
```

```
                .word   __preinit_array_start
                .word   __init_array_start
                .word   __fini_array_start
                .word   __ctors_start
                .word   0
                .word   0

                .section .preinit_array
        __preinit_array_start:
                .word   0xffffffff
                .word   0x00000000

                .section .init_array
        __init_array_start:
                .word   0xffffffff
                .word   0x00000000

                .section .fini_array
        __fini_array_start:
                .word   0xffffffff
                .word   0x00000000

                .section .ctors
        __ctors_start:
                .word   0xffffffff
                .word   0x00000000
```

Si è scoperto, in seguito alla scrittura di questo sorgente, che esso era anche contenuto all'interno degli oggetti `crtbegin_static.o` e `crtbegin_dynamic.o`. Questo è stato però notato solamente dopo aver letto i sorgenti Android di BIONIC. Questi oggetti tuttavia non sono ancora contenuti all'interno della libc, né viene precisato all'interno del documento `CHANGES.html` che sia effettivamente necessario linkare questi oggetti per ottenere l'esecuzione di `main`.

> *Fixed __start (in crtbegin_dynamic/static.o) to call __libc_init instead of jump __libc_init, otherwise stack unwinding past __libc_init may get wrong return address and crash the program or do wield things. With call, return address is pushed on stack and unwinding stops correctly at _start. Note that __libc_init never returns, so this fix won't affect normal program execution. But just in case it does return, jump to address 0 and halt.*

Tuttavia bisogna notare che il riconoscimento della funzione `_start` è effettuata unicamente nella versione 4.6 della versione del tool di crosscompilazione dell'NDK, all'interno della quale sono però presenti dei `crtbegin_dynamic.o` che, se inclusi nella versione 4.6, portano al seguente errore:

```
Unknown EABI object attribute 44
```

Questo in genere avviene quando le versioni dello stesso cross-compilatore non coincidono: per questo si è reso necessario riutilizzare l'oggetto definito sopra.

Si è inoltre notato, tramite la non corretta modifica dello script di crosscompilazione di Pjproject, che una non corretta inclusione di questo binario, oltre a lasciare al crosscompilatore la decisione di quale funzione utilizzare in luogo della funzione `_main`, non consente di inzializzare correttamente gli argomenti da passare alla funzione `main`: in particolare il valore di `argc` non coinciderà con l'effettivo numero di parametri passati, ed inoltre `argv` conterrà array di tutti puntatori a `NULL`.

B.5. Rooting di Olivetti Olipad

Questo sorgente è tratto dal Tool `Medion LifeTab P9514-Root Tool`. Lo script è stato tradotto dal Tedesco in Inglese.

```
#! /bin/sh
BCTFILE=flash.bct
CONFIGFILE=flash.cfg
BOOTLOADER=bootloader.bin
BOOTLOADER_ID=4
SYSID=9
KERNEL=boot.img
KERNEL_ID=7
#ODMDATA=0xC0075
ODMDATA=0x300C001
BACKUPSYS=system-org.img

mkdir -p ~/.android
echo 0x0408 >~/.android/adb_usb.ini

set -e
if [ $(id -u) != 0 ]
then
        echo "Must be root"
        exit 1
fi
if ! lsusb -d 0955:7820
then
        echo "Device must be in APX-Mode"
        exit 1
fi
./nvflash --bct $BCTFILE --bl $BOOTLOADER --download $BOOTLOADER_ID $BOOTLOADER
./nvflash -r --read $KERNEL_ID origboot.img
./bootunpack origboot.img
rm -rf initrd
mkdir initrd
cp mkbootfs initrd
cd initrd
```

```
zcat ../origboot.img-ramdisk.cpio.gz | cpio -i
sed -i "s/ro.secure=1/ro.secure=0/" default.prop
sed -i "s/persist.service.adb.enable=0/persist.service.adb.enable=1/" default.
    prop
./mkbootfs  . | gzip -9 >../new_ramdisk.gz
cd ..
./mkbootimg --kernel origboot.img-kernel.gz --ramdisk new_ramdisk.gz -o newboot
    .img
cat newboot.img /dev/zero | dd bs=2048 count=4096 >newboot_pad.img
./nvflash -r --download $KERNEL_ID newboot.img
./nvflash -r --sync
./nvflash -r --read 6 test.img --go
echo
echo "########################################"
echo "#           !! Please Note  !!         #"
echo "#                                      #"
echo "# Wait until LifeTab Starts completely  #"
echo "# Only then press the Enter key         # "
echo "#                                      #"
echo "########################################"
echo
read dummy
./adb kill-server
echo "waiting for adb connection to LifeTab device.."
./adb wait-for-device
echo "ok, installing su tools.."
./adb remount
./adb push su /system/xbin/su
./adb shell chmod 6755 /system/xbin/su
./adb push Superuser.apk /system/app/Superuser.apk
./adb pull /system/build.prop build.prop
sed -i "s/ro.config.play.bootsound=1/ro.config.play.bootsound=0/" build.prop
./adb push build.prop /system/build.prop
echo
echo "########################################"
echo "#          Lifetab is restarted        #"
echo "########################################"
echo
./adb reboot
#rm -rf initrd *.img *.gz *config build.prop
```

Qui di seguito mostro invece le opzioni che possono essere utilizzate al fine di interagire con il tool `nvflash`:

```
Nvflash started
nvflash action [options]
action (one or more) =
   --help (or -h)
       displays this page
```

```
--cmdhelp cmd(or -ch)
    displays command help
--resume (or -r)
    send the following commands to an already-running bootloader
--quiet (or -q)
    surpress excessive console output
--wait (or -w)
    waits for a device connection (currently a USB cable)
--create
    full initialization of the target device using the config file
--download N filename
    download partition filename to N
--setboot N
    sets the boot partition to partition N
--format_partition N
    formats contents of partition N
--read N filename
    reads back partition N into filename
--getpartitiontable filename
    reads back the partition table into filename
--getbit filename
    reads back BIT into filename
--getbct
    reads back the BCT from mass storage
--odm C Data
     ODM custom 32bit command 'C' with associated 32bit data
--go
    continues normal execution of the downloaded bootloader
 options =
--configfile filename
    indicates the configuration file used with the following commands:
    --create, --format_all
--bct filename
    indicates the file containing the BCT
--sbk 0x00000000 00000000 00000000 00000000
    indicates the secure boot key for the target device
--bl filename
    downloads and runs the bootloader specified by filename
--odmdata N
   sets 32bit customer data into a field in the BCT, either hex or
   decimal
--diskimgopt N
    sets 32bit data required for disk image convertion tool
--format_all
    formats all existing partitions on the target device using the config
        file,
    including partitions and the bct
--setbootdevtype S
    sets the boot device type fuse value for the device name.
```

```
      allowed device name string mentioned below:
          emmc, nand_x8, nand_x16, nor, spi
  --setbootdevconfig N
      sets the boot device config fuse value either hex or decimal
  --verifypart N
       verifies data for partition id = N specified. N=-1
       indicates all partitions
       Intended to be used with --create command only.
  --setbct
      updates the chip specific settings of the BCT in mass storage to
      the bct supplied,used with --create, should not be with --read,and
      --format(delete)_all,format(delete)_partition,--download, and--read
  --sync
      issues force sync commad
  --rawdeviceread S N filename
      reads back N sectors starting from sector S into filename
  --rawdevicewrite S N filename
      writes back N sectors from filename to device starting from sector S
```

APPENDICE C

Pjproject - modifiche al codice

Indice

Per poter effettuare la compilazione della libreria e delle applicazioni in ambiente Android tramite la versione *trunk* di *Pjproject*, è stato necessario effettuare una patch al sorgente.

Si consiglia di eseguire prima il comando:

```
patch --dry-run -p1 -i PATCH
```

per verificare l'effettiva conformità della patch al sorgente senza applicarla, e quindi ripetere lo stesso comando omettendo il flag `--dry-run`. Le modifiche minori attinenti al processo di configurazione del crosscompilatore e dell'aggiunta degli oggetti binari è stata riportata nella Sezione B.2 a pagina 118.

C.1. Gestione dei file audio

C.1.1. Lettura degli header del file WAVE. Qui di seguito riporto il codice che ho effettuato per la lettura dei primi header dei file WAVE.

Listato C.1. Lettura header WAVE

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>
#include <assert.h>
#include <string.h>

struct WAVE {
        char ChunkID[4];
```

```
        char ChunkSize[4];
        char Format[4];
        char SubChunk1ID[4];
        char SubChunk1Size[4];
        char AudioFormat[2];
        char NumChannels[2];
        char SampleRate[4];
        char ByteRate[4];
        char BlockAlign[2];
        char BitsPerSample[2];
        char SubChunk2ID[4];
        char SubChunk2Size[4];
        char* DATA;
};

struct Wave {
        char ChunkID[4];
        unsigned int ChunkSize;
        char  Format[4];
        unsigned int SubChunk1ID;
        unsigned int SubChunk1Size;
        unsigned short AudioFormat;
        unsigned short NumChannels;
        unsigned int SampleRate;
        unsigned int ByteRate;
        unsigned short BlockAlign;
        unsigned short BitsPerSample;
        unsigned int SubChunk2ID;
        unsigned int SubChunk2Size;
        char* DATA;
};

unsigned short int char2(char* w) {
   unsigned short int word;
   assert(w);
   strncpy((char*)&word, w, 2);
   return word;
}

unsigned int char4(char* w) {
   unsigned int word;
   assert(w);
   strncpy((char*)&word, w, 4);
   return word;
}

#define mstrncpy(where,from,field,size) strncpy((where)->field,(from)->field,
    size)
#define cpy2(where,from,field) (where)->field = char2((from)->field);
```

```
#define cpy4(where,from,field) (where)->field = char4((from)->field);

void convertType(struct Wave* dest, struct WAVE* from) {
        assert(dest);
        assert(from);

        mstrncpy(dest,from,ChunkID,4);
        mstrncpy(dest,from,Format,4);

        cpy2(dest,from,AudioFormat);
        cpy2(dest,from,NumChannels);
        cpy2(dest,from,BlockAlign);
        cpy2(dest,from,BitsPerSample);

        cpy4(dest,from,ChunkSize);
        cpy4(dest,from,SubChunk1ID);
        cpy4(dest,from,SubChunk1Size);
        cpy4(dest,from,SampleRate);
        cpy4(dest,from,ByteRate);
        cpy4(dest,from,SubChunk2ID);
        cpy4(dest,from,SubChunk2Size);

}

#define printL(len,chars) printf("%.*s", len, chars);


void printW(struct Wave* w) {
        printf("ChunkID: "); printL(4,w->ChunkID); printf("\n");
        printf("ChunkSize: "); printf("%u\n",w->ChunkSize);
        printf("Format: "); printL(4,w->Format); printf("\n");
        printf("SubChunk1ID: "); printf("%u\n",w->SubChunk1ID);
        printf("SubChunk1Size: "); printf("%u\n",w->SubChunk1Size);
        printf("AudioFormat: ");  printf("%i\n",w->AudioFormat);
        printf("NumChannels: "); printf("%i\n",w->NumChannels);
        printf("SampleRate: "); printf("%u\n",w->SampleRate);
        printf("ByteRate: "); printf("%u\n",w->ByteRate);
        printf("BlockAlign: "); printf("%i\n",w->BlockAlign);
        printf("BitsPerSample: "); printf("%i\n",w->BitsPerSample);
        printf("SubChunk2ID: "); printf("%u\n",w->SubChunk2ID);
        printf("SubChunk2Size: "); printf("%u\n",w->SubChunk2Size);
}

#define EXAMPLE_WAV "au.wav"
#define FAIL 0
#define GAIN 1

int getWavSettings(char* filename, struct Wave* wav) {
    int fd = 0;
```

```
    struct WAVE file;

    if ((fd = open(filename,O_RDONLY))==0) {
            perror ("open");
            return FAIL;
    }
    if (read(fd,(void*)&file,sizeof(file))<=0) {
                        perror ("read");
            return FAIL;
    }
    if (close(fd)<0) {
                    perror ("close");
            return FAIL;
    }

        convertType(wav, &file);
        return GAIN;
}

int main(int argc, char* argv[]) {

        int fd = 0;
        struct Wave wav;

        if (argc<=1) {
                printf("Usage: %s file.wav\n", argv[0]);
                return 1;
        }

        if (getWavSettings(argv[1],&wav)) printW(&wav);


        return 0;

}
```

C.1.2. `conference.c`. Oltre alle altre modifiche minori che ho già avuto modo di illustrare, riporto qui di seguito il codice completo delle funzioni che sono state da me modificate all'interno del file `conference.c`. Le parti da me modificate sono marcate da `jb09`.

