UNIVERSITÀ DEGLI STUDI DI PADOVA

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea in Ingegneria Informatica

ESPERIENZE DI ELABORAZIONE AUDIO SU PIATTAFORMA ANDROID

Relatore:

Prof. Carlo Fantozzi

Relazione di Tirocinio di:

Francesco Pedron

Anno accademico 2011/2012

Sommario

Questa relazione descrive il tirocinio svolto presso Bloop Srl per valutare la possibilità di sviluppare applicazioni di elaborazione audio su piattaforma Android.

In particolare sono state valutate le prestazioni per l'elaborazione audio in real-time, la latenza minima ottenibile per l'acquisizione e l'esecuzione di audio, la possibilità di integrare libpd in Android e la maturità del toolkit openFrameworks.

La capacità di elaborare audio in real-time è risultata al più sufficiente a causa delle scarse prestazioni. E' stata valutata sviluppando un'applicazione di filtraggio audio e scrivendo parti dell'applicazione utilizzando il codice nativo.

La latenza è stata misurata sia con prove sperimentali che ricercando informazioni su Internet. Si è concluso che la piattaforma non presenta valori di latenza adeguati per applicazioni audio real-time.

Libpd è una libreria che permette di integrare Pure Data in un'applicazione; la possibilità di integrare tale libreria in Android è assicurata dal progetto PD for Android.

OpenFrameworks è un framework multi-piattaforma che permette di sviluppare applicazioni in C++ ed eseguirle su diverse piattaforme. Il supporto ad Android si è rivelato scadente.

Indice

Capitolo 1: Introduzione

1.1 Android

1.1.1 Architettura

Android è una piattaforma per sistemi embedded composta da una versione modificata del kernel Linux, un insieme di librerie native, l'Android Runtime, l'Application Framework e un insieme di applicazioni fondamentali.

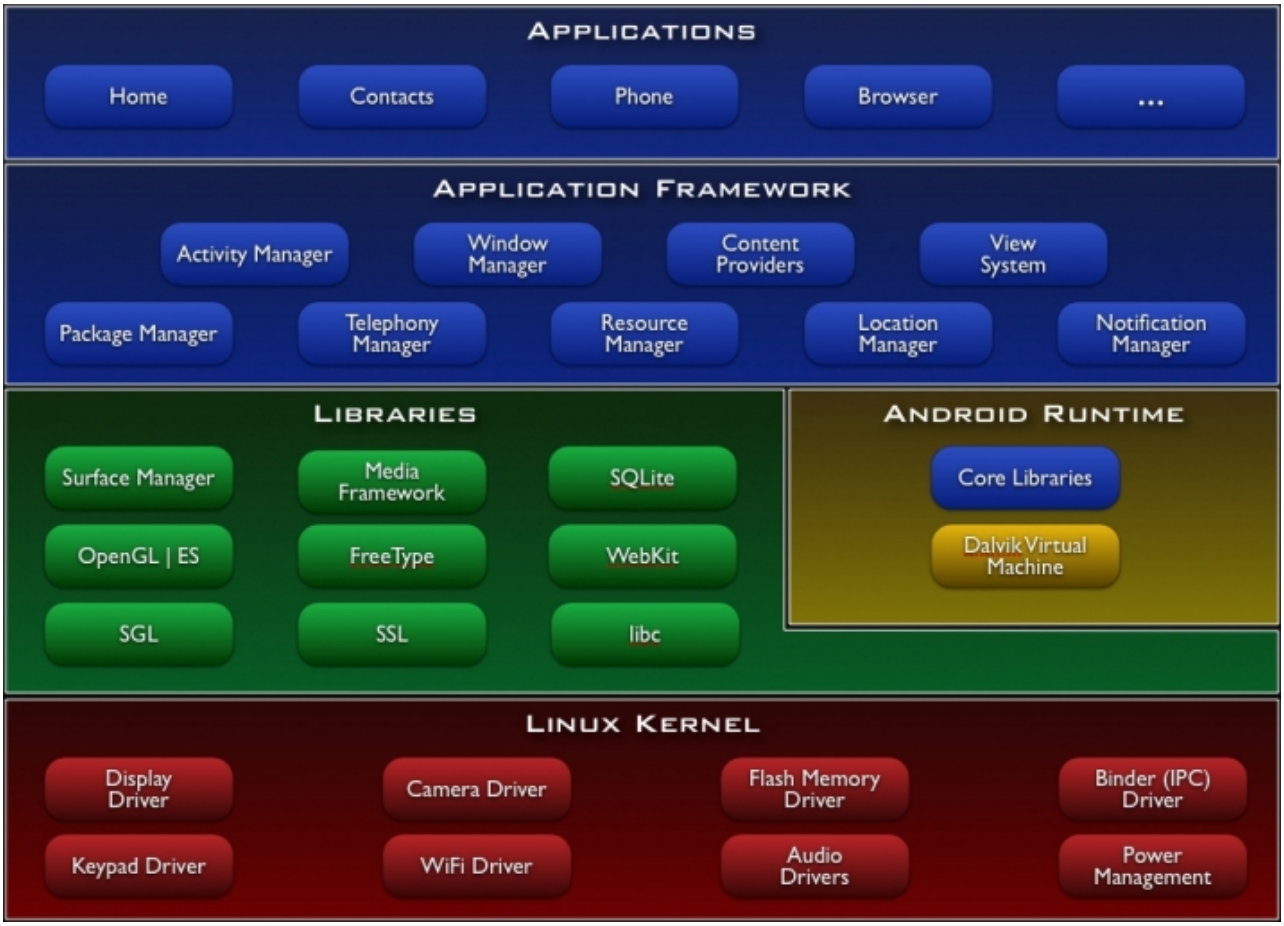

Illustrazione 1: Architettura di Android

Il kernel Linux, basato sulla versione 2.6, include alcuni servizi di sistema fondamentali come il controllo della memoria, il controllo dei processi, lo stack di rete e fornisce l'HAL

(Hardware Abstraction Layer) che permette ai produttori OEM di sviluppare driver per i nuovi dispositivi.

Le librerie native sono scritte in C e C++ e forniscono un grande potenziale alla piattaforma.

- **Libreria di sistema C (libc)**: è un'implementazione derivata da BSD della libreria che fornisce le chiamate di sistema e altre utilità come *open*, *malloc*, *printf*, ecc.
- **SSL**: permette di effettuare comunicazioni sicure in rete agendo a livello di trasporto.
- **SGL**: è una libreria di grafica 2D.
- **WebKit**: è un motore per browser web open source.
- **FreeType**: è un motore di rendering per font bitmap e vettoriali.
- **OpenGL|ES**: è un motore grafico 3D che può utilizzare l'accelerazione hardware se disponibile.
- **SQLite**: è una libreria open source che implementa un motore per database relazionali.
- **Media Framework**: permette di eseguire e registrare media utilizzando molti codec tra i quali MPEG4, H.264, MP3, AAC, AMR, JPG, PNG, …
- **Surface Manager**: si occupa di gestire l'accesso al sottosistema del display e di comporre la grafica 2D e 3D proveniente dalle varie applicazioni.

Le applicazioni per Android vengono sviluppate principalmente in Java ma affidandosi ad una libreria custom non compatibile né con Java SE né con Java ME. I sorgenti sono compilati in Java bytecode e tradotti nel formato proprietario DEX. Tale formato ottimizza la dimensione del bytecode fondendo più file .class in un unico file .dex all'interno del quale saranno quindi incluse più classi Java. In questo modo firme dei metodi, stringhe e costanti ripetute possono essere incluse una sola volta per risparmiare spazio. Tali ottimizzazioni rendono i file .dex più piccoli dei relativi .jar compressi, anche se i file .dex non sono compressi. Il formato DEX è indipendente dalla piattaforma e viene eseguito dalla Dalvik Virtual Machine, una Java Virtual Machine proprietaria che è ottimizzata per sistemi con limitate risorse di calcolo e memoria. Dall'introduzione di Android 2.2, alla macchina virtuale Dalvik è stato aggiunto un compilatore Just-in-Time che migliora le prestazioni. [Bornstein, 2008]

In Android ogni applicazione viene eseguita in un processo separato, da un utente separato e su una copia della macchina virtuale Dalvik separata: in questo modo ogni applicazione può accedere

solamente ai propri dati e i malfunzionamenti non si propagano tra le applicazioni.

La piattaforma fornisce allo sviluppatore il Native Development Kit (NDK) che dà la possibilità di sviluppare parti di applicazioni usando linguaggi di programmazione nativi come C e C++; ciò permette di riutilizzare il codice esistente e di migliorare le performance. L'adozione dell'NDK dev'essere valutata dallo sviluppatore per bilanciare costi e benefici; usare il codice nativo non comporta necessariamente un miglioramento delle prestazioni, ma incrementa sicuramente la complessità dell'applicazione. Se ne può valutare l'utilizzo quando devono essere eseguite

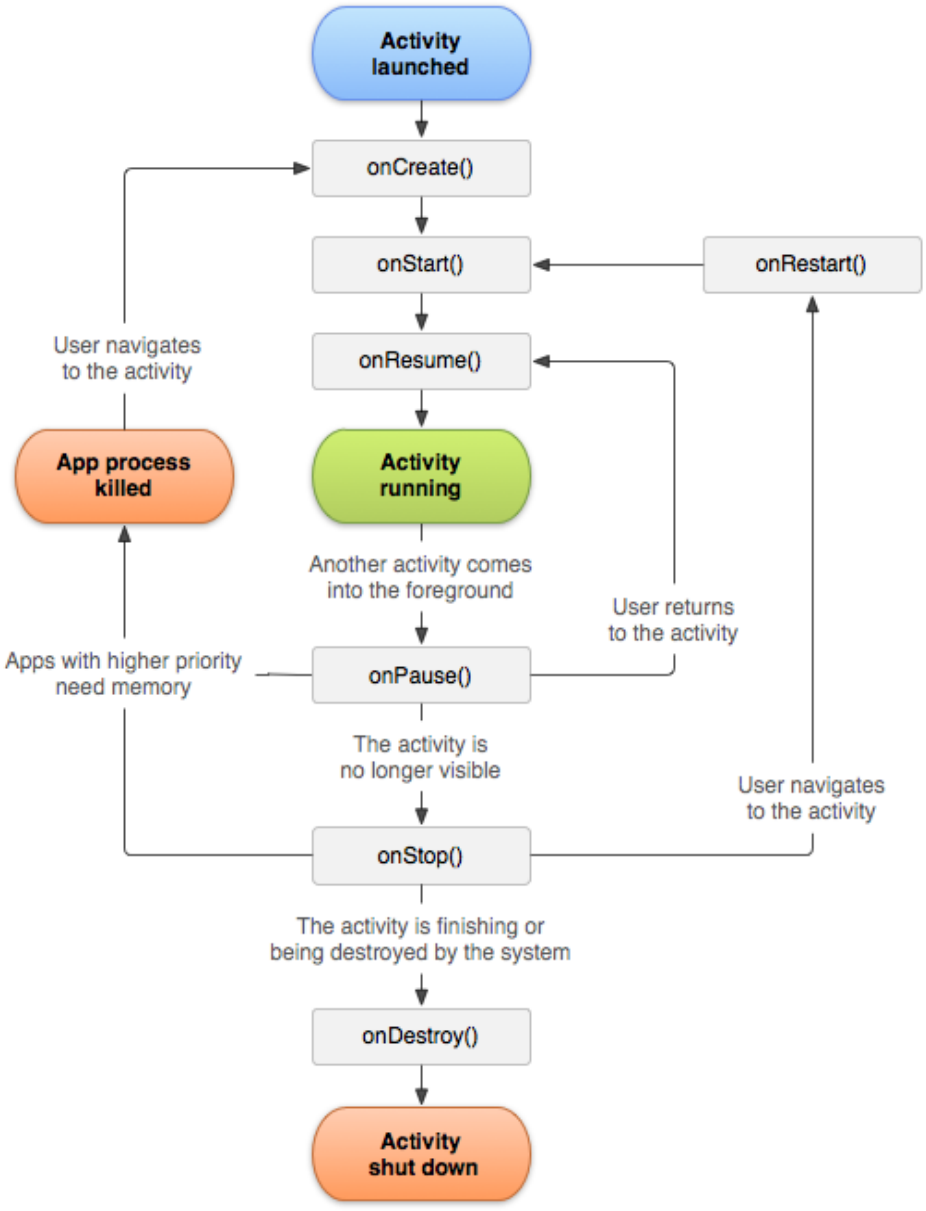

Illustrazione 2: Ciclo di vita di un'activity

operazioni di calcolo intensivo che richiedano comunque un modesto consumo di memoria, buona consuetudine per ogni applicazione in ambiente embedded sia essa scritta in Java o in codice nativo. L'NDK può essere utilizzato scrivendo applicazioni in Java con l'Android Framework e implementando parti specifiche dell'applicazione in codice nativo utilizzando la JNI (Java Native Inferface) oppure scrivendo direttamente activity native, cioè implementando i metodi di callback che gestiscono il loro ciclo di vita. Tratteremo l'NDK in maggiore dettaglio nel paragrafo 1.1.4.

L'Application Framework è scritta in Java ed è l'insieme di strumenti sui quali tutte le applicazioni (siano esse di sistema o di terze parti) si basano; l'API che fornisce è quindi comune e nega l'esistenza di applicazioni "privilegiate". Per esempio un'applicazione di messaggistica di terze parti può sostituire l'omologa applicazione di sistema semplicemente dichiarando di svolgere tale funzione; sarà poi l'utente o il sistema operativo a scegliere l'applicazione da usare per mezzo del "Intent Resolution". L'API contenuta nell'Application Framework è composto da:

- **View System**: fornisce i componenti per disegnare l'interfaccia grafica dell'applicazione.
- **Window Manager**: crea le finestre e gestisce gli eventi dell'interfaccia utente.
- **Activity Manager**: gestisce il ciclo di vita delle applicazioni e organizza le finestre in uno stack per permettere all'utente di muoversi tra le applicazioni.
- **Content Providers**: permettono di memorizzare e recuperare informazioni rendendole inoltre condivisibili tra varie applicazioni.
- **Package Manager**: mantiene traccia delle applicazioni installate nel dispositivo Android e delle loro funzionalità.
- **Telephony Manager**: fornisce l'accesso per la periferica al servizio telefonico.
- **Resource Manager**: gestisce l'accesso di ogni applicazione alle sue risorse come testo, immagini, layout, … .
- **Location Manager**: fornisce l'accesso ai sistemi di geolocalizzazione.
- **Notification Manager**: notifica all'utente gli eventi che si verificano in background.

1.1.2 Componenti fondamentali

I componenti fondamentali per lo sviluppo di un'applicazione sono: **Activity**, **Broadcast Receiver**, **Service**, **Content Provider** e **Intent**.

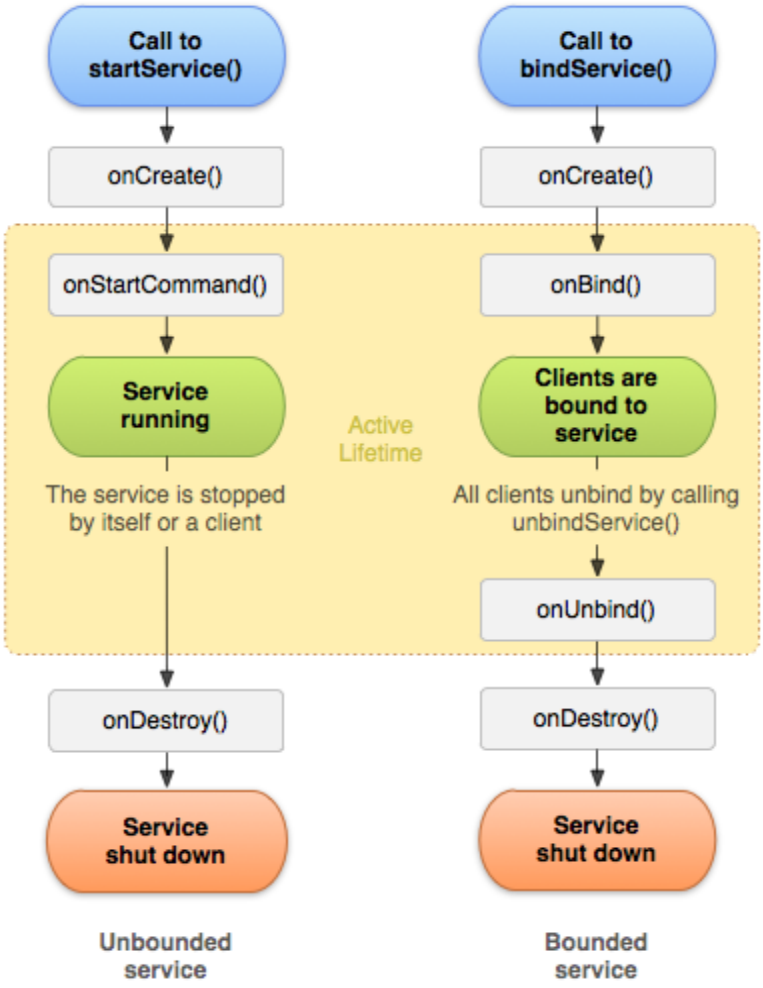

Illustrazione 3: Ciclo di vita di un Service

L'**Activity** è, solitamente, una singola finestra dell'applicazione che svolge un compito specifico. Ad esempio in una semplice galleria di foto potremmo avere due Activity: una visualizza l'elenco delle foto con le loro miniature e l'altra visualizza una foto a pieno schermo e permette di effettuare lo zoom. Il passaggio tra un'activity e l'altra è gestito per mezzo di Intent. E' stato fatto uso dell'Activity nella realizzazione delle applicazioni di filtraggio del paragrafo 3.2 e anche libpd (paragrafo 2.2) ne fa uso.

Il **Broadcast Receiver** è un componente che può rispondere a degli eventi di sistema (chiamata in arrivo, modifica dello stato della rete, …) ed eseguire del codice. Esso non dispone di un'interfaccia grafica.

Il **Service** è un componente che può eseguire delle operazioni in background che devono proseguire anche se l'utente dovesse passare ad un'altra applicazione. Esso non dispone di un'interfaccia grafica ma un'activity lo può controllare effettuando un binding e inviandogli dei messaggi. Un esempio tipico è un player audio: lo sviluppatore implementa un service tramite cui l'utente può far partire una playlist per mezzo di un'activity e, anche quando l'utente passa ad un'altra applicazione, la musica continua ad essere riprodotta; quando l'utente ritorna al player per passare alla prossima canzone, l'activity è stata istruita dallo sviluppatore per effettuare il binding sul service che le permette di inviargli il messaggio di cambiare traccia.
Lo studio del Service è stato necessario per la realizzazione dell'applicazione di filtraggio del paragrafo 3.2; inoltre vi si può ricorrere anche in libpd (paragrafo 2.2) per eseguire l'audio in background.

Il **Content Provider** permette ad un'applicazione di condividere dati in modo che possano essere riutilizzati da altre applicazioni. Un esempio è il database dei contatti che può essere utilizzato da ogni applicazione che ne faccia richiesta.