Listato C.2. Modifiche a conference.c

```
/* $Id: conference.c 3664 2011-07-19 03:42:28Z nanang $ */
/*
 * Copyright (C) 2008-2011 Teluu Inc. (http://www.teluu.com)
 * Copyright (C) 2003-2008 Benny Prijono <benny@prijono.org>
 *
 * This program is free software; you can redistribute it and/or modify
```

```
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 */
//Omissis
/*
 * DON'T GET CONFUSED WITH TX/RX!!
 *
 * TX and RX directions are always viewed from the conference bridge's point
 * of view, and NOT from the port's point of view. So TX means the bridge
 * is transmitting to the port, RX means the bridge is receiving from the
 * port.
 */


/**
 * This is a port connected to conference bridge.
 */
struct conf_port
{
    pj_str_t             name;          /**< Port name.                     */
    pjmedia_port        *port;          /**< get_frame() and put_frame()    */
    pjmedia_port_op      rx_setting;    /**< Can we receive from this port  */
    pjmedia_port_op      tx_setting;    /**< Can we transmit to this port   */
    unsigned             listener_cnt;  /**< Number of listeners.           */
    SLOT_TYPE           *listener_slots;/**< Array of listeners.            */
    unsigned             transmitter_cnt;/**<Number of transmitters.        */

    /* Shortcut for port info. */
    unsigned             clock_rate;    /**< Port's clock rate.             */
    unsigned             samples_per_frame; /**< Port's samples per frame.  */
    unsigned             channel_count; /**< Port's channel count.          */
    unsigned             bytes_per_sample; /**< jb09: particular
        bytes_per_sample           */

    /* Calculated signal levels: */
    unsigned             tx_level;      /**< Last tx level to this port.    */
    unsigned             rx_level;      /**< Last rx level from this port.  */

    /* The normalized signal level adjustment.
```

```
     * A value of 128 (NORMAL_LEVEL) means there's no adjustment.
     */
    unsigned             tx_adj_level;  /**< Adjustment for TX.             */
    unsigned             rx_adj_level;  /**< Adjustment for RX.             */

    /* Resample, for converting clock rate, if they're different. */
    pjmedia_resample    *rx_resample;
    pjmedia_resample    *tx_resample;

    /* RX buffer is temporary buffer to be used when there is mismatch
     * between port's sample rate or ptime with conference's sample rate
     * or ptime. The buffer is used for sampling rate conversion AND/OR to
     * buffer the samples until there are enough samples to fulfill a
     * complete frame to be processed by the bridge.
     *
     * When both sample rate AND ptime of the port match the conference
     * settings, this buffer will not be created.
     *
     * This buffer contains samples at port's clock rate.
     * The size of this buffer is the sum between port's samples per frame
     * and bridge's samples per frame.
     */
    pj_int16_t          *rx_buf;        /**< The RX buffer.                 */
    unsigned             rx_buf_cap;    /**< Max size, in samples           */
    unsigned             rx_buf_count;  /**< # of samples in the buf.       */

    /* Mix buf is a temporary buffer used to mix all signal received
     * by this port from all other ports. The mixed signal will be
     * automatically adjusted to the appropriate level whenever
     * there is possibility of clipping.
     *
     * This buffer contains samples at bridge's clock rate.
     * The size of this buffer is equal to samples per frame of the bridge.
     */

    int                  mix_adj;       /**< Adjustment level for mix_buf.  */
    int                  last_mix_adj;  /**< Last adjustment level.         */
    pj_int32_t          *mix_buf;       /**< Total sum of signal.           */

    /* Tx buffer is a temporary buffer to be used when there's mismatch
     * between port's clock rate or ptime with conference's sample rate
     * or ptime. This buffer is used as the source of the sampling rate
     * conversion AND/OR to buffer the samples until there are enough
     * samples to fulfill a complete frame to be transmitted to the port.
     *
     * When both sample rate and ptime of the port match the bridge's
     * settings, this buffer will not be created.
     *
     * This buffer contains samples at port's clock rate.
```

```
     * The size of this buffer is the sum between port's samples per frame
     * and bridge's samples per frame.
     */
    pj_int16_t          *tx_buf;        /**< Tx buffer.                     */
    unsigned             tx_buf_cap;    /**< Max size, in samples.          */
    unsigned             tx_buf_count;  /**< # of samples in the buffer.    */

    /* When the port is not receiving signal from any other ports (e.g. when
     * no other ports is transmitting to this port), the bridge periodically
     * transmit NULL frame to the port to keep the port "alive" (for example,
     * a stream port needs this heart-beat to periodically transmit silence
     * frame to keep NAT binding alive).
     *
     * This NULL frame should be sent to the port at the port's ptime rate.
     * So if the port's ptime is greater than the bridge's ptime, the bridge
     * needs to delay the NULL frame until it's the right time to do so.
     *
     * This variable keeps track of how many pending NULL samples are being
     * "held" for this port. Once this value reaches samples_per_frame