L'**Intent** è un componente della piattaforma che permette di attivare Activity, Service e Broadcast Receiver. E' una richiesta astratta di effettuare un'azione esplicita che consiste nell'attivazione di uno specifico componente (ad esempio una specifica classe), o implicita che fornisce indicazioni al sistema per selezionare il componente adatto tra quelli disponibili o propone la scelta all'utente (per mezzo dell'intent resolution).

1.1.2.1 Ciclo di vita dell'Activity

Durante la sua vita un'activity può cambiare di stato; ciò avviene in seguito ad azioni dell'utente o in base alla necessità del sistema di liberare risorse.

Un'activity si può trovare in quattro stati:

- **Attiva** (running): è visibile e riceve input dall'utente.
- **In Pausa** (onPause): è parzialmente visibile ma non riceve input dall'utente perché un'altra activity ha il focus. L'activity si trova ancora in memoria e mantiene le informazioni di stato delle variabili ma può essere rimossa qualora le risorse di sistema dovessero scarseggiare.

- **Fermata** (onStop): non è visibile, è ancora in memoria ma il sistema la può rimuovere non appena le risorse dovessero essere richieste da un altro processo.
- **Distrutta** (onDestroy): l'activity non è più utilizzabile e la memoria ad essa associata può essere recuperata dal garbage collector.

Le transizioni di stato non possono essere evitate e devono essere gestite dallo sviluppatore implementando alcuni metodi di callback durante i quali assumere le azioni necessarie. Quando un'activity è attiva l'unico metodo di callback garantito prima di un passaggio di stato è *onPause();* questo metodo va quindi usato per salvare lo stato dell'applicazione. *onStop()* e *onDestroy()* sono invece opzionali.

1.1.2.2 Ciclo di vita del Service

Un service può assumere due forme che possono coesistere implementando gli opportuni metodi di callback:

- **Started**: un service si dice started quando è avviato da un componente di un'applicazione per mezzo di *startService()* e rimane attivo anche quando il componente che l'ha avviato viene distrutto; il service provvederà a terminarsi autonomamente quando avrà concluso il suo compito. Il metodo di callback da implementare è *onStartCommand()* e permette ai componenti di avviare il service; il service si terminerà alla conclusione del suo lavoro mediante i metodi *stopSelf()* o *stopService()*.
- **Bound**: un service si dice bound quando un componente di un'applicazione effettua un bind su di esso (cioè vi si lega); in questo modo i componenti possono interagire con il service inviando richieste e ricevendo risultati. Un service bound verrà terminato quando tutti i componenti che vi si sono legati effettueranno l'unbind (cioè si slegano). Il metodo di callback da implementare è *onBind()* e permette ai componenti di legarsi al service.

Sia che il service sia di tipo started che di tipo bound, alla sua creazione viene richiamato il metodo *onCreate()* che può essere utilizzato per effettuare il setup.

Un service può comunque essere avviato con *startService()* ed essere quindi di tipo started ma subire poi il bind di uno o più componenti; a questo punto le due tipologie convivono e il service rimane attivo fino a quando non ci sono più componenti legati e non sono stati chiamati sia *stopService()* che *stopSelf()*.

E' importante ricordare che un service viene eseguito nel thread principale del processo che lo ospita. Operazioni di calcolo intensivo vanno eseguite all'interno di un nuovo thread per evitare blocchi o rallentamenti dell'interfaccia utente.

[Fantozzi, 2011] [Android Guide]

1.1.3 Gestione dell'audio

La piattaforma fornisce allo sviluppatore alcuni strumenti che permettono di registrare ed eseguire l'audio accedendovi con diverse complessità.

Al più alto livello la piattaforma propone un framework multimediale che fornisce supporto all'esecuzione e alla registrazione di audio (e video) nei formati più comuni. Ciò permette allo sviluppatore di integrare facilmente l'audio nelle sue applicazioni impedendo però l'accesso di basso livello (ovvero ai campioni). Le classi che fanno parte del framework sono *MediaPlayer* per l'esecuzione e *MediaRecorder* per la registrazione.

L'accesso a più basso livello si ottiene mediante le classi *AudioTrack* e *AudioRecord* che permettono di accedere all'audio nella forma di flussi PCM.

```
// Get minimum buffer size
audioBufferInBytes = AudioTrack.getMinBufferSize(sampleRate,
outputChannels, outputEncoding);
// Initialize audio
androidPlayer = new AudioTrack(AudioManager.STREAM_MUSIC, sampleRate,
outputChannels, outputEncoding, audioBufferInBytes,
AudioTrack.MODE_STREAM);
androidPlayer.play();
```

Frammento di Codice 1: Inizializzazione di AudioTrack in modalità stream

1.1.3.1 AudioTrack

La classe *AudioTrack* si occupa della riproduzione di audio e può operare con due modalità: stream e static.

Nella modalità stream l'applicazione invia uno stream continuo di dati a un'istanza di AudioTrack utilizzando il metodo *write();* in questo modo può eseguire suoni che sarebbero troppo grandi per risiedere in memoria a causa della loro durata o del loro bit rate (per banda o dimensione dei campioni), suoni che vengono ricevuti in tempo reale (per esempio dal microfono, dalla Rete, …) e suoni generati mentre l'audio viene eseguito.

Nella modalità static, invece, l'applicazione deve scrivere tutto il flusso di dati con un'unica chiamata al metodo *write()*; per questo è utilizzabile solo quando si dispone di un flusso audio che possa risiedere completamente in memoria. Tale modalità è particolarmente indicata quando si vuole raggiungere la minima latenza possibile e per brevi suoni che devono essere eseguiti spesso.

```
// Initialize audio
playerInStaticMode = new AudioTrack(AudioManager.STREAM_MUSIC,
SAMPLE_RATE, OUTPUT_CHANNELS, OUTPUT_ENCODING, audioData.length *
BYTES_PER_SAMPLE, AudioTrack.MODE_STATIC);
```

Frammento di Codice 2: Inizializzazione di AudioTrack in modalità static

Il metodo *write()* è un metodo bloccante e ritorna solo quando i dati sono stati trasferiti dallo strato Java allo strato nativo e accodati per l'esecuzione.

```
androidPlayer.write(audioData, 0, audioData.length);
```

Frammento di Codice 3: Esecuzione di audio in modalità stream

```
playerInStaticMode.write(audioData, 0, audioData.length);
playerInStaticMode.play();
```

Frammento di Codice 4: Esecuzione di audio in modalità static

Al momento della creazione di un'istanza di AudioTrack occorre specificare la modalità di funzionamento, la frequenza di campionamento, il numero di canali e la codifica dei campioni. Questi parametri permettono di calcolare anche la dimensione minima del buffer associato all'istanza dell'oggetto. Per calcolarla è sufficiente fornire tali parametri al metodo *getMinBufferSize()* di AudioTrack. In modalità static il buffer impostato rappresenta la massima dimensione dello stream audio che può essere eseguito, mentre nella modalità stream esso

rappresenta la massima quantità di dati che possono essere inseriti nel buffer di volta in volta e in questo caso l'applicazione si deve preoccupare di evitare lo svuotamento del buffer.

1.1.3.2 AudioRecord

La classe *AudioRecord* gestisce le risorse hardware della piattaforma per registrare audio.

Come per AudioTrack, anche per AudioRecord si definisce la dimensione del buffer di cattura; la dimensione minima è fornita dal metodo *getMinBufferSize()* di AudioRecord al quale vengono fornite le informazioni sul flusso audio da registrare (frequenza di campionamento, numero di canali e codifica dei campioni). L'applicazione è responsabile di estrarre i dati da un'istanza di AudioRecord usando il metodo *read()* e la dimensione del buffer impostata definisce per quanto tempo l'istanza può registrare prima di andare in “over-run” e perdere i vecchi campioni.

```
bufferSize = AudioRecord.getMinBufferSize(44100,
AudioFormat.CHANNEL_IN_MONO, AudioFormat.ENCODING_PCM_16BIT);
AudioRecord recorder = new AudioRecord(MediaRecorder.AudioSource.MIC,
44100, AudioFormat.CHANNEL_IN_MONO, AudioFormat.ENCODING_PCM_16BIT,
bufferSize);
if (recording.getState() == AudioRecord.STATE_INITIALIZED)
      recording.startRecording();
```

Frammento di Codice 5: Inizializzazione di AudioRecord

[Android Guide]

1.1.4 Native Development Kit

Come anticipato nel paragrafo 1.1.1, l'NDK permette di implementare parte dell'applicazione usando linguaggi di programmazione nativi come C e C++. L'NDK fornisce:

- un insieme di strumenti e di file di build per generare librerie in codice nativo a partire da sorgenti C e C++
- un sistema per integrare tali librerie scritte in codice nativo nel file .apk
- un insieme di headers e librerie di sistema native
- la documentazione, gli esempi e le guide

Il supporto delle headers native è garantito in tutte le versioni di Android successive alla 1.5 (e da Android 2.3 per quanto riguarda le activities native). Le headers messe a disposizione sono:

- libc (libreria C)
- libm (libreria di funzioni matematiche)
- interfaccia JNI (Java Native Interface)
- libz (compressione Zlib)
- liblog (logging per Android)
- OpenGL ES 1.1 e OpenGL ES 2.0 (librerie grafiche 3D)
- libjnigraphics (accesso a buffer di pixel) (a partire da Android 2.2)
- un insieme minimale di headers per il supporto a C++
- librerie audio native OpenSL ES (a partire da Android 2.3)
- APIs per sviluppare applicazioni in codice nativo per Android

Durante lo svolgimento del tirocinio si è fatto largo uso della JNI che sarà quindi trattata più estesamente nel paragrafo successivo.

1.1.4.1 Java Native Interface

La Java Native Interface (JNI) è il componente dell'NDK che permette al codice Java di interagire con codice e librerie scritti in C e C++. Allo scopo di testare l'utilizzo della JNI (che sarà impiegata intensivamente nel Capitolo 3) sono state sviluppate delle semplici applicazioni Android con metodi sviluppati in codice nativo C e C++. In questo paragrafo sono riportati frammenti di codice di tali applicazioni.

Supponiamo di dover sviluppare parte di un'applicazione in codice nativo: all'interno della classe Java *example.java* avremo sia metodi scritti in Java che metodi dichiarati come nativi che saranno implementati nel file *test-jni.c* scritto in C. In questo primo esempio realizzeremo un “Hello World!” usando la JNI. Nel Frammento di Codice 6 vediamo parte del sorgente Java dove dichiariamo il metodo nativo *helloFromJNI()* che è caratterizzato dalla parola chiave *native* per indicare che il metodo sarà implementato nativamente. Dopo aver dichiarato il metodo, usiamo il metodo *System.loadLibrary()* per indicare la libreria che lo contiene; in questo caso *test-jni*.

```
public native String helloFromJNI();
static {
      System.loadLibrary("test-jni");
}
```

Frammento di Codice 6: Un metodo dichiarato come nativo in example.java

Nel file C realizziamo un metodo che corrisponda a *helloFromJNI()*; tale metodo dovrà avere un nome del tipo *Java_package_class_method()* che nel nostro caso diventa *Java_unipd_dei_JniTest_Example_hellFromJni()*; *"package"* indica il nome del package Java dove i punti vengono sostituiti con dei trattini bassi *"_"*, *"class"* indica il nome della classe Java e *"method"* indica il nome del metodo nativo.

Oltre ai parametri del metodo dichiarato nativo in Java, la sua implementazione C prevede sempre la presenza di altri due parametri standard. Il primo è un puntatore all'interfaccia di JNI (*JNIEnv*) che può essere usato dai metodi nativi per accedere alle funzioni della JNI e alle strutture dati della Java Virtual Machine. La natura del secondo parametro varia qualora il metodo sia statico o relativo ad un'istanza: per i metodi statici l'oggetto è un riferimento alla classe nella quale il metodo è definito mentre per i metodi relativi ad un'istanza l'oggetto è un riferimento all'istanza stessa.

Java e C hanno la necessità di scambiarsi informazioni sotto forma di parametri di ingresso e di uscita dei metodi; a tale scopo esiste una mappatura tra tipi di dato in Java e in codice nativo con un comportamento diverso tra tipi Java primitivi (come int, float, char, …) e tipi Java referenziati (come classi, istanze e array). I tipi primitivi hanno un omologo in codice nativo; ad esempio *int* è mappato come *jint*, *float* come *jfloat*. I tipi referenziati sono passati ai metodi nativi come delle *opaque references*, puntatori C che referenziano strutture interne alla Java Virtual Machine; esse sono mantenute nascoste al programmatore che vi può accedere usando apposite funzioni fornite dalla JNI per mezzo di *JNIEnv*. Tutti questi puntatori sono del tipo *jobject*, ma esistono dei sottotipi come *jstring* o *jobjectArray* che facilitano il lavoro dello sviluppatore.
Un esempio di uso dei tipi referenziati è presente nel Frammento di Codice 7: il metodo *hellFromJNI()* ritorna un oggetto Java di tipo *String* che è mappato in C nel tipo referenziato *jstring*; per trasformare la sequenza di caratteri "Hello, this is a message from C using JNI!" in un

oggetto di tipo *String* si deve usare quindi il metodo *NewStringUTF()* di *JNIEnv* (l'operazione inversa può essere ottenuta invece usando il metodo *GetStringUTFChars()*).

```
#include <string.h>
#include <jni.h>

jstring Java_unipd_dei_jniTest_Example_helloFromJNI(JNIEnv* env, jobject
obj)
{
    return (*env)->NewStringUTF(env, "Hello, this is a message from C
using JNI!");
}
```

Frammento di Codice 7: Il contenuto di test-jni.c

Nel paragrafo 3.2 avremo la necessità di passare array di tipi primitivi (in particolare di *short*) tra Java e codice nativo. Qui di seguito vengono presentate le due modalità di implementazione utilizzabili con la JNI. Per testare queste funzionalità è stata sviluppata una classe nativa C++ denominata *test-jni-cpp.cpp* che può essere importata come libreria usando il comando *System.loadLibrary("test-jni-cpp")* esattamente come per la libreria C usata precedentemente.
La JNI fornisce anche la possibilità di accedere ad array di oggetti (tipi referenziati o matrici di tipi primitivi) ma questo esula dallo scopo di questa presentazione.
La prima modalità accede all'array copiandone tutti gli elementi ed è quindi utilizzabile per accedere ad array di piccole dimensioni; tale modalità fa uso del metodo *Get<primitive-*

```
JNIEXPORT jint JNICALL Java_unipd_dei_jniTest_Example_sumAnArray(JNIEnv*
env, jobject obj, jintArray arr)
{
	jint length=env->GetArrayLength(arr);
	jint buf[length];
	jint i, sum=0;
	env->GetIntArrayRegion(arr,0,length,buf);
	for(i=0;i<length;i++)
	{
		sum+=buf[i];
	}
	return sum;
}
```

Frammento di Codice 8: Somma degli elementi di un array usando JNI e copiando l'array

type>ArrayRegion() di *JNIEnv* che copia l'array Java in un array nativo definito localmente; esistono metodi per copiare array di ogni tipo primitivo. Un esempio per l'uso di questa modalità è riportato nel Frammento di Codice 8.

La seconda modalità accede all'array per riferimento permettendo di accedere ad array di grandi dimensioni e di rendere permanenti eventuali modifiche dei loro elementi; tale modalità fa uso del metodo *Get<primitive-type>ArrayElements()* per ottenere un puntatore agli elementi dell'array e del metodo *Release<primitive-type>ArrayElements()* per rilasciare l'array. Gli elementi dell'array possono essere acceduti con un indice come indicato nel Frammento di Codice 9.

```
JNIEXPORT jint JNICALL
Java_unipd_dei_jniTest_Example_sumAnArrayWithPointer(JNIEnv* env, jobject
obj, jintArray arr)
{
	jint length=env->GetArrayLength(arr);
	jint *carr;
	jint i, sum=0;
	carr = env->GetIntArrayElements(arr,NULL);
	if(carr==NULL)
	{
		return 0; //Exception
	}
	for(i=0;i<length;i++)
	{
		sum+=carr[i];
	}
	env->ReleaseIntArrayElements(arr,carr,0);
	return sum;
}
```

Frammento di Codice 9: Somma degli elementi di un array usando JNI e accedendo all'array con un puntatore

Non tutte le implementazioni della Java Virtual Machine assicurano l'accesso per riferimento agli array attraverso la JNI; in tal caso l'array sarà copiato localmente in maniera trasparente allo sviluppatore. Lo sviluppatore può comunque controllare se l'array viene acceduto per riferimento o meno; questa prova è stata fatta anche sui due dispositivi usati durante il tirocinio (versioni Android 2.2 e 3.2) rilevando che, in entrambi i casi, gli array erano acceduti per riferimento. Tale prova può essere effettuata con la seguente riga di codice che scrive il risultato nel LogCat:

```
__android_log_write(ANDROID_LOG_DEBUG, "app_tag", isCopy?"true":"false");
```

Le due librerie presentate in questo paragrafo possono essere compilate modificando il file Android.mk nel seguente modo:

```
 LOCAL_PATH := $(call my-dir)
#Compile in C
include $(CLEAR_VARS)
LOCAL_MODULE    := test-jni
LOCAL_SRC_FILES := test-jni.c
include $(BUILD_SHARED_LIBRARY)
#Compile in C++
include $(CLEAR_VARS)
LOCAL_MODULE    := test-jni-cpp
LOCAL_SRC_FILES := test-jni-cpp.cpp
include $(BUILD_SHARED_LIBRARY)
```

Frammento di Codice 10: File Android.mk per la compilazione delle librerie native

[Android SDK - NDK] [Sheng Liang, 2002]

1.2 Scopo del lavoro di stage

Bloop srl è interessata a valutare la possibilità di sviluppare applicazioni di elaborazione audio su piattaforma Android. In particolare bisogna valutare le prestazioni per l'elaborazione audio in real-time, la latenza minima ottenibile per l'acquisizione e l'esecuzione di audio, la possibilità di integrare libpd e la maturità del toolkit openFrameworks.

Nei capitoli successivi sono presentate le applicazioni sperimentali sviluppate a tale scopo e gli strumenti utilizzati. In particolare nel capitolo 2 si trattano i tool sperimentati che non fanno parte della piattaforma Android come libpd e openFrameworks, mentre nel capitolo 3 sono presentate le esperienze condotte sulla latenza e le prestazioni per l'elaborazione audio senza l'ausilio di tool esterni alla piattaforma. In ogni capitolo dopo un'introduzione all'argomento vengono presentati il lavoro svolto e i risultati ottenuti.

Capitolo 2: Tool

In questo capitolo vengono presentati gli strumenti sperimentati per valutare la loro portabilità e maturità sulla piattaforma Android e i risultati ottenuti. Gli strumenti utilizzati sono libpd, una porting di Pure Data come libreria realizzata principalmente per gli ambienti mobile, e openFrameworks, un framework per lo sviluppo di applicazioni multi-piattaforma.

2.1 Pure Data

Pure Data è un ambiente di programmazione visuale per l'elaborazione in tempo reale di audio, video e grafica. Il core di PD, abbreviazione comunemente usata, è scritto e mantenuto da Miller Puckette. Il progetto è open source e beneficia del supporto di una vasta comunità. Nel corso del tirocinio si è fatto riferimento al tutorial di Johannes Kreidler "Programming Electronic Music in PD" [Kreidler, 2009] per quanto riguarda Pure Data.

In PD è possibile scrivere patch ed external per estendere le sue funzionalità; le patch permettono di programmare in maniera visuale collegando tra loro dei blocchi. Inoltre lo sviluppatore ha la possibilità di scrivere nuovi blocchi usando linguaggi di programmazione come C o C++, i nuovi blocchi prodotti sono gli external. Molte patch ed external sono resi disponibili dalla comunità e spesso sono inclusi nel pacchetto standard di Pure Data.

Illustrazione 4: Blocchi fondamentali in PD

PD usa 4 tipi di blocchi testuali: atoms, messages, objects, e comments. Gli atoms sono le unità fondamentali in PD e possono consistere di float (a 32 bit), symbol o puntatori a strutture dati. Unendo vari atoms possono essere composti messages che forniscono istruzioni agli objects.

Gli objects svolgono le funzioni più disparate, dalle classiche funzioni matematiche e logiche presenti nei comuni linguaggi di programmazione, a funzioni dedicate al DSP (Digital Signal Processing) che sono caratterizzate per convenzione da una tilde (~), ad esempio *osc~* produce un coseno alla frequenza indicata come argomento. I comments possono essere usati come commenti in qualsiasi parte della patch.

I blocchi dispongono di punti di ingresso (inlets) e di uscita (outlets) per i dati. L'ordine con cui i dati giungono agli inlets è importante. Una regola generale è che gli objects ricevono gli ingressi da destra a sinistra e non producono nulla in uscita fino a quando non ricevono qualcosa sull'inlet più a sinistra; si può quindi distinguere tra inlets freddi (detti “cold”) che non producono cambiamenti immediati e inlets caldi (detti “hot”) che causano cambiamenti visibili istantaneamente quando ricevono dati.

I blocchi possono essere uniti tra loro per mezzo di segmenti. In analogia con il mondo analogico, i dispositivi analogici sono simboleggiati dai blocchi, mentre i cavi che li collegano sono astratti come dei segmenti.

Illustrazione 5: Inlet e Outlets

Si può notare che i segmenti presenti nell'Illustrazione 4 sono diversi da quello presente nell'Illustrazione 5: i primi sono fini, mentre il secondo è più grosso, ciò è dovuto al fatto che segmenti fini trasportano segnali di controllo mentre segmenti grossi trasportano dati.

Durante il tirocinio lo studio di PD si è protratto esclusivamente per il tempo necessario a comprenderne il funzionamento per riuscire a scrivere semplici patch di prova e per poter analizzare le patch di esempio fornite con libpd e openFrameworks.

Non esiste una versione di Pure Data per sistemi embedded ma le sue grandi potenzialità possono essere sfruttate ovunque grazie a libpd.

2.2 libpd

Lo scopo di libpd è trasformare Pure Data in una libreria di sintesi audio integrabile in applicazioni desktop e mobile; in questo modo le grandi potenzialità di PD possono essere portate ovunque. Libpd si basa su PD Vanilla e non è quindi un fork di PD, ma è privo dell'interfaccia grafica di PD. Libpd permette di instaurare semplicemente una comunicazione tra PD e il codice in cui è integrato. Tale comunicazione permette di scambiare dati di controllo e MIDI (libpd supporta ogni tipo di messaggio esclusi i puntatori); il codice manda messaggi alla patch PD dove sono ricevuti da blocchi di tipo *r receiver_name* (ad esempio “r test” nell'Illustrazione 6) e può sottoscrivere la ricezione di messaggi che sono inviati dalla patch con blocchi del tipo *s sender_name* (ad esempio “s android” nell'Illustrazione 6).

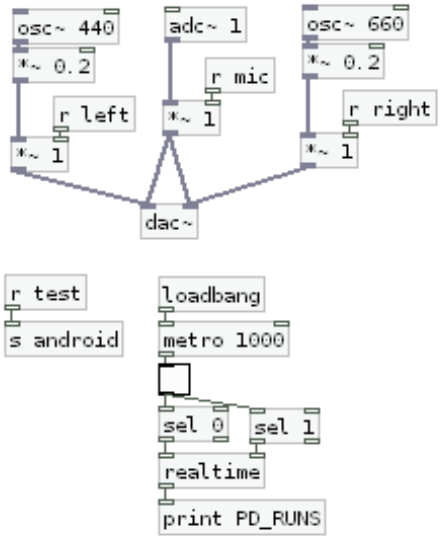

Illustrazione 6: Patch di prova fornita con PD for Android

Libpd include binder per Java, Objective-C e Python; essi espongono le funzionalità di libpd al linguaggio per il quale sono stati scritti, effettuano la conversione tra i tipi di dato in PD e nello specifico linguaggio, forniscono un'interfaccia orientata agli oggetti per le funzioni per il passaggio dei messaggi e sono thread-safe.

Libpd può essere portato su ogni piattaforma che permetta di eseguire codice nativo ed esiste un porting specifico per Android denominato “PD for Android” che sfrutta l'NDK di Android.

[libpd Wiki] [Brinkmann e altri, 2011]

2.2.1 PD for Android

Lo sviluppo di libpd iniziò come un porting di Pure Data su Android, solo successivamente è stato esteso ad altri linguaggi e piattaforme. Nel 2010, quando iniziò lo sviluppo, non era possibile accedere all'API audio Android da codice nativo (tale funzionalità è stata aggiunta nell'API di livello 9, corrispondente alla versione di Android 2.3), gli sviluppatori furono quindi obbligati a sviluppare un wrapper Java per Pure Data che fornisse un'API per l'elaborazione audio e il passaggio di messaggi; a questo scopo è stato realizzato PD for Android, un wrapper JNI che si occupa di interfacciare Java con libpd, che è invece scritto in C e si interfaccia con Pure Data.

La stratificazione creata con PD for Android e libpd ha portato naturalmente a scrivere inoltre un wrapper per Objective-C, denominato PD for iOS, che implementa un'interfaccia simile a quella di PD for Android e permette di utilizzare comodamente la libreria in iOS; iOS permette comunque di accedere direttamente a libpd usando la sua interfaccia C.

Successivamente verrà presentato esclusivamente il wrapper Java PD for Android.

2.2.1.1 Interagire con libpd in Java

Nel seguente paragrafo viene presentata esclusivamente l'API Java di libpd ma metodi simili esistono per Objective-C.

```
static int openPatch(File file) throws IOException;
static int openPatch(String path) throws IOException;
static void closePatch(int handle);
```

Frammento di Codice 11: Aprire e chiudere le patch con libpd

Per aprire un patch in libpd è sufficiente indicargli il percorso del file attraverso il metodo *openPatch* di *PdBase*, questo ritorna un intero che funge da riferimento alla patch aperta. Quando si vuole chiudere la patch è sufficiente passare questo riferimento al metodo *closePatch*.

```
static int sendBang(String receiver);
static int sendFloat(String receiver, float value);
static int sendSymbol(String receiver, String symbol);
static int sendList(String receiver, Object... list);
static int sendMessage(String receiver, String message, Object... list);
```

Frammento di Codice 12: Inviare messaggi a libpd

Per inviare messaggi a libpd si possono usare i metodi statici *sendBang*, *sendFloat*, *sendSymbol*, *sendList*, e *sendMessage* di *PdBase*. Tali metodi inviano un messaggio ad un symbol di tipo receive come visibile nell'Illustrazione 6. Questi metodi, come tutti quelli presenti in *PdBase*, non lanciano eccezioni ma ritornano un intero diverso da 0 in caso di errore, per esempio se non esiste il ricevitore indicato o se il messaggio fornito è del tipo sbagliato.

Per ricevere messaggi da PD il codice dell'applicazione deve implementare l'interfaccia di *PdReceiver* e registrane come ricevitore l'istanza di un oggetto per mezzo di *PdBase*. Vedremo più in dettaglio nell'analisi di *PdTest*.

Quando termina il lavoro con PD, l'applicazione deve prendersi cura di rilasciare le risorse mantenute da libpd usando il metodo *PdBase.release()*. Rilasciare le risorse non appena non sono più necessarie è sempre una buona abitudine, ma, in PD for Android, è indispensabile ricordarsi di farlo per ottenere un comportamento affidabile. Ciò è dovuto alla particolare gestione del ciclo di vita delle activity in Android e alla struttura di PD. Nel paragrafo 1.1.2.1 abbiamo visto che quando l'activity viene distrutta (*onDestroy*) essa non è più utilizzabile e la memoria può essere liberata dal garbage collector. Va ricordato però che può rimanere traccia di variabili statiche il cui valore sarà quindi mantenuto all'avvio successivo. PD, inoltre, è strutturato in modo da non permettere istanze multiple ma ne esiste esclusivamente una globale che mantiene molte variabili statiche. Questi due fattori comportano che se durante la distruzione dell'activity non si indica a PD di rilasciare le risorse esse saranno mantenute fino al prossimo avvio dell'activity, dove si reinizializzerà PD. Il problema principale è che una delle risorse mantenute è la patch e al secondo avvio di PD ci saranno due patch identiche aperte che, invece di produrre il loro suono in uscita, produrranno solo del rumore. Occorre dunque resettare lo stato di PD nel metodo *onDestroy* dell'activity come indicato nel Frammento di Codice 13.

```
@Override
public void onDestroy() {
    super.onDestroy();
    PdAudio.release();
    PdBase.release();
}
```

Frammento di Codice 13: Come rilasciare le risorse in PD for Android

2.2.1.2 Eseguire PD in un Service

Quando si sviluppa un'applicazione audio spesso si desidera che esso possa continuare ad essere eseguito anche quando passiamo ad un'altra activity o addirittura ad un'altra applicazione. PD for Android permette di sviluppare applicazioni con due modalità: la prima obbliga ad eseguire PD nello stesso thread dell'activity dal quale lo si avvia usando direttamente *PdAudio*, questa scelta lega la vita di PD a quella dell'activity e quindi relega la vita di PD al tempo in cui l'activity è visibile e impedisce di continuare ad eseguire audio quando si cambia activity o applicazione; la seconda modalità si ottiene mediante la classe *PdService* che fa uso dei service di Android (paragrafo 1.1.2.2), con questa modalità PD è gestito in un service che viene eseguito in background e rimane attivo anche quando si passa ad un'altra activity o applicazione e che può essere condiviso tra più activities. *PdService* permette quindi di ottenere il comportamento desiderato, vedremo come usarlo nell'analisi di *PdTest*.

2.2.1.3 Preparare l'ambiente di sviluppo

Prima di iniziare a sviluppare applicazioni con PD for Android occorre preparare l'ambiente di sviluppo. Quanto segue si intende testato per un ambiente Linux (ma si può applicare anche ad ambienti Microsoft o Apple); i prerequisiti sono:

- Eclipse (versione 3.7 o superiore)
- Android Software Development Kit (Android SDK)
- Android Development Tools (ADT)
- JDK (versione 1.6)
- Git

Il sito per gli sviluppatori Android spiega molto bene la procedura da seguire per avere un ambiente di sviluppo Android funzionante, vi si rimanda per le istruzioni dettagliate: http://developer.android.com/sdk/installing/index.html.
PD for Android richiede Java 1.6; tale impostazione va specificata nelle preferenze di Eclipse tramite il menu Window → Preferences → Java → Compiler e impostando “Compiler Compliance Level” su “1.6”.

Il primo passo è procurarsi una copia dei sorgenti di PD for Android dal repository git; è sufficiente aprire un terminale, spostarsi nella directory nella quale si intende scaricare PD for

Android ed eseguire i seguenti comandi che scaricano: libpd, PD for Android e alcuni esempi (incluso PdTest, l'applicazione che analizzeremo nel paragrafo 2.2.2).

```
$ git clone git://github.com/libpd/pd-for-android.git
$ cd pd-for-android
$ git submodule init
$ git submodule update
```

Il passo successivo è l'importazione di libpd e dei progetti di test in Eclipse: dopo aver aperto Eclipse selezioniamo File → Import... , quindi General → Existing Projects into Workspace; indichiamo come root directory quella nella quale abbiamo scaricato i sorgenti di libpd con git, quindi Finish (Illustrazione 7).

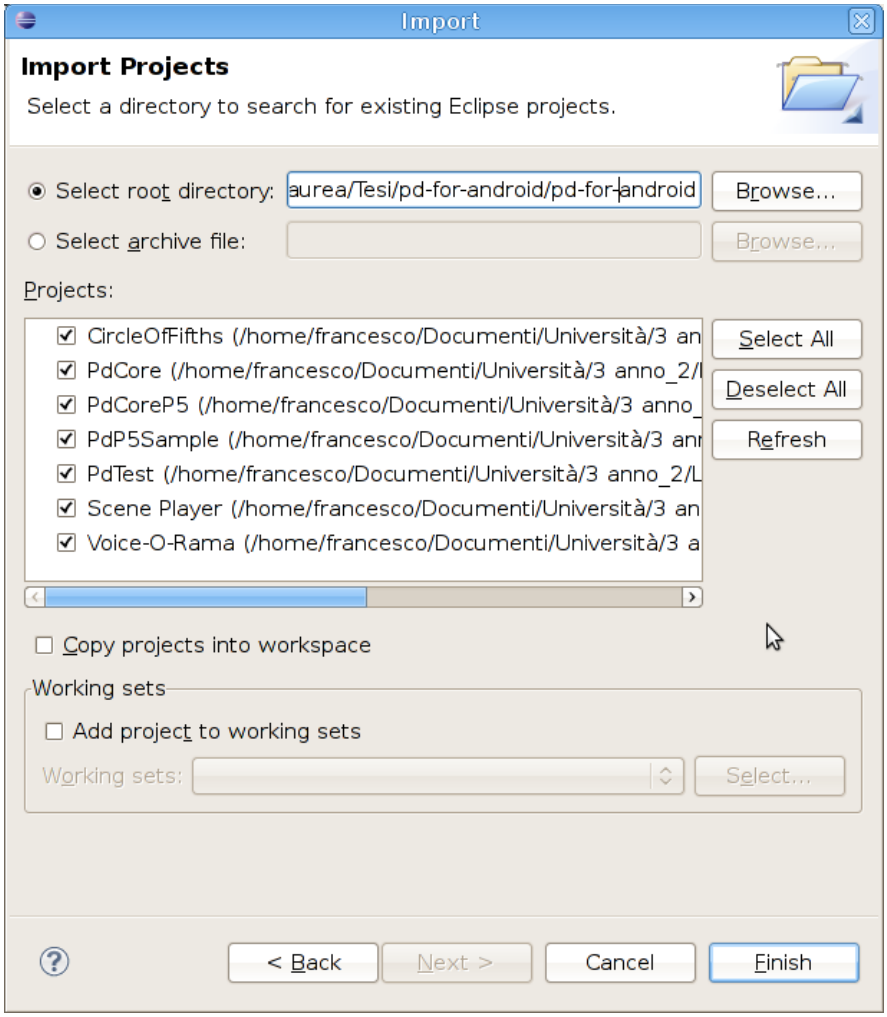

Illustrazione 7: Importazione di libpd, PD for Android e degli esempi

Il progetto più importante tra quelli importati è *PdCore*; è la libreria che include libpd, la parte che gestisce l'audio e alcune utilities. Questa libreria deve essere indicata come dipendenza per ogni nuovo progetto si voglia sviluppare usando PD for Android.

Tra i progetti importati troviamo anche *PdTest*, una semplice applicazione particolarmente adatta per capire come sviluppare applicazioni usando PD su piattaforma Android; analizzeremo questa applicazione passo passo nel paragrafo successivo.

L'emulatore Android fornito con l'SDK non può essere usato per testare le prestazioni dell'applicazione sviluppata, soprattutto per quanto riguarda l'audio, la massima frequenza di campionamento supportata è 8KHz, inadeguata per qualsiasi applicazione audio.

2.2.2 PdTest – Analisi dell'applicazione di prova di PD for Android

Durante lo svolgimento del tirocinio è stato necessario affrontare autonomamente il primo approccio a libpd (questo strumento è stato sperimentato nel periodo gennaio-febbraio 2012 e il libro di Peter Brinkmann “Making Musical Apps” [Brinkmann, 2012] è stato pubblicato soltanto nel febbraio 2012); a tale scopo è stata analizzata l'applicazione di esempio fornita con PD for Android *“PdTest”*; l'applicazione è stata riscritta cercando di comprenderne il funzionamento, individuando i passi fondamentali da seguire e cercando di migliorarne la leggibilità, qui viene riproposta quell'analisi.

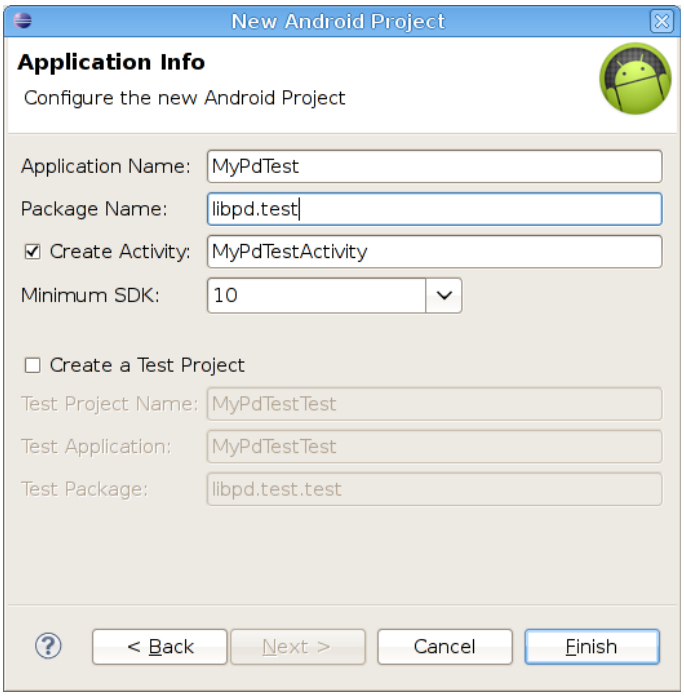

Illustrazione 8: Creazione del progetto Android in Eclipse

PdTest è una semplice applicazione che sfrutta tutte le funzionalità di PD for Android, apre una patch PD (quella visibile nell'Illustrazione 6), esegue audio e lo cattura dal microfono, permette di interagire con PD, invia e riceve messaggi da PD e usa un Service per eseguire PD.

2.2.2.1 Impostazione del progetto Eclipse

Il primo passo è la creazione del progetto Android in Eclipse al quale diamo un nome, *MyPdTest* in questo caso, e scegliamo una versione dell'SDK di Android; API Level 10 (Android 2.3.3) o superiore è richiesto da PdCore (Illustrazione 8).

Come indicato nel paragrafo 2.2.1.3 il progetto *PdCore* deve essere indicato come libreria per ogni altro progetto che usi libpd, occorre quindi selezionare Project → Properties → Android → nella sezione Library scegliere Add → selezionare PdCore (Illustrazione 9).