     * value of the port, a NULL frame is sent. The samples value on this
     * variable is clocked at the port's clock rate.
     */
    unsigned             tx_heart_beat;

    /* Delay buffer is a special buffer for sound device port (port 0, master
     * port) and other passive ports (sound device port is also passive port).
     *
     * We need the delay buffer because we can not expect the mic and speaker
     * thread to run equally after one another. In most systems, each thread
     * will run multiple times before the other thread gains execution time.
     * For example, in my system, mic thread is called three times, then
     * speaker thread is called three times, and so on. This we call burst.
     *
     * There is also possibility of drift, unbalanced rate between put_frame
     * and get_frame operation, in passive ports. If drift happens, snd_buf
     * needs to be expanded or shrinked.
     *
     * Burst and drift are handled by delay buffer.
     */
    pjmedia_delay_buf   *delay_buf;
};


/*
 * Conference bridge.
 */
struct pjmedia_conf
{
    unsigned             options;      /**< Bitmask options.               */
```

```
    unsigned              max_ports;    /**< Maximum ports.                 */
    unsigned              port_cnt;     /**< Current number of ports.       */
    unsigned              connect_cnt;  /**< Total number of connections    */
    pjmedia_snd_port     *snd_dev_port; /**< Sound device port.             */
    pjmedia_port         *master_port;  /**< Port zero's port.              */
    char                  master_name_buf[80]; /**< Port0 name buffer.      */
    pj_mutex_t           *mutex;        /**< Conference mutex.              */
    struct conf_port    **ports;        /**< Array of ports.                */
    unsigned              clock_rate;   /**< Sampling rate.                 */
    unsigned              channel_count;/**< Number of channels (1=mono).   */
    unsigned              samples_per_frame;    /**< Samples per frame.     */
    unsigned              bits_per_sample;      /**< Bits per sample.       */
};


/* Prototypes */
static pj_status_t put_frame(pjmedia_port *this_port,
                             pjmedia_frame *frame);
static pj_status_t get_frame(pjmedia_port *this_port,
                             pjmedia_frame *frame);
static pj_status_t get_frame_pasv(pjmedia_port *this_port,
                                  pjmedia_frame *frame);
static pj_status_t destroy_port(pjmedia_port *this_port);
static pj_status_t destroy_port_pasv(pjmedia_port *this_port);


/*
 * Create port.
 */
static pj_status_t create_conf_port( pj_pool_t *pool,
                                     pjmedia_conf *conf,
                                     pjmedia_port *port,
                                     const pj_str_t *name,
                                     struct conf_port **p_conf_port)
{
    struct conf_port *conf_port;
    pj_status_t status;

    /* Create port. */
    conf_port = PJ_POOL_ZALLOC_T(pool, struct conf_port);
    PJ_ASSERT_RETURN(conf_port, PJ_ENOMEM);

    /* Set name */
    pj_strdup_with_null(pool, &conf_port->name, name);

    /* Default has tx and rx enabled. */
    conf_port->rx_setting = PJMEDIA_PORT_ENABLE;
    conf_port->tx_setting = PJMEDIA_PORT_ENABLE;
```

```
    /* Default level adjustment is 128 (which means no adjustment) */
    conf_port->tx_adj_level = NORMAL_LEVEL;
    conf_port->rx_adj_level = NORMAL_LEVEL;

    /* Create transmit flag array */
    conf_port->listener_slots = (SLOT_TYPE*)
                                pj_pool_zalloc(pool,
                                               conf->max_ports * sizeof(SLOT_TYPE));
    PJ_ASSERT_RETURN(conf_port->listener_slots, PJ_ENOMEM);

    /* Save some port's infos, for convenience. */
    if (port) {
        pjmedia_audio_format_detail *afd;

        afd = pjmedia_format_get_audio_format_detail(&port->info.fmt, 1);
        conf_port->port = port;
        conf_port->clock_rate = afd->clock_rate;
        conf_port->samples_per_frame = PJMEDIA_AFD_SPF(afd);
        conf_port->channel_count = afd->channel_count;
        conf_port->bytes_per_sample = (afd->bits_per_sample / 8); //jb09
    } else {
        conf_port->port = NULL;
        conf_port->clock_rate = conf->clock_rate;
        conf_port->samples_per_frame = conf->samples_per_frame;
        conf_port->channel_count = conf->channel_count;
        conf_port->bytes_per_sample = BYTES_PER_SAMPLE;         //jb09
    }

    /* If port's clock rate is different than conference's clock rate,
     * create a resample sessions.
     */
    if (conf_port->clock_rate != conf->clock_rate) {

        pj_bool_t high_quality;
        pj_bool_t large_filter;

        high_quality = ((conf->options & PJMEDIA_CONF_USE_LINEAR)==0);
        large_filter = ((conf->options & PJMEDIA_CONF_SMALL_FILTER)==0);

        /* Create resample for rx buffer. */
        status = pjmedia_resample_create( pool,
                                          high_quality,
                                          large_filter,
                                          conf->channel_count,
                                          conf_port->clock_rate,/* Rate in */
                                          conf->clock_rate, /* Rate out */
                                          conf->samples_per_frame *
                                            conf_port->clock_rate /
                                            conf->clock_rate,
```

```