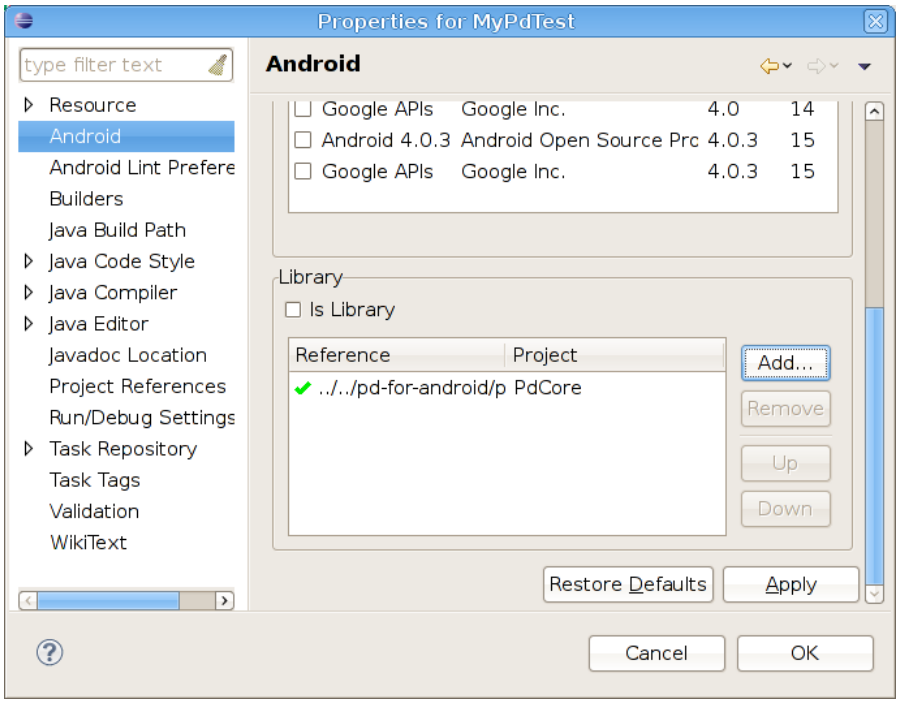

Illustrazione 9: Inserimento di PdCore come Libreria per MyPdTest

PdTest accede al microfono e necessita quindi del permesso Android di registrare audio. Tale permesso può essere aggiunto per via grafica oppure testuale agendo sul file AndroidManifest.xml; scegliamo la via testuale e aggiungiamo al file, tra i tag *manifest,* la seguente riga:

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

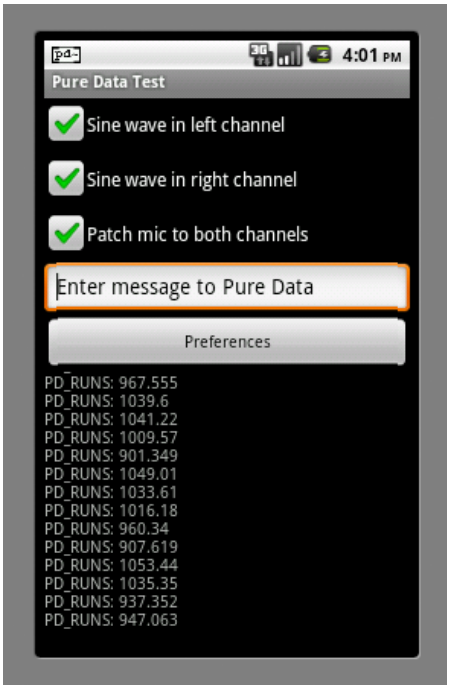

Illustrazione 10: Main activity di PdTest

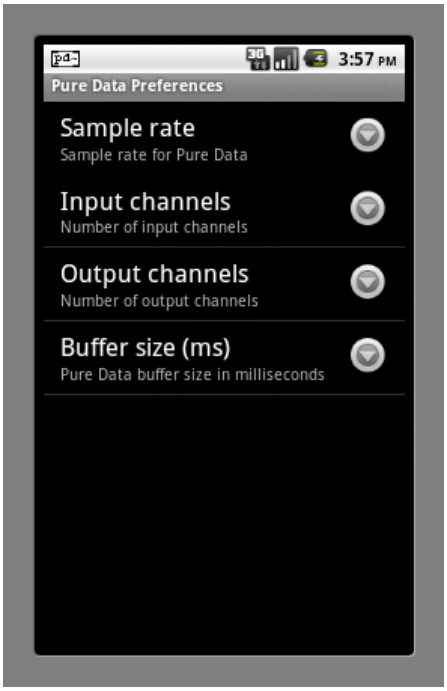

Illustrazione 11: Preferenze dell'applicazione PdTest

2.2.2.2 Interfaccia grafica e preferenze

Come si può vedere nell'Illustrazione 10 l'interfaccia grafica di PdTest è composta da:

- 3 checkbox che permettono di inviare dei messaggi alla patch PD indicando di eseguire una sinusoide sul canale sinistro o destro e di ripetere i suoni catturati dal microfono del dispositivo
- 1 textbox tramite la quale l'utente può inviare messaggi a PD
- 1 pulsante per accedere alle preferenze dell'applicazione che permettono di impostare PD
- 1 area nella quale vengono visualizzati i messaggi provenienti da PD

L'impostazione dell'interfaccia grafica della main activity di questa applicazione è effettuata tramite file XML, come si usa solitamente in Android, ed esula dallo scopo di questa presentazione. Per le checkbox e la textbox viene impostata l'activity stessa come listener e vedremo nel paragrafo 2.2.2.5 come l'applicazione invierà messaggi a PD al cambiare di stato di questi componenti.

```
@Override
public void onClick(View v) {
	[...]
	case R.id.pref_button:
		startActivity(new Intent(this, PdPreferences.class));
		break;
	[...]
	}
}
```

Frammento di Codice 14: Apertura dell'activity delle preferenze

Anche per il pulsante delle preferenze viene impostata l'activity stessa come listener e alla pressione del pulsante verrà visualizzata l'activity delle preferenze mostrata nell'Illustrazione 11. Per avviare l'activity delle preferenze è sufficiente forgiare un nuovo intent indicando come classe *PdPreferences.class,* come mostrato nel Frammento di Codice 14.
L'activity delle preferenze viene fornita con PD for Android e semplifica il lavoro di impostazione di PD permettendo allo sviluppatore di delegare gran parte della configurazione di PD a PD for Android. Le preferenze permettono di selezionare la frequenza di campionamento, il numero di canali in ingresso, il numero di canali in uscita e la dimensione del buffer di PD in millisecondi. All'avvio dell'applicazione, nel metodo *onCreate()* della main activity, si inizializzano le preferenze (delegando a *PdPreferences* di ricercare le impostazioni supportate della periferica) e si

registra l'activity stessa come listener per la modifica delle impostazioni; il metodo *onSharedPreferenceChanged()* si occupa di reinizializzare l'audio con le nuove impostazioni quando l'utente le modifica. La gestione delle preferenze è visibile nel Frammento di Codice 15.

```
@Override
protected void onCreate(android.os.Bundle savedInstanceState) {
	[...]
	PdPreferences.initPreferences(getApplicationContext());
PreferenceManager.getDefaultSharedPreferences(getApplicationContext()).reg
isterOnSharedPreferenceChangeListener(this);
	[...]
}

@Override
public void onSharedPreferenceChanged(SharedPreferences sharedPreferences,
String key) {
	startAudio();
}
```

Frammento di Codice 15: Impostazione delle preferenze per PdTest

Occorre infine indicare ad Android l'esistenza e l'ubicazione dell'activity delle preferenze *PdPreferences*; per farlo si aggiunge al file *AndroidManifest.xml* la seguente riga, tra i tag *application*:

```
<activity android:label="Pure Data Preferences"
	android:name="org.puredata.android.service.PdPreferences"
	android:screenOrientation="portrait" />
```

2.2.2.3 Avvio del service

Come già accennato, PdTest esegue PD all'interno di un service che gli permette di continuare ad eseguire audio anche quando l'utente passa ad altre applicazioni; il service utilizzato è *PdService* ed è fornito con PD for Android. Anche in questo caso si può delegare la gestione del service a PD for Android e allo sviluppatore rimane da gestire esclusivamente l'avvio e la connessione di *PdService*.

Per prima cosa aggiungiamo in *AndroidManifest.xml* il riferimento a *PdService* inserendo tra i tag *application* la seguente riga:

```
<service android:name="org.puredata.android.service.PdService" />
```

Quindi prepariamo il codice che gestisce la connessione e disconnessione dell'activity dal servizio che a loro volta sono gestite mantenendo un riferimento ad un oggetto di tipo *Pdservice.* A connessione avvenuta il riferimento viene inizializzato con l'istanza del servizio avviato e successivamente utilizzato per comunicare con essa come vedremo nei paragrafi successivi. La connessione è avvenuta quando viene richiamato il metodo di callback *onServiceConnected()* dell'oggetto *pdConnection* (di tipo *ServiceConnection)* che lo contiene.

Quando il service è avviato si procede alla configurazione di PD richiamando il metodo *initPd()*, come vedremo nei paragrafi successivi.

```
private PdService pdService = null;
private final Object lock = new Object();
private final ServiceConnection pdConnection = new ServiceConnection() {
	@Override
	public void onServiceConnected(ComponentName name, IBinder service)
	{
		synchronized (lock) {
		pdService = ((PdService.PdBinder) service).getService();
		initPd();
		}
	}
	@Override
	public void onServiceDisconnected(ComponentName name) {
		// this method will never be called
	}
};
```

Frammento di Codice 16: Gestione della connessione e disconnessione da PdService

L'istanza *pdConnection* contiene anche il metodo di callback *onServiceDisconnected()* che viene richiamato qualora il servizio dovesse essere disconnesso; questa condizione non si può verificare perché PdService è un servizio di tipo *bound* che, come visto nel paragrafo 1.1.2.2, si termina autonomamente quando non ci sono più componenti legati ad esso. Vedremo nel paragrafo 2.2.2.7 come gestire la disconnessione dal servizio.

L'ultimo passo per l'avvio del service è riportato nel Frammento di Codice 17 ed è la richiesta di connessione che viene effettuata nel metodo *onCreate()* dell'activity; per la connessione si usa il metodo *bindService* di Android indicando il service mediante il nome della classe che lo definisce

(*PdService.class*) *e* l'istanza *pdConnection,* appena definita, che gestisce i metodi di callback per la connessione e disconnessione.

E' possibile effettuare la connessione al servizio all'interno di un thread separato, come indicato nel frammento di codice; ciò può essere utile per migliorare la reattività dell'interfaccia grafica ma non è indispensabile. Un esempio della connessione al servizio nello stesso thread dell'interfaccia utente è visibile nella versione originale di PdTest.

```
@Override
public void onCreate(Bundle savedInstanceState) {
	[...]
	initPdService();
}
private void initPdService() {
	new Thread() {
		@Override
		public void run() {
			bindService(new Intent(MyPdTestActivity.this,
PdService.class), pdConnection, BIND_AUTO_CREATE);
		}
	}.start();
}
```

Frammento di Codice 17: Connessione a PdService

2.2.2.4 Inizializzazione di PD

L'inizializzazione di PD avviene con il metodo *initPd()* che è richiamato appena viene stabilita la connessione al servizio, cioè nel metodo *onServiceConnected* (come visibile nel Frammento di Codice 16). Il metodo *initPd* richiama quindi il metodo *startAudio()* che si occupa di impostare l'audio di PD usando le preferenze.

La prima operazione svolta da *initPd()* consiste nell'impostare un ricevitore di messaggi provenienti da PD e nell'indicare quali messaggi si vogliono ricevere. Il ricevitore dei messaggi viene impostato indicandone un'istanza al metodo *setReceiver()* di *PdBase,* vedremo nel paragrafo 2.2.2.6 come implementare il ricevitore (la cui istanza verrà chiamata *receiver).* I segnali per i quali si vuole ricevere messaggi possono essere indicati usando il metodo *subscribe()* di *PdBase*; in questo caso si riceveranno messaggi per il segnale *"android"*, corrispondente al blocco "*s android*" nella patch PD dell'Illustrazione 6.

Si procede quindi ad aprire la patch rappresentata da un file di nome *test.pd* che va posizionato nella directory delle risorse (e più precisamente in *"/res/raw"*) del progetto Eclipse.

Durante l'inizializzazione la procedura per aprire la patch prevede di ottenere un riferimento alle risorse dell'applicazione usando il metodo *getResources(),* quindi aprire il file come risorsa di tipo "raw" usando il metodo *openRawResource()* dell'istanza di risorse appena ottenuta (alla quale passiamo come parametro *R.raw.test* come d'abitudine in Android) e, infine, occorre indicare il file appena ottenuto al metodo *extractResource()* di *IoUtils*, una classe messa a disposizione tra le utilities di *PdCore*.

```
private void initPd() {
	Resources res = getResources();
	File patchFile = null;
	try {
		PdBase.setReceiver(receiver);
		PdBase.subscribe("android");
		InputStream in = res.openRawResource(R.raw.test);
		patchFile = IoUtils.extractResource(in, "test.pd",
getCacheDir());
		PdBase.openPatch(patchFile);
		startAudio();
	} catch (IOException e) {
		Log.e(TAG, e.toString());
		finish();
	} finally {
		if (patchFile != null) patchFile.delete();
	}
}
```

Frammento di Codice 18: Inizializzazione di PD

Si passa quindi all'inizializzazione dell'audio richiamando il metodo *startAudio().* Come anticipato nel paragrafo 2.2.2.2 l'uso dell'activity *PdPreferences* permette di delegare completamente l'impostazione di PD a PD for Android, all'interno di *startAudio()* è quindi sufficiente passare come parametri "-1" (un valore non accettabile) al metodo *initAudio()* di *pdService* (l'istanza del service inizializzata nel paragrafo 2.2.2.3) per inizializzare l'audio, come visibile nel Frammento di Codice 19; questi parametri "errati" obbligano libpd a scegliere o dei valori di default o quelli scelti dall'utente nelle preferenze.

```
private void startAudio() {
      String name = getResources().getString(R.string.app_name);
      try {
            pdService.initAudio(-1, -1, -1, -1);
            pdService.startAudio(new Intent(this,
MyPdTestActivity.class), R.drawable.icon, name, "Return to " + name +
".");
      } catch (IOException e) {
            toast(e.toString());
      }
}
```

Frammento di Codice 19: Inizializzazione dell'audio di PD

A questo punto è possibile indicare al servizio di avviare l'audio usando il metodo *startAudio()* di *pdService;* se usato privo di argomenti questo metodo fa partire correttamente l'audio ma il service rimarrà completamente in background e non visualizzerà alcuna notifica all'utente riguardo al suo stato; se si desidera visualizzare una notifica, come è stato fatto in PdTest, è sufficiente forgiare un intent riferito alla main activity dell'applicazione (*MyPdTestActivity*), indicare un'icona significativa per il service attivo (nel caso di PdTest il logo di PD), il nome dell'applicazione e un messaggio significativo (“Return to Pure Data Test.”). In questo modo l'utente può ritornare comodamente all'applicazione trascinando la barra superiore e cliccando sul servizio avviato da PdTest.

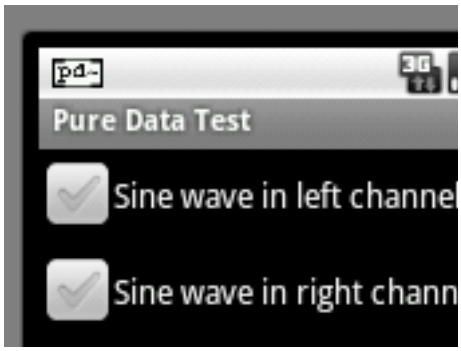

Illustrazione 12: Notifica del Service in PdTest

2.2.2.5 Spedizione di messaggi a PD

```
@Override
public void onClick(View v) {
	switch (v.getId()) {
	case R.id.left_box:
		PdBase.sendFloat("left", left.isChecked() ? 1 : 0);
		break;
	case R.id.right_box:
		PdBase.sendFloat("right", right.isChecked() ? 1 : 0);
		break;
	case R.id.mic_box:
		PdBase.sendFloat("mic", mic.isChecked() ? 1 : 0);
		break;
	...]
	}
}
```

Frammento di Codice 20: Inviare messaggi a PD in PdTest

Come anticipato nel paragrafo 2.2.1.1 è possibile inviare messaggi di ogni tipo a PD (bang, float, symbol, list, message). In PdTest devono essere inviati messaggi a PD quando l'utente tocca le check box o modifica il testo della textbox, nel paragrafo 2.2.2.2 abbiamo quindi istruito l'interfaccia grafica per richiamare il metodo *onClick()* quando l'utente tocca le checkbox e il metodo *onEditorAction()* quando modifica il testo della textbox.
Tralasciamo il codice che gestisce la textbox perché non aggiunge nulla di necessario a questa analisi e concentriamoci sulle checkbox: come visibile nel Frammento di Codice 21, quando l'utente tocca una checkbox viene generato un messaggio utilizzando il metodo *sendFloat()* al quale vengono passati come parametri il nome del ricevitore (ad esempio *"left"* per indicare *"r left"* in test.pd) e lo stato della checkbox come valore float (0 o 1). Questi messaggi modificano il comportamento della patch e il suono prodotto.

Altri tipi di messaggi possono essere inviati usando i relativi metodi di tipo *"send<Type>()"* e impostando i ricevitori corretti nella patch PD.

2.2.2.6 *Ricezione di messaggi da PD*

```
private PdReceiver receiver = new PdReceiver() {
	private void pdPost(String msg) {
		toast("Pure Data says, \"" + msg + "\"");
	}
	@Override
	public void print(String s) {
		post(s);
	}
	@Override
	public void receiveBang(String source) {
		pdPost("bang");
	}
	@Override
	public void receiveFloat(String source, float x) {
		pdPost("float: " + x);
	}
	@Override
	public void receiveList(String source, Object... args) {
		pdPost("list: " + Arrays.toString(args));
	}
	@Override
	public void receiveMessage(String source, String symbol, Object...
args) {
		pdPost("message: " + Arrays.toString(args));
	}
	@Override
	public void receiveSymbol(String source, String symbol) {
		pdPost("symbol: " + symbol);
	}
};
```

Frammento di Codice 21: Ricezione di messaggi da PD in PdTest

Come visto nel Frammento di Codice 18, durante l'inizializzazione di PD è stata indicata un'istanza di *PdReceiver* (*receiver*) come ricevitore di messaggi; in questo paragrafo sarà presentata l'implementazione di questo ricevitore.

Per ogni tipologia di messaggio che PD può inviare è definito in *PdReceiver* un metodo di callback (*receiveBang()* per i bang, *receiveFloat()* per i float, …); ognuno di questi metodi riceve il nome della sorgente che ha generato il messaggio (ad esempio riceve “android” per l'oggetto *“s android”* in test.pd) e, eventualmente, uno o più parametri per il contenuto del messaggio.
In PdTest i messaggi provenienti da PD vengono semplicemente visualizzati utilizzando un *toast* (una piccola notifica che appare sullo schermo del dispositivo Android) e riportando il messaggio ricevuto da PD, a tale scopo è stato realizzato il metodo *pdPost()*.

2.2.2.7 Terminazione dell'applicazione e pulizia

```
@Override
protected void onDestroy() {
	super.onDestroy();
	try {
		unbindService(pdConnection);
	} catch (IllegalArgumentException e) {
		pdService = null; // already unbound
	}
}
```

Frammento di Codice 22: Terminazione di PdTest

Nel paragrafo 2.2.1.1 era stato indicato che al termine dell'applicazione, nel metodo *onDestroy()*, sarebbe stato necessario indicare a PD di rilasciare le risorse utilizzando i metodi *release()* di *PdAudio* e *PdBase;* questo è vero quando si esegue PD direttamente (senza usare *PdService*). Quando si usa un service è sufficiente ricordarsi di scollegarsi nel metodo *onDestroy()*; il service è di tipo *bound*, quindi sarà lui a rilasciare le risorse e a terminarsi quando non ci saranno più activity connesse.

2.2.3 Conclusioni

Libpd è un progetto attivo da quasi 2 anni e ha raggiunto un ottimo livello di stabilità al punto che già molte applicazioni che ne fanno uso sono state pubblicate negli store di iOs e Android; durante le esperienze condotte non sono stati rilevati problemi.

Con la pubblicazione del libro di Peter Brinkmann “Making Musical Apps” [Brinkmann, 2012] l'approccio iniziale a libpd risulta molto semplificato, il libro fornisce un'autorevole base per lo studio della libreria. Oltre al libro di Brinkmann, altri riferimenti bibliografici per libpd sono [Brinkmann e altri, 2011] e [PD for Android Wiki].

PD for Android è sviluppato principalmente da Peter Brinkmann, altri sviluppatori partecipano allo sviluppo di libpd e PD for iOS.

Peter Brinkmann ha recentemente (giugno 2012) annunciato nel suo blog (http://nettoyeur.noisepages.com/) di aver aggiunto a PD for Android il supporto per le librerie audio native OpenSL ES; si dovrebbe ottenere in questo modo una riduzione dell'overhead per il

thread audio, in quanto si può evitare di passare i campioni audio tra C (dove vive PD) e Java (l'unico modo per accedere alle API audio di Android senza OpenSL ES). L'aggiornamento è completamente trasparente allo sviluppatore finale perché è PD for Android ad occuparsi di scegliere se usare l'API audio Java o la libreria nativa OpenSL.

2.3 openFrameworks

OpenFrameworks è un framework open source scritto in C++ che permette di sviluppare applicazioni multimediali multi-piattaforma; le piattaforme supportate sono Mac OS, Windows, Linux, iOS, Android.

OpenFrameworks è rilasciato sotto forma di libreria non compilata e quando lo si scarica si ottengono i suoi sorgenti che lo descrivono completamente (non ci sono altre dipendenze); in questo modo si possono mantenere diverse versioni di openFrameworks semplicemente inserendo ogni versione in una cartella diversa. Il framework ha una cartella specifica per le applicazioni (*/apps*), quindi le applicazioni sviluppate possono essere compilate su una specifica versione di openFrameworks semplicemente posizionandole nella directory */apps* della versione corretta. Questo è utile anche perché spesso diverse versioni di openFrameworks non sono compatibili tra loro: in questo modo si può quindi compilare l'applicazione sviluppata sulla versione di openFrameworks corretta.

Oltre al core di openFrameworks vengono forniti alcuni addons ufficiali (contenuti nella cartella */addons*); essi vengono mantenuti insieme al core di openFrameworks e garantiti per ogni piattaforma; altri addons possono essere scaricati dal sito ofxaddons.com.

OpenFrameworks è in realtà un insieme di librerie che sono state integrate tra loro e per le quali il framework fornisce dei wrapper; alcune delle librerie più importanti sono: openGL per la grafica, freeImage per le immagini e fmod per l'audio.

Nonostante la natura multi-piattaforma del framework, durante il tirocinio lo si è testato solamente su piattaforma Android; quanto riportato in questa presentazione si intende quindi applicabile esclusivamente a tale piattaforma.

2.3.1 Struttura delle directory in openFrameworks

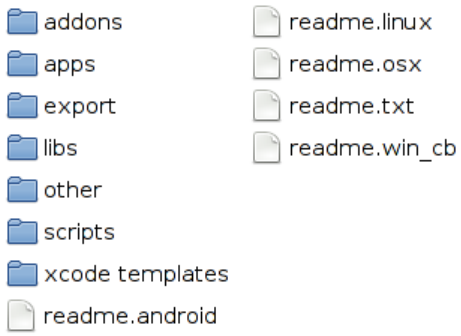

Illustrazione 13: Struttura delle cartelle in openFrameworks

La struttura delle directory in openFrameworks è importante e deve essere mantenuta. Il framework è strutturato in modo che le applicazioni effettuino i collegamenti alle librerie localmente, quindi in generale non è possibile muovere o rinominare le cartelle perché si romperebbero i riferimenti alle librerie.

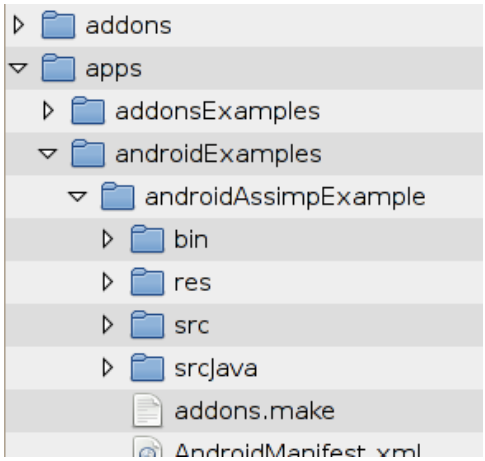

Illustrazione 14: Struttura della directory apps in openFrameworks

Le cartelle più importanti sono:

- *addons*: contiene gli addons forniti insieme al core di openFrameworks e gli addons aggiunti dallo sviluppatore
- *apps*: contiene le applicazioni di esempio e quelle sviluppate dallo sviluppatore
- *libs*: contiene il core di openFrameworks e le librerie che lo compongono

Quando si sviluppa una nuova applicazione è importante mantenere una struttura del tipo: *apps → cartella di gruppo (es. AndroidExamples) → nome del progetto → files del progetto*.

2.3.2 Preparare l'ambiente di sviluppo

Per openFrameworks esistono diversi ambienti di sviluppo da utilizzare in base al sistema operativo dal quale si sviluppa e al sistema per il quale si sviluppa. Gli IDE utilizzabili sono: xcode, code::blocks, Visual Studio 2010 ed Eclipse.

Lo sviluppo per Android è possibile esclusivamente tramite Eclipse ed è supportato ufficialmente solo su Linux e Mac OS X (esistono guide per preparare un ambiente di sviluppo Android+openFrameworks anche su Windows). La preparazione dell'ambiente di sviluppo è piuttosto complessa, ma sul sito di openFrameworks viene mantenuta una guida molto dettagliata (http://www.openframeworks.cc/setup/android-eclipse/); in particolare è molto importante la sezione “FAQ” di tale guida, che fornisce alcuni consigli fondamentali per risolvere problemi comuni. Il problema rilevato più frequentemente durante il tirocinio è stato relativo al file *build.xml* considerato obsoleto da Eclipse; la soluzione a questo problema è di eseguire il comando *android update project -p /percorso/del/progetto*.

2.3.3 Struttura delle applicazioni openFrameworks

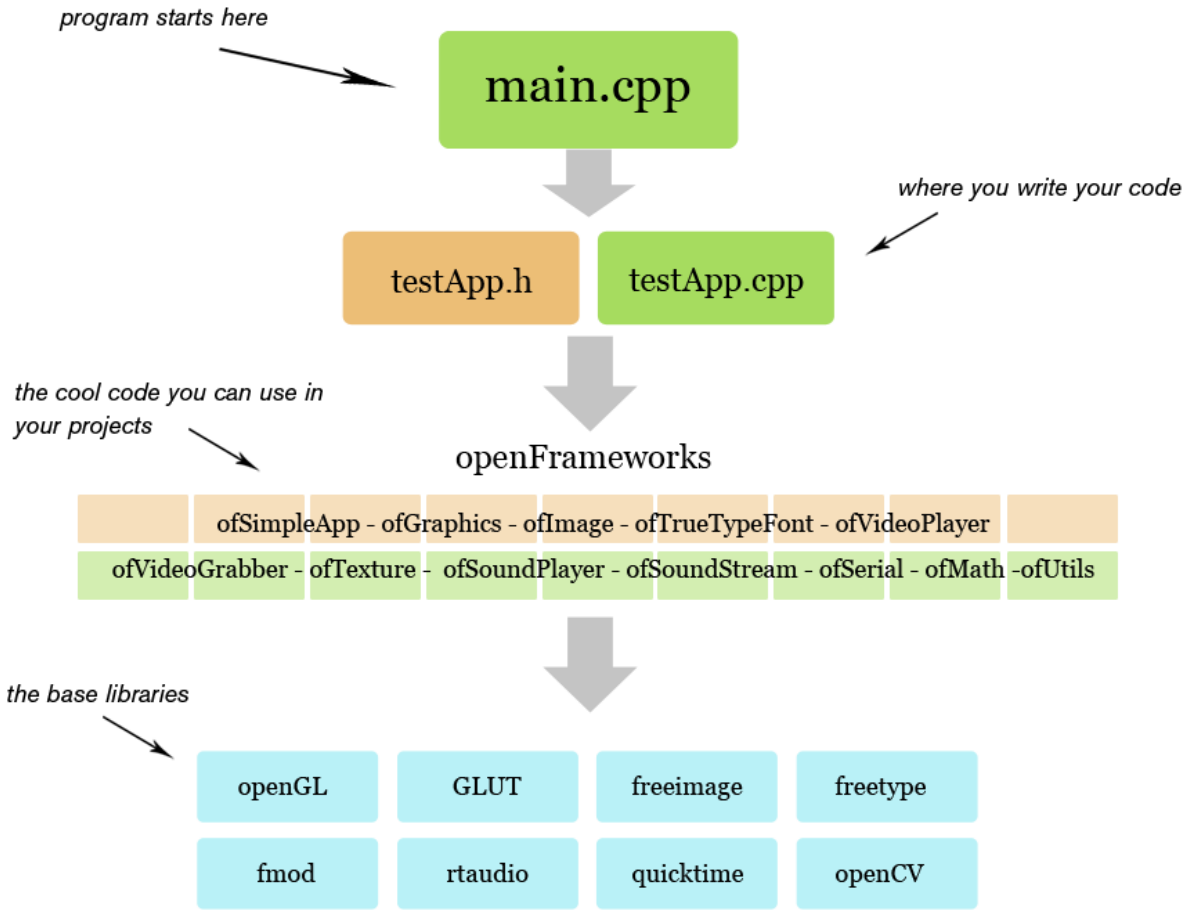

Illustrazione 15: Struttura di un'applicazione openFrameworks

Una rappresentazione della struttura delle applicazioni openFrameworks è visibile nell'Illustrazione 15: ogni applicazione openFrameworks ha inizio nel file *main.cpp* dove si comunicano al compilatore le risorse che si intendono usare (ad esempio lo stesso openFrameworks), si eseguono le impostazioni necessarie e si passa il controllo a *testApp.cpp*, dove lo sviluppatore può scrivere il proprio codice.

Analizziamo ognuno di questi file basandoci sull'applicazione di esempio per Android *androidEmptyExample*.

2.3.3.1 main.cpp

```
#include "ofMain.h"
#include "testApp.h"
#ifdef TARGET_ANDROID
	#include "ofAppAndroidWindow.h"
#else
	#include "ofAppGlutWindow.h"
#endif
int main(){
#ifdef TARGET_ANDROID
	ofAppAndroidWindow *window = new ofAppAndroidWindow;
#else
	ofAppGlutWindow *window = new ofAppGlutWindow;
#endif
	ofSetupOpenGL(window, 1024,768, OF_WINDOW);
	ofRunApp( new testApp() );
	return 0;
}
#ifdef TARGET_ANDROID
#include <jni.h>
extern "C"{
	void Java_cc_openframeworks_OFAndroid_init( JNIEnv*  env, jobject
thiz ){
		main();
	}
}
#endif
```

Frammento di Codice 23: Il file main.cpp

Le prime due righe del file indicano al compilatore quali sorgenti importare per poter accedere ad openFrameworks e all'applicazione *testApp* che andremo a scrivere. Il blocco *#ifdef* seguente serve per indicare al compilatore di importare librerie diverse al variare dell'architettura per la quale stiamo compilando; in questo caso se l'architettura è Android si usa *ofAppAndroidWindow*, altrimenti si usa una libreria standard per le altre architetture.

Troviamo quindi il metodo *main()* che si occupa di effettuare l'impostazione della finestra grafica (anche in questo caso l'impostazione è diversa a seconda dell'architettura) e di eseguire l'applicazione scritta dallo sviluppatore: *testApp*.

L'ultima parte del file contiene del codice che sfrutta la JNI per esporre a Java il metodo *init()* (scritto in C) attraverso il package Java *cc.openframeworks.OFAndroid*. Il metodo *init()* si occupa solamente di avviare il metodo *main()* ed è il punto di ingresso di Android a openFrameworks.

2.3.3.2 testApp.cpp

L'applicazione presentata in questo esempio è un'applicazione completamente vuota, quindi *testApp* contiene esclusivamente le firme dei metodi, che sono vuoti; presenteremo quindi il file *testApp.h* perché più conciso ed esauriente.

Le prime due righe del file rappresentano una direttiva per il compilatore e sono utili per evitare inclusioni multiple dello stesso file.

La terza riga, come in *main.cpp*, serve per accedere ai sorgenti di openFrameworks.

Si entra quindi nella definizione della classe e dei suoi metodi, la classe implementa l'interfaccia *ofBaseApp*. I metodi più importanti di *testApp* sono *setup()*, *update()* e *draw()*.
Il metodo *setup()* viene eseguito una sola volta all'avvio dell'applicazione, ed è quindi indicato per inizializzare le variabili ed effettuare le impostazioni iniziali. Una tipica impostazione in questo metodo è il frame rate, selezionabile usando il metodo *ofSetFrameRate(),* che regola la frequenza con la quale vengono richiamati in sequenza i metodi *update()* e *draw()*.
Il metodo *update()* può essere utilizzato per aggiornare lo stato delle variabili ma non può utilizzare le funzioni di disegno che sono riservate al metodo *draw()*.
Il metodo *draw()* si occupa di disegnare l'interfaccia grafica.

```
#ifndef _TEST_APP
#define _TEST_APP
#include "ofMain.h"
class testApp : public ofBaseApp{
	public:
		void setup();
		void update();
		void draw();
		void keyPressed(int key);
		void keyReleased(int key);
		void mouseMoved(int x, int y );
		void mouseDragged(int x, int y, int button);
		void mousePressed(int x, int y, int button);
		void mouseReleased(int x, int y, int button);
		void windowResized(int w, int h);
};
#endif
```

Frammento di Codice 24: Il file testApp.h

[Wiki openFrameworks]

2.3.4 ofxPd – libpd su openFrameworks

OfxPd è un addon per openFrameworks che permette di usare Pure Data su openFrameworks; ofxPd si basa su libpd e supporta tutte le funzionalità viste nel paragrafo 2.2.

OfxPd è un progetto ancora giovane sviluppato esclusivamente da Dan Wilcox e il supporto ad Android è praticamente assente; si pensi al fatto che mentre viene scritto questo documento non esiste ancora un esempio ufficiale di ofxPd per Android. Nei paragrafi successivi verrà presentato un progetto di test di ofxPd su Android sviluppato durante il tirocinio. Inoltre, ofxPd su Android presenta un bug che sarà presentato nelle conclusioni.

2.3.4.1 Installazione di ofxPd

OfxPd è disponibile per il download su GitHub, la versione master di ofxPd è da usare con la versione master di openFrameworks e può essere considerata relativamente stabile. Per lavorare con versioni precedenti e stabili di openFrameworks (ad esempio 0062, 007, …) è possibile selezionare un tag tramite git per ottenere una versione precedente di ofxPd; i seguenti comandi, ad esempio, scaricano ofxPd e selezionano la versione 0062:

```
git clone git://github.com/danomatika/ofxPd.git
cd ofxPd
git checkout 0062
```

Come ogni addon per openFrameworks, ofxPd va posizionato nella directory *addons*.

[Dan Wilcox, 2011]

2.3.5 AndroidPdTest – ofxPd su Android

Come accennato nei paragrafi precedenti non esiste un'applicazione di test per ofxPd su Android; durante il tirocinio ne è stata sviluppata una sulla base di *androidEmptyExample* (l'applicazione vuota presentato nel paragrafo 2.3) e dell'esempio fornito con ofxPd. L'esempio fornito con ofxPd ha lo scopo di testare il funzionamento di PD su openFrameworks; a tale scopo apre un patch PD e permette di inviare e ricevere messaggi a PD.

L'applicazione è strutturalmente simile ad *androidEmptyExample*, con il file *main.cpp* che si occupa di inizializzare la finestra grafica (ed è esattamente identico a quello presentato nel paragrafo 2.3.3.1) e quindi avvia *testApp.cpp*; *testApp* gestisce il comportamento della finestra

grafica implementando i metodi *setup()*, *update()*, *draw()* e quelli che gestiscono gli eventi (come la pressione di un tasto della tastiera o di uno del mouse), ogni metodo di *testApp,* però, non si interfaccia direttamente con ofxPd, ma delega tale gestione ad *AppCore*, che si occupa di ricevere e inviare i messaggi a PD come vedremo in dettaglio nei paragrafi successivi.

Tralasciamo il file *main.cpp* ed analizziamo la patch PD *test.pd*, la operazioni necessarie per creare una nuova applicazione openFrameworks per Android e il contenuto di *testApp* e di *AppCore*.

La patch *test.pd* permette di testare ognuna delle caratteristiche di ofxPd, ci sono dei receiver (l'oggetto *"r fromOF"*) che permettono alla patch di ricevere messaggi, ci sono dei signal (non visibili nell'Illustrazione 16) che permettono alla patch di inviare messaggi all'applicazione, l'oggetto *"adc~"* cattura l'input microfonico, l'oggetto *"dac~"* emette dei suoni, … .

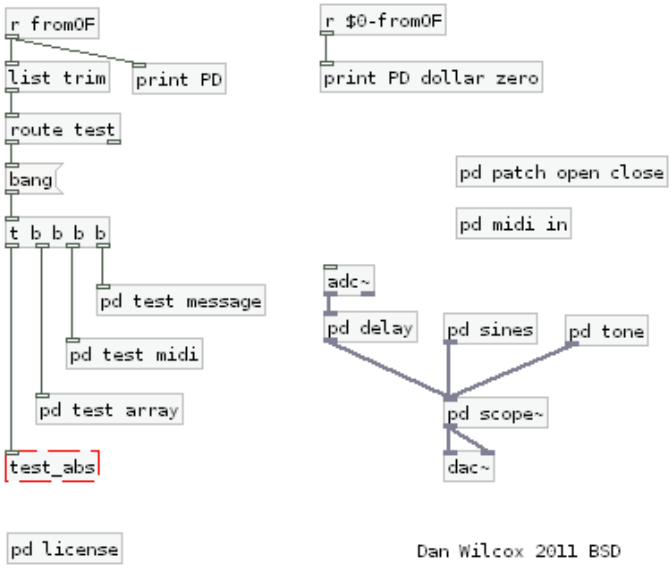

Illustrazione 16: La patch PD test.pd in AndroidPdTest

2.3.5.1 AndroidPdTest – impostazione di una nuova applicazione

In questo paragrafo vengono presentati i passi seguiti per impostare un nuovo progetto openFrameworks in ambiente di sviluppo Linux, comunque tutti i passaggi sono comuni ad ogni piattaforma escluso dove diversamente indicato in maniera esplicita.