                                        &conf_port->rx_resample);
        if (status != PJ_SUCCESS)
            return status;


        /* Create resample for tx buffer. */
        status = pjmedia_resample_create(pool,
                                         high_quality,
                                         large_filter,
                                         conf->channel_count,
                                         conf->clock_rate,  /* Rate in */
                                         conf_port->clock_rate, /* Rate out */
                                         conf->samples_per_frame,
                                         &conf_port->tx_resample);
        if (status != PJ_SUCCESS)
            return status;
    }

    /*
     * Initialize rx and tx buffer, only when port's samples per frame or
     * port's clock rate or channel number is different then the conference
     * bridge settings.
     */
    if (conf_port->clock_rate != conf->clock_rate ||
        conf_port->channel_count != conf->channel_count ||
        conf_port->samples_per_frame != conf->samples_per_frame)
    {
        unsigned port_ptime, conf_ptime, buff_ptime, dbld = 1;

        port_ptime = conf_port->samples_per_frame / conf_port->channel_count *
            1000 / conf_port->clock_rate;
        conf_ptime = conf->samples_per_frame / conf->channel_count *
            1000 / conf->clock_rate;

PJ_LOG(4,(THIS_FILE, "conf_port->samples_per_frame %u, conf_port->channel_count
     %u, conf_port->clock_rate %u, port_ptime %u, conf_ptime %u\n",(unsigned int
    )conf_port->samples_per_frame, (unsigned int)conf_port->channel_count,
    conf_port->clock_rate, port_ptime , conf_ptime));//jb09

        /* Calculate the size (in ptime) for the port buffer according to
         * this formula:
         *   - if either ptime is an exact multiple of the other, then use
         *     the larger ptime (e.g. 20ms and 40ms, use 40ms).
         *   - if not, then the ptime is sum of both ptimes (e.g. 20ms
         *     and 30ms, use 50ms)
         */

         //jb09: START
        if (port_ptime > conf_ptime) {
```

```
            buff_ptime = port_ptime;

           /* if (port_ptime % conf_ptime)
                buff_ptime += conf_ptime;
        else PJ_LOG(4,(THIS_FILE, "no inner if "));*/

        } else {
            buff_ptime = conf_ptime;
            /* if (conf_ptime % port_ptime)
                buff_ptime += port_ptime;
                else PJ_LOG(4,(THIS_FILE, "no inner if")); */
        }

        if (port_ptime % conf_ptime) dbld = 2;
            buff_ptime *= dbld;
            PJ_LOG(4,(THIS_FILE, "case 1: port_ptime %u, conf_ptime %u,
                buff_ptime %u\n", port_ptime, conf_ptime, buff_ptime));
//jb09:END
        /* Create RX buffer. */
        //conf_port->rx_buf_cap = (unsigned)(conf_port->samples_per_frame +
        //                                   conf->samples_per_frame *
        //                                   conf_port->clock_rate * 1.0 /
        //                                   conf->clock_rate + 0.5);

#define MYMAX(a,b) ((a)>(b) ? a : b)

        conf_port->rx_buf_cap = 2 * conf_port->samples_per_frame * dbld;//jb09
        if (conf_port->channel_count > conf->channel_count)
            conf_port->rx_buf_cap *= conf_port->channel_count;
        else
            conf_port->rx_buf_cap *= conf->channel_count;

        conf_port->rx_buf_count = 0;
        conf_port->rx_buf = (pj_int16_t*)
                            pj_pool_alloc(pool, conf_port->rx_buf_cap *
                                                sizeof(conf_port->rx_buf[0]));
        PJ_ASSERT_RETURN(conf_port->rx_buf, PJ_ENOMEM);

        /* Create TX buffer. */
        conf_port->tx_buf_cap = conf_port->rx_buf_cap;
        conf_port->tx_buf_count = 0;
        conf_port->tx_buf = (pj_int16_t*)
                            pj_pool_alloc(pool, conf_port->tx_buf_cap *
                                                sizeof(conf_port->tx_buf[0]));
        PJ_ASSERT_RETURN(conf_port->tx_buf, PJ_ENOMEM);
    }


    /* Create mix buffer. */
```

```
    conf_port->mix_buf = (pj_int32_t*)
                         pj_pool_zalloc(pool, conf->samples_per_frame *
                                              sizeof(conf_port->mix_buf[0]));
    PJ_ASSERT_RETURN(conf_port->mix_buf, PJ_ENOMEM);
    conf_port->last_mix_adj = NORMAL_LEVEL;


    /* Done */
    *p_conf_port = conf_port;
    return PJ_SUCCESS;
}


//Omissis


/*
 * Read from port.
 */
static pj_status_t read_port( pjmedia_conf *conf,
                              struct conf_port *cport, pj_int16_t *frame,
                              pj_size_t count, pjmedia_frame_type *type )
{

    pj_assert(count == conf->samples_per_frame);


    TRACE_((THIS_FILE, "read_port %.*s: count=%d",
                       (int)cport->name.slen, cport->name.ptr,
                       count));

    /*
     * If port's samples per frame and sampling rate and channel count
     * matche conference bridge's settings, get the frame directly from
     * the port.
     */
    if (cport->rx_buf_cap == 0) {
        pjmedia_frame f;
        pj_status_t status;

        f.buf = frame;
        f.size = count * cport->bytes_per_sample; //XXX: VALORE DI DEFAULT
            BYTES_PER_SAMPLE (jb09)

        TRACE_((THIS_FILE, "  get_frame %.*s: count=%d",
                   (int)cport->name.slen, cport->name.ptr,
                   count));

        status = pjmedia_port_get_frame(cport->port, &f);
```

```
        *type = f.type;