Per impostare una nuova applicazione openFrameworks è conveniente copiare un esempio e modificarlo, a tale scopo copiamo il progetto *AndroidEmptyExample*. Scegliamo poi un nuovo nome per il progetto (*"androidPdTest"*) e seguiamo questi passi:

- modifichiamo il nome della cartella che contiene il progetto con il nome del progetto stesso, in questo caso *"androidPdTest"*
- nel file *res/value/strings.xml* modifichiamo *app_name* inserendo il nome scelto per il progetto
- nel file *AndroidManifest.xml* modificchiamo il package con il nome del progetto, ottenendo *package="cc.openframeworks.androidPdTest"*
- all'interno di *srcJava* selezioniamo il nome del package e rinominiamolo in *cc.openframeworks.androidPdTest*
- per usare *ofxPd* come addon aggiungiamo *ofxPd* nel file *addons.make* e modifichiamo nel file *config.make* le righe relative a *USER_CFLAGS*, *USER_LDFLAGS* e *USER_LIBS* come indicato nel Frammento di Codice 25 (questo passaggio si riferisce ad un ambiente di sviluppo Linux, per altre piattaforme si rimanda a https://github.com/danomatika/ofxPd/blob/master/README.markdown#adding-ofxpd-to-an-existing-project)
- copiamo infine i file i file *testApp.cpp*, *testpApp.h*, *AppCore.cpp* e *AppCore.h* dall'esempio di *ofxPd.*

```
USER_CFLAGS = -DHAVE_UNISTD_H -DUSEAPI_DUMMY -DPD -shared
USER_LDFLAGS = --export-dynamic
USER_LIBS = -ldl -lm
```

Frammento di Codice 25: Le righe da modificare nel file config.make

E' importante mantenere il prefisso del package *cc.openframeworks* altrimenti qualcosa potrebbe non funzionare più.

2.3.5.2 AndroidPdTest – testApp

```
void testApp::update() {
	core.update();
}
void testApp::draw() {
	core.draw();
}
void testApp::exit() {
	core.exit();
}
void testApp::keyPressed(int key) {
	core.keyPressed(key);
}
```

Frammento di Codice 26: Alcuni metodi di testApp in androidPdTest

Per *androidPdTest, testApp* è simile a quella presentata per *androidEmptyExample* nel paragrafo 2.3.3.2; in questo paragrafo presentiamo quindi esclusivamente le differenze.

```
void testApp::setup() {
	int ticksPerBuffer = 8;	// 8 * 64 = buffer len of 512
	ofSoundStreamSetup(2, 2, this, 44100,
ofxPd::blockSize()*ticksPerBuffer, 4);
	core.setup(2, 2, 44100, ticksPerBuffer);
}
```

Frammento di Codice 27: Il metodo setup() di testApp in androidPdTest

In questo caso il metodo *setup()* si occupa di inizializzare l'audio e di inviare i parametri che ha usato per l'inizializzazione al metodo omonimo in *AppCore*. L'audio viene inizializzato usando il metodo *ofSoundStreamSetup()* che riceve come parametri il numero di canali in ingresso e in uscita, un riferimento all'istanza di *testApp*, la frequenza di campionamento, la dimensione di ogni buffer e il numero di buffer.

I metodi di *testApp* si riferiscono ad *AppCore* mediante l'istanza *core* dichiarata in *testApp.h* con la riga di codice: *"AppCore core"*.

I metodi *update()*, *draw()*, *exit()* e *keyPressed()* si limitano a chiamare i loro omonimi in *AppCore* come visibile nel Frammento di Codice 26.

```
void testApp::audioReceived(float * input, int bufferSize, int nChannels)
{
	core.audioReceived(input, bufferSize, nChannels);
}
void testApp::audioRequested(float * output, int bufferSize, int
nChannels) {
	core.audioRequested(output, bufferSize, nChannels);
}
```

Frammento di Codice 28: Metodi per la gestione dell'audio di testApp

Inoltre troviamo due metodi nuovi dedicati alla gestione dell'audio; *audioReceived()* viene richiamato quando è stato richiesto dell'audio in ingresso utilizzando *ofSoundStreamSetup()* e il sistema possiede un buffer pieno di campioni audio catturati che devono essere letti dall'applicazione; *audioRequested()* viene richiamato quando è stato richiesto dell'audio in uscita; il sistema in questo caso richiede all'applicazione di fornire un buffer di dati da riprodurre.

```
void AppCore::setup(const int numOutChannels, const int numInChannels,
const int sampleRate, const int ticksPerBuffer) {
	ofSetFrameRate(60);
	ofSetVerticalSync(true);
	pd.init(numOutChannels, numInChannels, sampleRate, ticksPerBuffer);
	midiChan = 1; // midi channels are 1-16
	// subscribe to receive source names
	pd.subscribe("toOF");
	pd.subscribe("env");
	// add message receiver
	pd.addReceiver(*this);
	// add midi receiver
	pd.addMidiReceiver(*this);
	// audio processing on
	pd.start();
	// open patch
	Patch patch = pd.openPatch("test.pd");
	// test basic atoms
	pd.sendBang("fromOF");
	pd.sendFloat("fromOF", 100);
	pd.sendSymbol("fromOF", "test string");
	// send functions
	pd.sendNoteOn(midiChan, 60);
}
```

Frammento di Codice 29: Alcune parti fondamentali del metodo setup() di AppCore

```
class AppCore : public PdReceiver, public PdMidiReceiver {
	public:
		void setup(const int numOutChannels, const int numInChannels,
				const int sampleRate, const int
ticksPerBuffer);
		void update();
		void draw();
	 void exit();
		void playTone(int pitch);
		void keyPressed(int key);
		void audioReceived(float * input, int bufferSize, int
nChannels);
		void audioRequested(float * output, int bufferSize, int
nChannels);
		// pd message receiver callbacks
		void print(const std::string& message);

		void receiveBang(const std::string& dest);
		void receiveFloat(const std::string& dest, float value);
		void receiveSymbol(const std::string& dest, const
std::string& symbol);
		void receiveList(const std::string& dest, const List& list);
		void receiveMessage(const std::string& dest, const
std::string& msg, const List& list);
	 // pd midi receiver callbacks
		void receiveNoteOn(const int channel, const int pitch, const
int velocity);
		void receiveControlChange(const int channel, const int
controller, const int value);
		void receiveProgramChange(const int channel, const int
value);
		void receivePitchBend(const int channel, const int value);
		void receiveAftertouch(const int channel, const int value);
		void receivePolyAftertouch(const int channel, const int
pitch, const int value);
		void receiveMidiByte(const int port, const int byte);
		ofxPd pd;
		vector<float> scopeArray;
	 int midiChan;
};
```

Frammento di Codice 30: Il file AppCore.h

2.3.5.3 AndroidPdTest – AppCore

La classe *AppCore* è quella che si occupa di interagire con ofxPd e, oltre ai metodi richiamati da *testApp* come *setup()*, *update()*, *draw()*, …, ne troviamo molti dedicati all'interazione con PD come *receiveBang()*, *receiveFloat()*, *receiveSymbol()*, … che sono dichiarati nelle classi

PdReceiver e *PdMidiReceiver* e che *AppCore* implementa (come si può vedere nel Frammento di Codice 30).

Vediamo ora in dettaglio alcuni di questi metodi.

Il metodo *setup()* si occupa di inizializzare PD usando il metodo *init()* dell'istanza di ofxPd *pd* e i parametri che riceve da *testApp*, si registra per ricevere i segnali provenienti dalla patch PD usando il metodo *subscribe()* di *pd* e indica se stesso come ricevitore usando i metodi di *pd* *addReceiver()* per i messaggi e *addMidiReceiver*() per le note MIDI, da inizio al processing dell'audio da parte di PD usando il metodo *start()* di *pd*, apre la patch *test.pd* usando il metodo *openPatch()* di *pd* e infine invia dei messaggi di prova alla patch.
Nel Frammento di Codice 29 è riportata la parte fondamentale del metodo *setup()* appena presentata.

```
void AppCore::update() {
	ofBackground(100, 100, 100);
	// update scope array from pd
	pd.readArray("scope", scopeArray);
}
void AppCore::draw() {
	// draw scope
	ofSetColor(0, 255, 0);
	ofSetRectMode(OF_RECTMODE_CENTER);
	float x = 0, y = ofGetHeight()/2;
	float w = ofGetWidth() / (float) scopeArray.size(), h =
ofGetHeight()/2;
	for(int i = 0; i < scopeArray.size()-1; ++i) {
		ofLine(x, y+scopeArray[i]*h, x+w, y+scopeArray[i+1]*h);
		x += w;
	}
}
```

Frammento di Codice 31: I metodi update() e draw() di AppCore

Come abbiamo visto nel paragrafo 2.3.3.2 il metodo *update()* viene eseguito continuamente, in questo caso si occupa, usando il metodo *readArray()* di *pd*, di leggere dalla patch PD il contenuto dell'array *"scope"* e copiarlo localmente nel vettore *scopeArray*. Tale vettore contiene i campioni catturati dal microfono che vengono disegnati nel metodo *draw()*.

Come si può vedere nel Frammento di Codice 30 i metodi di tipo *receive<type>()* quando sono richiamati ricevono come parametri il nome del signal che li ha richiamati e, quando presente, il valore da ricevere.

2.3.6 Conclusioni

OpenFrameworks è un progetto attivo su github da quasi 3 anni con una buona comunità di sviluppatori attivi, la documentazione è adeguata e esiste un forum ufficiale. Il framework è particolarmente interessante perché permette di portare su una grande varietà di piattaforme applicazioni scritte in codice nativo ed è particolarmente indicato per applicazioni di grafica e di elaborazione audio e video. Una delle grandi potenzialità del framework è la possibilità di utilizzare degli addons resi disponibili da una grande comunità e che spesso, come nel caso di ofxPd, si basano su librerie o software open source.
Nonostante ciò Android è una piattaforma ancora poco popolare per questo framework, dato che la prima versione di openFrameworks con supporto beta ad Android è stata introdotta solamente un anno fa. Un altro ostacolo che gli sviluppatori di openFrameworks stanno incontrando è il cattivo supporto di Eclipse alle applicazioni Android scritte in codice nativo. Ciò rende lo sviluppo più laborioso e spesso le applicazioni non compilano esclusivamente a causa di questo cattivo supporto.

Per quanto riguarda ofxPd il supporto ad Android risulta inesistente e, come già accennato nel paragrafo 2.3.4, non è disponibile un esempio d'uso di ofxPd in Android.
Durante lo sviluppo dell'esempio *AndroidPdTest* e in altre applicazioni di test sviluppate appositamente è stato rilevato un bug di ofxPd quando usato su Android; tale bug si presenta qualora venga abilitato l'audio in ingresso e provoca un fortissimo rallentamento dell'interfaccia grafica che rende l'applicazione inutilizzabile.

Capitolo 3: Misure di prestazioni

Oltre alle esperienze presentate nel Capitolo 2, è stato posto tra gli scopi del tirocinio quello di valutare le capacità di elaborazione audio real-time della piattaforma Android. In particolare per sviluppare questo tipo di applicazioni è necessaria una bassa latenza tra la cattura e l'esecuzione dell'audio, e la capacità di eseguire elaborazioni in tempo reale.

In questo capitolo verranno presentati inizialmente le informazioni raccolte e i test condotti per quanto riguarda la latenza, quindi verrà presentata l'applicazione di filtraggio audio sviluppata. Quest'ultima ha lo scopo di valutare le capacità di elaborazione della piattaforma ricorrendo anche al codice nativo, ed è stata sviluppata dopo aver appurato che la piattaforma non è adatta ad applicazioni di elaborazione audio in real-time a causa dell'elevata latenza della catena audio.

3.1 Latenza

Il primo e più importante obiettivo del tirocinio è stato la valutazione della latenza di Android e in particolare la latenza minima ottenibile usando libpd; è stata condotta quindi una ricerca preliminare di informazioni sull'argomento.

Da subito le informazioni ottenute hanno rivelato che la piattaforma presenta una latenza molto elevata sia nella cattura che nell'esecuzione di audio; come vedremo nel paragrafo 3.1.2 la latenza è nell'ordine delle centinaia di millisecondi, un valore inaccettabile per qualsiasi applicazione che intenda eseguire elaborazione di audio in tempo reale. Tale limitazione viene rilevata da moltissimi sviluppatori ed è stata aperta una issue (la numero 3434 http://code.google.com/p/android/issues/detail?id=3434) nel repository ufficiale di Android per sollevare il problema e richiedere l'introduzione di un supporto per l'audio in tempo reale. Nonostante la protesta della comunità, nessuna risposta ufficiale è giunta dai membri del team Android e, solo nella recente “Android Fireside Chat” tenutasi durante il “Google I/O 2012”, Dave

Burke, engineering manager di Google UK, ha accennato che il team di Android sta lavorando per portare la latenza della piattaforma sotto i 10 millisecondi.

L'unica speranza rimaneva quella che libpd potesse superare le limitazioni della piattaforma accedendo alla API audio di sistema direttamente. Anche in questo caso è stata condotta una ricerca preliminare di informazioni che ha portato a constatare che anche libpd è costretto ad accedere all'audio usando l'API Java e presenta quindi la stessa latenza. Per libpd esiste un'applicazione sviluppata appositamente allo scopo di valutare la latenza dell'intera catena audio. Il ritardo viene calcolato riproducendo un suono, catturandolo con il microfono del dispositivo e quindi rilevandolo usando una patch PD. Il tempo che intercorre tra la riproduzione del suono e la sua rilevazione è la latenza dell'intera catena audio. L'applicazione per il test della latenza di libpd si chiama *PureDataLatencyTester* ed è disponibile sul sito di PD (http://puredata.info/docs/AndroidLatency), dove possono essere pubblicati i risultati del test della latenza eseguito sul dispositivo del quale si dispone. Durante il tirocinio ho pubblicato i risultati del test eseguiti su Samsung Galaxy S Plus (con Android 2.3.3) e Samsung Galaxy Tab 10.1 (con Android 3.1) rilevando rispettivamente latenze di 530ms e 539ms. L'applicazione non veniva eseguita nell'Archos 32 (con Android 2.2) ma è stata modificata per potervi funzionare. La latenza misura su questo dispositivo è di 590ms.

Durante il tirocinio è stato deciso che sarebbe stato comunque utile valutare come questi ritardi siano distribuiti lungo la catena audio, ovvero quanto sia dovuto alla riproduzione, quanto alla cattura e quanto all'elaborazione. Di queste tre componenti di ritardo l'unica per la quale si è trovato un metodo di misura è la riproduzione; nei prossimi paragrafi vengono presentati il metodo e l'applicazione sviluppata (*LatencyTester*).

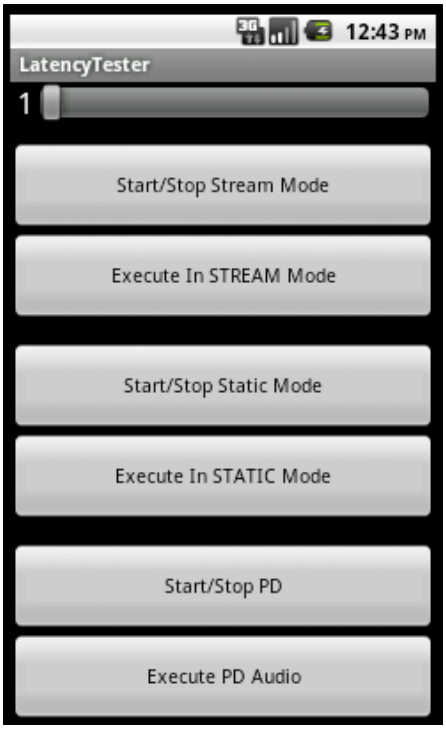

Illustrazione 17: L'applicazione LatencyTester

3.1.1 Misure di latenza – Applicazione sperimentale

Al fine di eseguire una misura della latenza introdotta dalla piattaforma Android durante l'esecuzione di audio è stata sviluppata l'applicazione *LatencyTester*.

Si è deciso che sarebbe stato più semplice ed affidabile misurare la latenza esternamente e lasciare all'applicazione il solo compito di eseguire un suono. Come si può vedere nell'Illustrazione 20, l'applicazione dispone solo dei pulsanti per inizializzare l'audio e di quelli per eseguirlo. Usando un'applicazione di registrazione audio esterna (durante il tirocinio è stato usato Audacity) e un microfono è possibile registrare il rumore dovuto al tocco del pulsante di avvio dell'audio e, poco dopo, l'effettivo inizio del suono. Come si può intuire dall'Illustrazione 18, è possibile misurare approssimativamente il tempo che intercorre tra i due suoni. Dopo un adeguato numero di ripetizioni, si può quindi calcolare una media di tutte le misurazioni con una precisione nell'ordine della decina di millisecondi, adeguata al nostro scopo.