        return status;

    } else {
        unsigned samples_req;

        /* Initialize frame type */
        if (cport->rx_buf_count == 0) {
            *type = PJMEDIA_FRAME_TYPE_NONE;
        } else {
            /* we got some samples in the buffer */
            *type = PJMEDIA_FRAME_TYPE_AUDIO;
        }

        /*
         * If we don't have enough samples in rx_buf, read from the port
         * first. Remember that rx_buf may be in different clock rate and
         * channel count!
         */

        samples_req = (unsigned) (count * 1.0 *
                      cport->clock_rate / conf->clock_rate + 0.5);


        while (cport->rx_buf_count < samples_req) {




            pjmedia_frame f;
            pj_status_t status;
            int spf = cport->samples_per_frame; //(jb09)
            int tmpsize = cport->samples_per_frame * cport->bytes_per_sample;
               //(jb09)

        //(jb09) start
        if (samples_req > cport->rx_buf_cap) {
        PJ_LOG(4,(THIS_FILE, "WARNING: ASSERTING. bufcount = %u, bufcap = %u,
            tmpsize=%u, spf=%u, count=%u, cpclock=%u, cclock=%u\n",
                                    cport->rx_buf_count, cport->rx_buf_cap,
                                       tmpsize,spf,count,cport->clock_rate /
                                       conf->clock_rate));

                                    pj_assert(samples_req <= cport->rx_buf_cap
                                       );
        }
```

```
            f.buf = cport->rx_buf + cport->rx_buf_count;
            if (cport->rx_buf_count + tmpsize > cport->rx_buf_cap) {
                PJ_LOG(4,(THIS_FILE, "WARNING: EXCEEDING. bufcount = %u, bufcap
                    = %u, tmpsize=%u, spf=%u\n",
                                    cport->rx_buf_count, cport->rx_buf_cap,
                                        tmpsize,spf));
                                    //BREAK;
            }//(jb09) end

            f.size = tmpsize;

            TRACE_((THIS_FILE, "  get_frame, count=%d",
                       cport->samples_per_frame));

            status = pjmedia_port_get_frame(cport->port, &f);

            if (status != PJ_SUCCESS) {
                /* Fatal error! */
                return status;
            }

            if (f.type != PJMEDIA_FRAME_TYPE_AUDIO) {
                TRACE_((THIS_FILE, "  get_frame returned non-audio"));
                pjmedia_zero_samples( cport->rx_buf + cport->rx_buf_count,
                                      spf);//(jb09)
            } else {
                /* We've got at least one frame */
                *type = PJMEDIA_FRAME_TYPE_AUDIO;
            }

            /* Adjust channels */
            if (cport->channel_count != conf->channel_count) {
                if (cport->channel_count == 1) {
                    pjmedia_convert_channel_1ton((pj_int16_t*)f.buf,
                                                 (const pj_int16_t*)f.buf,
                                                 conf->channel_count,
                                                 spf,//(jb09)
                                                 0);
                    cport->rx_buf_count += (spf * conf->channel_count);//(jb09)
                } else { /* conf->channel_count == 1 */
                    pjmedia_convert_channel_nto1((pj_int16_t*)f.buf,
                                                 (const pj_int16_t*)f.buf,
                                                 cport->channel_count,
                                                 spf, //(jb09)
                                                 PJMEDIA_STEREO_MIX, 0);
                    cport->rx_buf_count += (spf / cport->channel_count);//(jb09
                        )
                }
```

```
            } else {
                cport->rx_buf_count += cport->samples_per_frame;
            }

            TRACE_((THIS_FILE, "  rx buffer size is now %d",
                    cport->rx_buf_count));

        PJ_LOG(4,(THIS_FILE, "bufcount = %u, bufcap = %u, tmpsize=%u, spf=%u\n"
            ,
                                    cport->rx_buf_count, cport->rx_buf_cap,
                                        tmpsize,spf));

            pj_assert(cport->rx_buf_count <= cport->rx_buf_cap);
        }

        /*
         * If port's clock_rate is different, resample.
         * Otherwise just copy.
         */
        if (cport->clock_rate != conf->clock_rate) {

            unsigned src_count;

            TRACE_((THIS_FILE, "  resample, input count=%d",
                    pjmedia_resample_get_input_size(cport->rx_resample)));

            pjmedia_resample_run( cport->rx_resample,cport->rx_buf, frame);

            src_count = (unsigned)(count * 1.0 * cport->clock_rate /
                                   conf->clock_rate + 0.5);
            cport->rx_buf_count -= src_count;
            if (cport->rx_buf_count) {
                pjmedia_move_samples(cport->rx_buf, cport->rx_buf+src_count,
                                    cport->rx_buf_count);
            }

            TRACE_((THIS_FILE, "  rx buffer size is now %d",
                    cport->rx_buf_count));

        } else {

            pjmedia_copy_samples(frame, cport->rx_buf, count);
            cport->rx_buf_count -= count;
            if (cport->rx_buf_count) {
                pjmedia_move_samples(cport->rx_buf, cport->rx_buf+count,
                                    cport->rx_buf_count);
            }
        }
    }
```

```
    return PJ_SUCCESS;
}
//Omissis
```

Riferimenti bibliografici

Bibliografia

[Bia10] Alessio Bianchi. "Gestione della mobilità verticale su base applicazione: progetto e realizzazione per la piattaforma Android". Tesi di Laurea Magistrale. Università degli Studi di Roma "Tor Vergata", 2010 (cit. alle pp. 10, 15, 22).

[Car11] Massimo Carli. *Android™ 3: Guida per lo sviluppatore.* Apogeo, 2011 (cit. alle pp. 33, 36).

[Lia99] Sheng Liang. *The Java Native Interface. Programmer's Guide and Specification*. Addison-Wesley, 1999 (cit. a p. 27).

[NK11] Mohammad Nauman e Sohail Khan. "Design and implementation of a fine-grained resource usage model for the android platform". In: *Int. Arab J. Inf. Technol.* 8.4 (2011), pp. 440–448 (cit. a p. 105).

[Ong+09] Machigar Ongtang et al. "Semantically Rich Application-Centric Security in Android". In: ACSAC '09 (2009), pp. 340–349. DOI: 10.1109/ACSAC.2009.39. URL: http://dx.doi.org/10.1109/ACSAC.2009.39 (cit. alle pp. 9, 105).

[Sch11] Thorsten Schreiber. "Android Binder. Android Interprocess Communication". Seminarthesis. Ruhr-Universität Bochum, 2011 (cit. alle pp. 30, 36, 37, 58).

[Sil09] Vladimir Silva. *Pro Android Games*. Apress, 2009 (cit. a p. 8).

[Yag11] Karim Yaghmour. *Embedded Android*. O'Reilly, 2011 (cit. alle pp. 14, 15, 34, 35, 75).

Sitografia

[Gar07] M. Gargenta. *Remixing Android*. Marakana. 2007. URL: http://marakana.com/s/remixing_android,1044/index.html (cit. alle pp. 16, 22, 30).

[GJG12] A. Gargenta, K. Jones e M. Gargenta. *Marakana - Android Internals*. Marakana. 2012. URL: http://marakana.com/static/courseware/android/internals/index.html (cit. alle pp. 31, 35, 54).

[Hua12] Jim Huang. *Android IPC Mechanism*. 0xlab. 2012. URL: http://0xlab.org/~jserv/android-binder-ipc.pdf (cit. a p. 41).

[Jol09] Chen Jollen. *Android Porting*. Il documento è scritto prevalentemente in Cinese, ed è quindi stato a volte necessario utilizzare un traduttore automatizzato. In particolare il sottotitolo recita "Fondamenti di Crosscompilazione ed Analisi Strategica". CTimes. 2009. URL: `http://140.116.72.245/ESW/download/s2.pdf` (cit. a p. 71).

[PP10] C. Papathanasiou e N. J. Percoco. *This is not the droid you're looking for...* Trustwave. 2010. URL: `http://plagiarism.repec.org/negrila/negrila2.pdf` (cit. a p. 13).

[Pri06] Benny Prijono. *PJSIP Developer's Guide*. PJSIP. 2006. URL: `http://www.pjproject.net` (cit. a p. 80).

[saf10] safransx@gmail.com. *IPC Analysis on Android*. In particolare ho fatto riferimento alla versione fornita da `http://blogimg.chinaunix.net/blog/upfile2/081203105044.pdf`. blog.csdn.net. 2010. URL: `http://blog.csdn.net/safrans/article/details/6272652` (cit. alle pp. 41, 45).