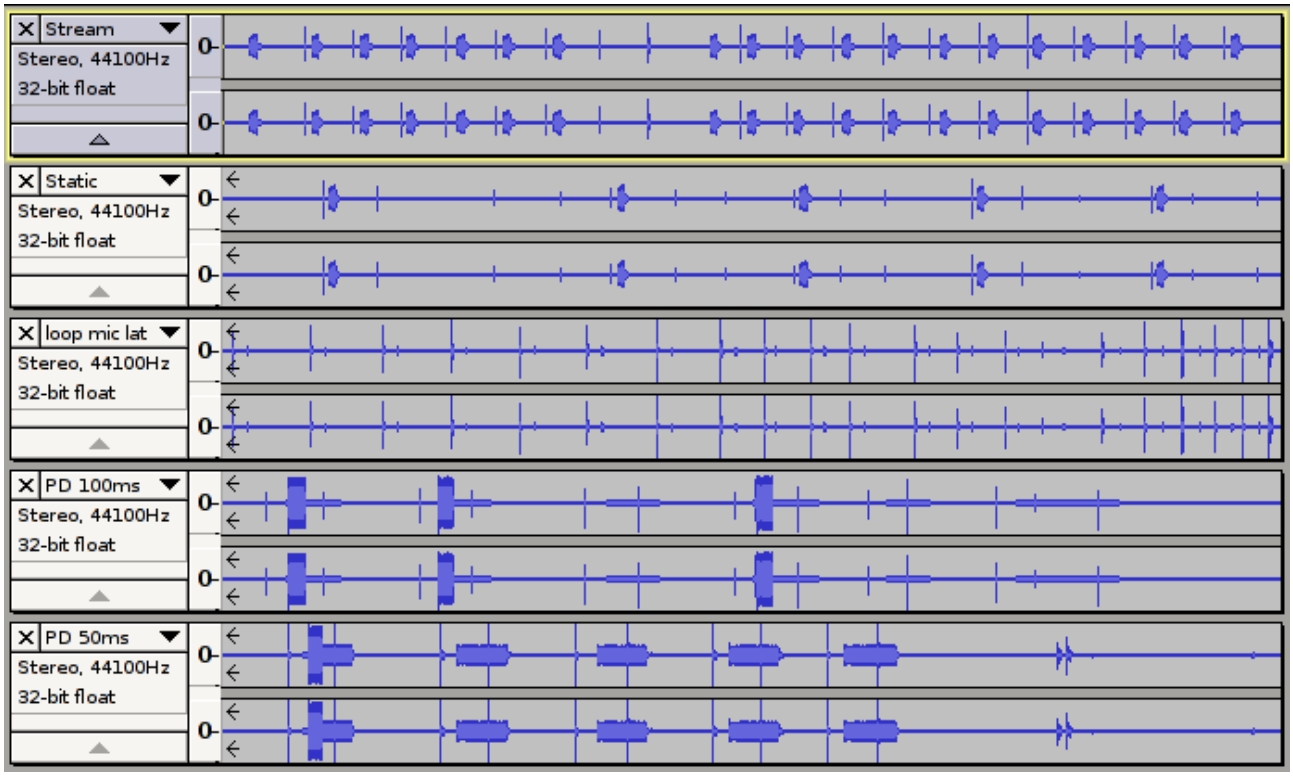

Illustrazione 18: Le registrazioni effettuate per misurare la latenza in Android

```
public void loadStreamMode(View v) {
	int playBufferInBytes = AudioTrack.getMinBufferSize(SAMPLE_RATE,
OUTPUT_CHANNELS, OUTPUT_ENCODING);
	audioData = createAudioData(playBufferInBytes);
	playerInStreamMode = new AudioTrack(AudioManager.STREAM_MUSIC,
SAMPLE_RATE, OUTPUT_CHANNELS, OUTPUT_ENCODING, playBufferInBytes,
AudioTrack.MODE_STREAM);
	playerInStreamMode.play();
}
public void loadStaticMode(View v) {
	int playBufferInBytes = AudioTrack.getMinBufferSize(SAMPLE_RATE,
OUTPUT_CHANNELS, OUTPUT_ENCODING);
	audioData = createAudioData(playBufferInBytes);
	playerInStaticMode = new AudioTrack(AudioManager.STREAM_MUSIC,
SAMPLE_RATE, OUTPUT_CHANNELS, OUTPUT_ENCODING, audioData.length *
BYTES_PER_SAMPLE, AudioTrack.MODE_STATIC);
}
public void executeStreamMode(View v) {
	playerInStreamMode.write(audioData, 0, audioData.length);
}
public void executeStaticMode(View v) {
	playerInStaticMode.write(audioData, 0, audioData.length);
	playerInStaticMode.play();
}
```

Frammento di Codice 32: Versioni semplificate dei metodi di inizializzazione e di esecuzione di AudioTrack in LatencyTester

Per le modalità *stream* e *static* viene eseguita una sinusoide a 440Hz i cui campioni vengono precalcolati usando il metodo *createAudioData()* quando l'utente clicca sui pulsanti *"Start/Stop [Stream o Static] Mode"*.

L'inizializzazione di *AudioTrack* nelle due modalità è abbastanza simile e, nello sviluppo dell'applicazione, si è cercato di non favorirne una rispetto all'altra. Infatti, mentre nella modalità *stream* i dati devono essere inseriti nel buffer al momento dell'esecuzione, nella modalità *static* è possibile precaricare i dati nel buffer di *AudioTrack* e, quando li si vuole eseguire, è sufficiente richiamare il metodo *play()*. Operando in questo modo la modalità *static* sarebbe sicuramente avvantaggiata ma ciò falserebbe i risultati, dato che quando si elabora un segnale in tempo reale non si dispone dei dati fino al momento in cui è necessario eseguirli.

Nel Frammento di Codice 32 sono riportati i metodi di inizializzazione di *AudioTrack* semplificati eliminando i controlli di stato, i commenti e i *Toast* visualizzati come feedback all'utente. Come si può vedere, i due metodi sono molto simili: entrambi calcolano la minima dimensione possibile per il buffer di *AudioTrack*, precalcolano i campioni audio da eseguire e inizializzano *AudioTrack*. L'unica differenza è che in modalità *stream* il metodo *play()* di *AudioTrack* deve essere chiamato prima di inserire i dati nel buffer.

Nel Frammento di Codice 32 sono riportati inoltre i metodi di esecuzione per le due modalità di *AudioTrack*; anche in questo caso i due metodi sono molto simili e si limitano ad inserire i dati nel

```
public void loadPd(View v) {
	bindService(new Intent(LatencyTesterActivity.this, PdService.class),
pdConnection, BIND_AUTO_CREATE);
}
public void executePdAudio(View v) {
	if(!isPdOn){
		PdBase.sendFloat("on", 1);
		isPdOn=true;
	}
	else{
		PdBase.sendFloat("on", 0);
		isPdOn=false;
	}
}
```

Frammento di Codice 33: Versioni semplificate dei metodi di inizializzazione e di esecuzione di libpd in LatencyTester

buffer di *AudioTrack;* la sola differenza è che in modalità *static,* dopo aver inserito i dati nel buffer, bisogna comunicare ad *AudioTrack* di eseguirli richiamando il metodo *play()*.

Anche nella terza modalità (Frammento di Codice 33), che fa uso di libpd, troviamo un metodo di inizializzazione e uno di esecuzione che riproduce una sinusoide a 440Hz.
Alla pressione del pulsante *"Start/Stop PD"* viene richiamato il metodo *loadPd()* che si occupa di inizializzare libpd; questo metodo esegue il bind su *PdService* proprio come abbiamo visto nel paragrafo 2.2.2.3 (il codice che gestisce il service non è mostrato perché identico a quello già visto), anche qui, come nel paragrafo 2.2.2.2, si usa l'activity delle preferenze *PdPreferences* per permettere all'utente di scegliere le preferenze di impostazione di PD.
Quando l'utente tocca il pulsante *"Execute PD Audio"* viene richiamato il metodo *executePdAudio()* che attiva o disattiva l'audio inviando un messaggio a PD.

```
private short[] createAudioData(int playBufferInBytes) {
	int nOfSamplesToCreate = (playBufferInBytes / BYTES_PER_SAMPLE)
				* getSeekBarValue();
	double sinCycleStepPercentage = SIN_FREQUENCY / SAMPLE_RATE;
	double sinCyclePosition = 0;
	short[] audioData = new short[nOfSamplesToCreate];
	for (int i = 0; i < audioData.length; i++) {
		audioData[i] = (short) (Short.MAX_VALUE * Math.sin(2 *
Math.PI * sinCyclePosition));
		sinCyclePosition += sinCycleStepPercentage;
		if (sinCyclePosition >= 1)
			sinCyclePosition = 0;
	}
	return audioData;
}
```

Frammento di Codice 34: Il metodo createAudioData() di LatencyTester

3.1.2 Misure di latenza – Risultati sperimentali

Per quanto riguarda *AudioTrack* le misure sperimentali confermano che la latenza minima si ottiene utilizzandolo in modalità *static* (come era stato detto nel paragrafo 1.1.3.1).
Per le due modalità di *AudioTrack* i test sono stati eseguiti utilizzando un Archos 32 (con Android 2.2) e configurando l'audio con frequenza di campionamento 44.1KHz, in mono e con campioni da 16bit (44.1KHz a 16bit sono le specifiche audio massime supportate da Android). Utilizzando queste configurazioni otteniamo una dimensione minima del buffer di *AudioTrack* di 12288 byte che, con la configurazione usata, corrisponde ad un suono di 139ms. La dimensione minima del buffer varia da periferica a periferica in funzione della potenza di calcolo e di scelte del costruttore.

Usando la modalità *stream* è stata misurata una latenza media di 150ms; in questa modalità può succedere che la latenza cresca fino a qualche secondo (sono stati misurati picchi fino a 4s) quando si lascia A*udioTrack* inattivo anche solo per qualche secondo senza inserire campioni nel suo buffer. Nonostante ciò, con un utilizzo continuo o al più con brevi istanti di pausa, la latenza rimane costante attorno a 150ms.
Usando la modalità *static* è stata misurata una latenza media di 110ms con scostamenti non valutabili al livello di approssimazione raggiunto.

Per quanto riguarda libpd la configurazione dell'audio prevedeva una frequenza di campionamento di 44.1KHz, 1 canale in uscita e 1 canale in ingresso. Per quanto riguarda la dimensione del buffer di PD sono stati effettuati due test per valutare quale fosse l'effetto di questo parametro sulla latenza finale.
Il primo test prevedeva un buffer PD di 100ms ed è stata misurata una latenza media di 410ms. Il secondo test è stato effettuato con un buffer di 50ms, la latenza media misurata è stata di 370ms. Anche in questo caso il livello di approssimazione è nell'ordine della decina di millisecondi; possiamo quindi sostenere che la variazione della dimensione del buffer PD provoca una variazione lineare della latenza totale. Sottraendo la latenza dovuta alla bufferizzazione di PD ritroviamo una latenza residua di circa 300ms, 150ms in più di quanto avevamo misurato usando *AudioTrack*; non possiamo comunque stabilire se questo ulteriore ritardo sia introdotto dall'elaborazione eseguita da PD o sia un ritardo dovuto all'invio del messaggio che comunica alla patch PD di eseguire la sinusoide.

Un ultimo test è stato condotto utilizzando l'applicazione che verrà presentata nel paragrafo 3.2 che, tra le sue funzionalità, effettua l'echo del microfono introducendo il minimo ritardo possibile. Per questo test la configurazione di *AudioTrack* e di *AudioRecord* è identica a quella utilizzata in *LatencyTester*; sull'Archos 32 con questa configurazione otteniamo una dimensione del buffer di *AudioRecord* di 3528 byte, che corrisponde ad un suono di 40ms. *AudioTrack* è usato in modalità *stream*.
Il risultato misurato con questa esperienza è una latenza media di 350ms. Questo test ci aiuta a stimare la latenza introdotta dal sistema durante la cattura dell'audio; sappiamo che la latenza per l'esecuzione è di 150ms e, sottraendola al totale di questo test, rimangono 200ms. Considerando

nulla in questo caso la latenza per l'elaborazione, possiamo stimare la latenza in registrazione a 200ms circa.

Questi test confermano le informazioni raccolte in maniera preliminare: attualmente non è possibile realizzare applicazioni che richiedano elaborazione di audio in real-time su piattaforma Android.

3.2 Implementazione di filtri

Per valutare le prestazioni di elaborazione audio della piattaforma è stato scelto di realizzare un'applicazione di filtraggio digitale. Tale applicazione permette di ricevere in ingresso dei campioni audio provenienti dal microfono o letti da un file WAVE, quindi di eseguire sul segnale un filtraggio digitale e infine di riprodurre il segnale. Inoltre, qualora l'input provenga da un file WAVE, è possibile memorizzare il risultato del filtraggio in un altro file WAVE.

Per poter realizzare i filtri digitali mi sono basato su alcune informazioni ricavate dal libro di Julios O. Smith III “Introduction to Digital Filters with Audio Applications” [Smith, 2007]. Lo studio dei filtri digitali durante il tirocinio è stato limitato alla comprensione dei diagrammi di flusso della prima forma diretta e della seconda forma trasposta (le due forme implementate).

3.2.1 Filtri digitali

Un filtro digitale è del tutto equivalente ad un filtro analogico: si tratta di un sistema che modifica un segnale al fine di esaltarne o ridurne alcune caratteristiche. Un filtro digitale elabora solamente segnali campionati a tempo discreto, riceve in ingresso dei campioni e restituisce in uscita gli stessi campioni modificati. Ogni filtro analogico può essere approssimato con un filtro digitale ad un certo livello di precisione deciso dal realizzatore ma non sempre è possibile portare un filtro digitale in analogico. Il funzionamento dei filtri digitali si basa sulla presenza di memorie e di uno o più processori.

$$y(n) = b_0\,x(n) + b_1\,x(n-1) + \cdots + b_M\,x(n-M) \\ -a_1\,y(n-1) - \cdots - a_N\,y(n-N)$$

$$\sum_{i=0}^{M} b_i\,x(n-i) - \sum_{j=1}^{N} a_j\,y(n-j)$$

Illustrazione 19: Equazione differenziale del filtro tratta da [Smith, 2007]

I filtri possono essere realizzati in forme diverse e, per rappresentare il loro funzionamento, si ricorre a dei diagrammi di flusso. In questa presentazione sono presentate la prima forma diretta e la seconda forma trasposta.

La prima forma diretta fa uso dei coefficienti dell'equazione differenziale del filtro e di alcuni elementi di ritardo (z^{-1}). Tali elementi di ritardo possono essere realizzati mediante delle memorie; vanno dunque mantenuti nel tempo M-1 campioni in ingresso e N campioni in uscita (dove M ed N sono il numero di coefficienti del filtro, Illustrazione 19). Per calcolare la nuova uscita, vengono sommati, utilizzando i coefficienti del filtro, l'ingresso e tutte le memorie .

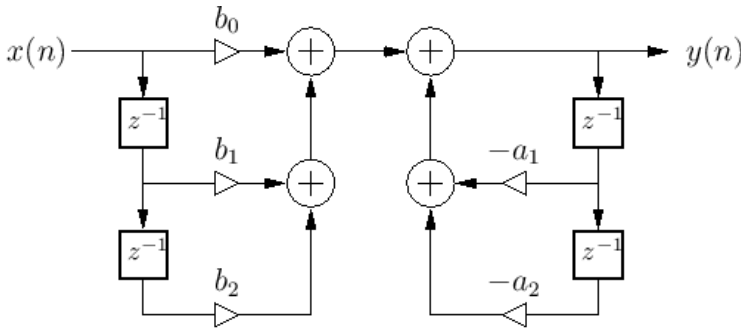

Illustrazione 20: Prima forma diretta tratta da [Smith, 2007]

La seconda forma trasposta fa uso degli stessi coefficienti usati per la prima forma diretta ma con un minore uso di ritardi (e quindi di memorie). Vedremo il funzionamento di questo tipo di filtro nel paragrafo 3.2.2.2.

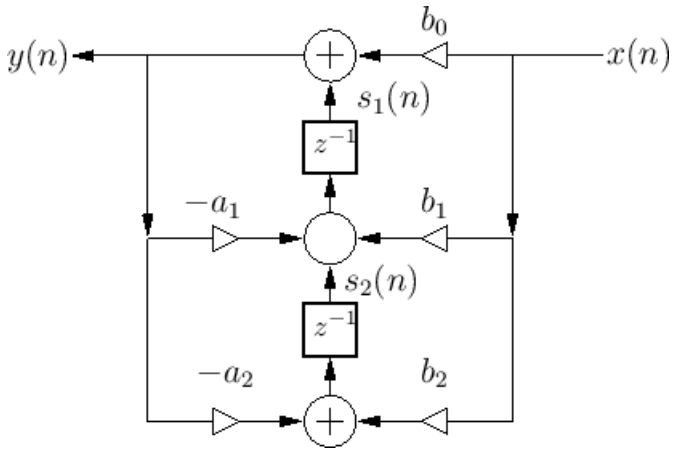

Illustrazione 21: Seconda forma trasposta tratta da [Smith, 2007]

3.2.2 FilterPlayer

FilterPlayer è un'applicazione che permette di eseguire file WAVE (solo 16bit, 44.1KHz, stereo) o di effettuare l'eco del microfono. A questo scopo sono state realizzate e verranno presentate nei paragrafi seguenti le classi *WaveDecoder* e *MicDecoder*; queste due classi implementano l'interfaccia *AudioDecoder* definita all'interno dell'applicazione e sono utilizzate per accedere in maniera standardizzata a sorgenti differenti. A questi flussi audio possono dunque essere applicati alcuni filtri che vengono organizzati in una catena di filtri. Infine, il risultato può essere eseguito usando *AudioTrack*. Il percorso dei campioni all'interno dell'applicazione è quindi il seguente:

```
-------------     ---------------------------------------------------     ---------------
|  Decoder  | --> |  FilterChain -> |Filtro 1| -> ... -> |Filtro n|  | --> |  AudioTrack  |
-------------     ---------------------------------------------------     ---------------
```

I filtri sono stati sviluppati inizialmente in Java e successivamente portati in C++ utilizzando JNI per migliorare le prestazioni dell'applicazione. I filtri Java sono implementati in prima forma diretta e in seconda forma trasposta, mentre in C++ solamente in seconda forma trasposta . Lo sviluppatore che intenda usare le classi di filtraggio sviluppate non si deve preoccupare del fatto che possano essere scritte in codice nativo perché ogni filtro (sia esso scritto in Java o in C++) implementa l'interfaccia Java *Filter*. La classe che effettivamente implementa il filtro in C++ è *filter* con la “f” minuscola. Per permettere a *filter* di essere visibile in Java come implementazione

della classe *Filter* è stato realizzato un wrapper che è composto di una classe Java (*JNIFilterWrapper*) e di una C++ (*FilterAdapter*).

L'interfaccia utente dell'applicazione permette all'utente di utilizzare due filtri; il primo filtro è un volume ed agisce semplicemente sul guadagno, il secondo è un filtro passa-banda tra 4 e 6KHz che può essere attivato e disattivato.

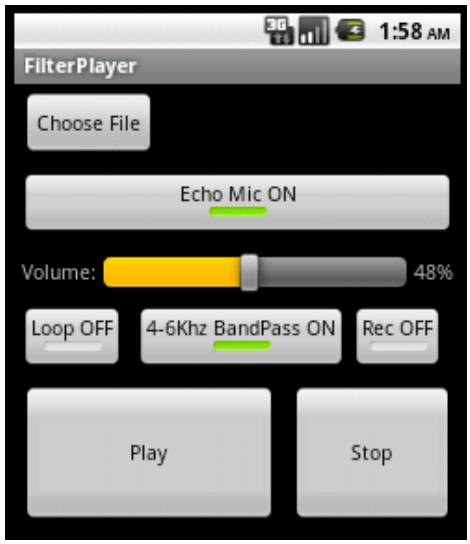

Illustrazione 22: L'applicazione FilterPlayer

L'elaborazione e l'esecuzione dell'audio viene eseguita all'interno di un service. Come abbiamo visto nei paragrafi 1.1.2.2 e 2.2.1.2, ciò permette di continuare ad eseguire l'audio anche dopo il passaggio in stop dell'applicazione.

Nei paragrafi seguenti vengono discussi in dettaglio i componenti presentati in questa introduzione. Frammenti di codice di questa applicazioni sono riportati in un paragrafo separato (4.1) per migliorare la leggibilità; nel testo sono presenti riferimenti precisi per aiutare il lettore.

3.2.2.1 AudioDecoder

AudioDecoder è l'interfaccia che è stata realizzata durante il tirocinio per utilizzare comodamente il decoder WAVE *WaveDecoder* e quello per il microfono *MicDecoder*. Questa interfaccia dispone di alcuni metodi per leggere la configurazione del decoder: *getChannels()* e *getChannelsConfiguration()* per conoscere quanti canali sono decodificati, *getFrequency()* per la frequenza di campionamento e *getBytesPerSample()* per la dimensione dei campioni. L'interfaccia

espone inoltre un metodo *read()* per leggere dei campioni dal decoder e un metodo *stop()* per fermare il decoder.

```
public interface AudioDecoder {
	public int getChannels();
	public int getChannelsConfiguration();
	public int getFrequency();
	public int getBytesPerSample();
	public short[] read(int nOfSamples) throws IOException,
DataFormatException;
	public void stop() throws IOException;
}
```

Frammento di Codice 35: L'interfaccia AudioDecoder di FilterPlayer

La classe *WaveDecoder* implementa i metodi di *AudioDecoder* e ne aggiunge di specifici; oltre ad alcuni costruttori, troviamo metodi come *readHeader()* che legge l'header WAVE e *getDuration()* che fornisce la durata del brano in millisecondi.

WAVE è un formato di file di tipo RIFF ed è composto da 3 parti:

1. la prima parte è l'header RIFF ed indica che tipo di stream è contenuto nel file; in questo caso uno stream WAVE
2. la seconda parte è l'header WAVE che indica il formato del flusso audio, il numero di canali, la frequenza di campionamento e il numero di byte per campione
3. l'ultima parte contiene informazioni sulla dimensione utile del payload (i campioni) ed i campioni stessi.

Il metodo *readHeader()* (riportato nel paragrafo 4.1.1) si occupa di ricavare questi dati dal file WAVE e di fornirli agli altri metodi della classe. Questo metodo è basato sull'articolo “Android Audio: play a WAV file on an AudioTrack” pubblicato su [Blog "Mind The Robot"].

Il metodo *read()* provvede quindi a leggere i campioni dal file WAVE e fornirli in uscita. Durante lo sviluppo di questo metodo si è cercato di ottimizzare il codice; la versione finale è riportata nel paragrafo 4.1.2. Si è cercato di ridurre l'intervento del Garbage Collector evitando di istanziare ad ogni esecuzione nuovi array sui quali memorizzare i campioni letti. Si è visto infatti che in quasi tutte le esecuzioni veniva richiesto di leggere lo stesso numero di campioni; infatti solamente giunti in prossimità del termine del file è probabile dover leggere un numero di campioni inferiore. Sfruttando questo comportamento ad ogni esecuzione invece di istanziare nuovi array si controlla se quelli usati precedentemente possano essere riutilizzati.

Un'altra sezione del metodo che è stata ottimizzata è quella che si occupa di leggere i campioni dal file. Per capire come questa sezione sia stata ottimizzata, bisogna dire che nel file WAVE la sezione dove sono memorizzati i campioni presenta un ordine dei byte di tipo Little Endian; Java funziona normalmente in modalità Big Endian e, sebbene sia possibile leggere un *ByteBuffer* in modalità Little Endian (come peraltro è stato fatto in *readHeader()*), ciò comporta performance molto scarse. E' stato quindi deciso di effettuare manualmente la conversione tra i due ordini di byte. Dopo aver sviluppato diverse soluzioni si è individuata come la migliore quella proposta da Peter Franza nel suo blog [Blog "Software Evolved"]. Questa soluzione prevede di prelevare due byte, quindi di effettuare l'AND bit a bit di ogni byte con *0xff*, di traslare il byte più significativo a sinistra di 8bit e infine di effettuare l'OR bit a bit del byte meno significativo e di quello più significativo appena traslato.

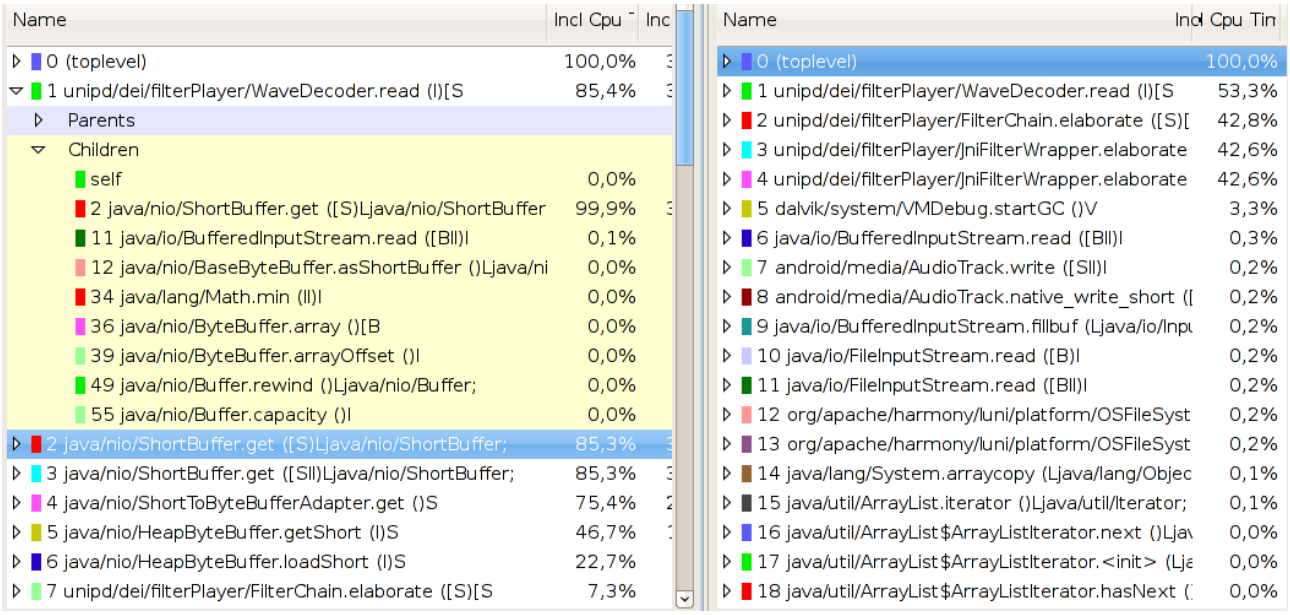

Illustrazione 23: Confronto tra la conversione Little Endian → Big Endian usando Java (sinistra) e il metodo di Peter Franza (destra)

Nell'Illustrazione 23 è riportato un confronto tra i due metodi di conversione da Little Endian a Big Endian; nella figura di sinistra il metodo fa uso di *ByteBuffer* (Frammento di Codice 36) mentre nella figura di destra usa il metodo proposto da Peter Franza. Si nota che nel primo caso la lettura del file occupa circa il 90% del tempo dell'applicazione mentre nel secondo caso circa il 55%: un miglioramento notevole. Inoltre il dettaglio dei metodi richiamati da *read()* nel primo caso mostra come la totalità del tempo sia impiegata proprio per la conversione dell'ordine dei byte.

```
buffer = ByteBuffer.allocate(nOfSamples * bytesPerSample);
buffer.order(ByteOrder.LITTLE_ENDIAN);
input.read(buffer.array(), buffer.arrayOffset(), buffer.capacity());
buffer.rewind();
ShortBuffer shorts = buffer.asShortBuffer();
shorts.get(audioData);
```

Frammento di Codice 36: Lettura dei campioni dal file WAVE usando ByteBuffer

Il secondo decoder sviluppato è *MicDecoder*. Questo decoder funge da interfaccia verso *audioRecord* e l'unico metodo interessante è *read()* riportato nel paragrafo 4.1.3.
Tale metodo mantiene una struttura semplice sfruttando la caratteristica del metodo *read()* di *AudioRecord* di essere di tipo bloccante; perciò, dopo esser stato richiamato, tale metodo non ritorna dall'invocazione fin tanto che non sia disponibile il numero di campioni richiesto.

3.2.2.2 Filter

Come già accennato in precedenza, in *FilterPlayer* i filtri sono organizzati in una catena di tipo FIFO rappresentata dalla classe *FilterChain* (paragrafo 4.1.4). Questa classe permette di inserire nuovi filtri nella catena e dispone di un metodo di nome *elaborate()* che richiama in successione l'omologo metodo di ogni filtro.

L'interfaccia *Filter* è minimale e impone l'implementazione del solo metodo *elaborate()* che riceve in ingresso un array di *short* contenente i campioni da elaborare. Quest'interfaccia permette di inserire in *FilterChain* filtri Java e C++ in maniera completamente trasparente.

Per quanto riguarda l'implementazione dei filtri, sarà presentata solamente l'implementazione del filtro in seconda forma trasposta perché più performante; ci riferiremo esclusivamente alla versione C++, cioè quella contenuta nella classe *filter;* la versione Java è simile.
L'implementazione del filtro in prima forma diretta è stata realizzata in una fase preliminare solamente a scopo didattico. Per sviluppare il filtro in seconda forma trasposta ci si è basati sul diagramma di flusso dell'Illustrazione 21. Come già accennato in precedenza, gli elementi contraddistinti da *"z^{-1}"* si comportano come dei ritardi e riportano parte del segnale di ingresso e di uscita nei campioni successivi. Questi elementi di ritardo possono essere realizzati mediante delle memorie; è sufficiente memorizzare il valore di alcune variabili di stato (quelle denominate $s_1(n)$ e

$s_2(n)$ in figura) e applicarle al campione successivo. Durante l'elaborazione di ogni nuovo campione è quindi possibile calcolare:

- l'uscita in funzione del campione attuale $x(n)$ (*x* in *filter*) e della memoria $s_1(n)$ (che ha indice 0 nel codice di *filter*)
- le nuove memorie in funzione dell'uscita $y(n)$ (*y* in *filter*), del campione attuale e delle vecchie memorie.

La classe *filter* permette di utilizzare un numero di parametri arbitrario deciso al momento della creazione di una sua istanza. Il sorgente del metodo *elaborate()* è riportato nel paragrafo 4.1.5.

Il codice che effettua il wrapping di *filter* in Java è composto di due parti: *filterAdapter* in C++ e *JniFilterWrapper* in Java. *filterAdapter* si occupa di implementare i metodi dichiarati come nativi in *JniFilterWrapper*. Per inizializzare oggetti C++ di tipo *filter* da Java si utilizza un semplice espediente: le due classi si passano un puntatore all'oggetto C++ istanziato. Vediamo il funzionamento in maggior dettaglio: in *JniFilterWrapper* viene mantenuta una variabile di tipo long denominata *pointer*; quando viene chiamato il costruttore di *JniFilterWrapper,* questo richiama il metodo *init()* di *filterAdapter*; *filterAdapter* istanzia quindi un oggetto di tipo *filter* e ne restituisce il puntatore a *JniFilterWrapper* che lo memorizza in *pointer*. Quando *JniFilterWrapper* deve accedere ad uno dei metodi di *filterAdapter,* esso inserisce tra i parametri la variabile *pointer*, *filterAdapter* può quindi usare *pointer* perché rappresenta un indirizzo di memoria corretto in C++ per accedere all'istanza di *filter*.

3.2.2.3 Service

Il service realizzato per questa applicazione si chiama *PlayerService* ed è un service di tipo misto *started* e *bound*. Come abbiamo visto nel paragrafo 1.1.2.2, un servizio può infatti essere avviato con *startService()* e successivamente subire il *bind*. *FilterPlayerActivity*, la main activity dell'applicazione, avvia il service con *startService()* al primo avvio, quindi effettua l'unbind quando viene messa in stop nel metodo *onStop()* e il bind quando viene riattivata nel metodo *onStart()*; il codice di questi metodi è descritto nella sezione 4.1.6. Per eseguire il bind al service si usa l'istanza *myPlayerConnection* della classe *ServiceConnection* (codice al paragrafo 4.1.7); la classe prevede i metodi *onServiceConnected()* e *onServiceDisconnected().* Il primo viene richiamato non appena sia

avvenuta la connessione al servizio e viene utilizzato per ottenere le informazioni utili ad aggiornare l'interfaccia grafica; l'altro viene richiamato qualora la connessione al servizio dovesse venire a mancare inaspettatamente.

PlayerService viene avviato eseguendo il metodo *onStartCommand()*. In questo metodo viene inizializzato il decoder, viene costituita la catena di filtri, viene inizializzato l'audio di Android usando *AudioTrack* in modalità *stream* e, infine, viene notificato l'avvenuto avviamento del servizio inviando un messaggio broadcast con *sendBroadcast()*.

Per permettere il bind, all'interno di *PlayerService* è definita la classe *PlayerBinder* che estende *Binder* ed espone il metodo *getService()*. Tale metodo viene utilizzato da *FilterPlayerActivity* per ottenere un riferimento all'istanza di *PlayerService*. Il codice è riportato nel paragrafo 4.1.9.

Dopo aver effettuato il bind ed ottenuto un riferimento all'istanza di *PlayerService*, *FilterPlayerActivity* può avviare, fermare o mettere in pausa l'audio utilizzando i metodi *play()*, *stop()* e *pause()* di *PlayerService* (codice nel paragrafo 4.1.10).

L'effettiva elaborazione dell'audio viene eseguita all'interno del *Runnable PlayAudio*. Il metodo *run()* si occupa di leggere i campioni usando il metodo *read()* del decoder, elaborarli usando il metodo *elaborate()* della catena di filtri e inserirli nel buffer di *AudioTrack*. Il codice è riportato nel paragrafo 4.1.11.

3.2.3 Prestazioni di FilterPlayer – filtri in Java e in C++

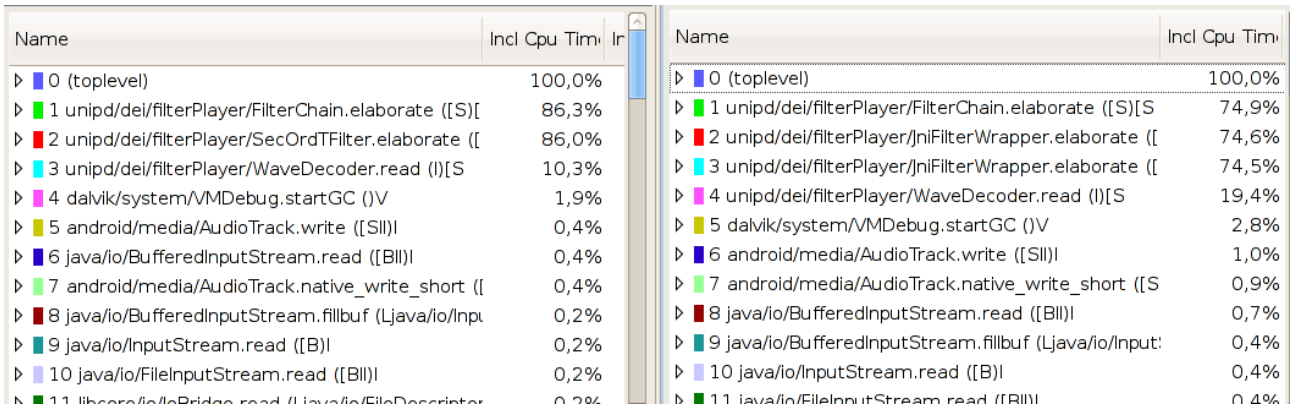

Illustrazione 24: Confronto delle prestazioni dell'implementazione del filtro in Java (sinistra) e in C++ (destra)

Quando, durante il tirocinio, si è iniziato a pensare di spostare l'implementazione del filtro in seconda forma trasposta da Java a C++, ciò che più preoccupava era la possibilità che la grande mole di dati (i campioni) passata tra Java e C++ potesse inficiare l'aumento di prestazioni ottenibile

programmando in codice nativo. In realtà, come anticipato nel paragrafo 1.1.4.1, proprio durante lo sviluppo di *FilterPlayer* si è constatato che sui dispositivi utilizzati durante il tirocinio (versioni Android 2.2 e 3.2) è possibile accedere per riferimento agli array Java utilizzando il codice nativo. Nonostante ciò, non è stato possibile ottenere un notevole incremento di prestazioni dei metodi di filtraggio. L'Illustrazione 24 confronta la distribuzione del tempo di CPU tra i metodi; l'immagine di sinistra riguarda il filtro in Java mentre quella di destra il filtro in C++. Si può notare come la riduzione sia stata solo del 10% a fronte di un importante incremento nella complessità dell'applicazione.

3.2.4 Conclusioni

Durante lo sviluppo di *FilterPlayer* è stato constatato come, nel nostro caso, l'uso della JNI, abbia migliorato solo marginalmente le prestazioni dell'applicazione. Le prestazioni generali dell'applicazione sono piuttosto deludenti; si pensi che sull'Archos 32 (equipaggiato con un ARM Cortex A8 a 800MHz) l'esecuzione di un file WAVE e il filtraggio con il volume e il filtro passa-banda provoca un uso di CPU del 30-40% circa (la misurazione è stata effettuata usando *top*).

Sarebbe stato interessante utilizzare l'API fornita da OpenSL ES per eseguire i campioni elaborati direttamente dal codice nativo. Si sarebbero potuti quindi portare anche *WaveDecoder* e *FilterChain* in codice nativo. Ciò avrebbe permesso di spostare tutta la catena audio in codice nativo e di valutare se in questo caso l'incremento di prestazioni fosse sostanziale. Purtroppo nel corso del tirocinio è stato affrontato un vasto numero di argomenti e non è stato possibile effettuare anche questo tipo di esperimento, che avrebbe introdotto ulteriore complessità nell'applicazione.

Capitolo 4: Appendice

4.1 Sorgenti di FilterPlayer

In questa sezione viene riportata la parte più significativa dei sorgenti di *FilterPlayer*.

4.1.1 Il metodo readHeader() di WaveDecoder

```
    public int readHeader() throws IOException, DataFormatException {
            int bytesRead = 44;
            ByteBuffer buffer = ByteBuffer.allocate(HEADER_SIZE);
            buffer.order(ByteOrder.LITTLE_ENDIAN);

            // Read Header
            input.read(buffer.array(), buffer.arrayOffset(), buffer.capacity());

            buffer.rewind();
            // Jump to audio Format
            buffer.position(buffer.position() + 20);
            int format = buffer.getShort();
            if (format != 1) // 1 means linear PCM
                  throw new DataFormatException("Invalid data format, only linear PCM
is supported.");
            // get channels configuration
            channels = buffer.getShort();
            if (channels == 1)
                  channelsConfiguration = AudioFormat.CHANNEL_CONFIGURATION_MONO;
            else if (channels == 2)
                  channelsConfiguration = AudioFormat.CHANNEL_CONFIGURATION_STEREO;
            else
                  throw new DataFormatException("Invalid channel configuration, only
mono and stereo is supported.");
            // get sampling rate
            frequency = buffer.getInt();
            if (frequency != 44100)
                  throw new DataFormatException("Only 44100 is a suitable frequency.");
            // jump to bits per sample
            buffer.position(buffer.position() + 6);
            int bits = buffer.getShort();
            if (bits == 16)
                  bytesPerSample = 2;
            else
                  throw new DataFormatException("Only 16 bit samples are supported.");

            // Search for data
            while (buffer.getInt() != 0x61746164) { // "data" marker
                  Log.d(TAG, "Skipping non-data chunk");
                  int size = buffer.getInt();
```

```
                input.skip(size);

                // Load 2 integers
                buffer.rewind();
                input.read(buffer.array(), buffer.arrayOffset(), 8);
                bytesRead += 8;
                buffer.rewind();
            }

            dataSize = buffer.getInt();
            if (dataSize < 0)
                throw new DataFormatException("Invalid data size in file header.");

            readSamplesCount = 0;
            totalNumberOfSamples = dataSize / (bytesPerSample);

            isHeaderRead = true;

            headerLength = bytesRead;
            return bytesRead;
        }
```

4.1.2 Il metodo read() di WaveDecoder

```
public short[] read(int nOfSamples) throws IOException, DataFormatException {
        // check header
        if (!isHeaderRead)
                readHeader();

        // check if it is the end of the file
        nOfSamples = Math.min((totalNumberOfSamples - readSamplesCount), nOfSamples);

        // reduce GC
        if (!areArraysInitialized || audioData.length != nOfSamples) {
                audioData = new short[nOfSamples];
                tempAudioData = new byte[nOfSamples * 2];
                areArraysInitialized = true;
        }

        input.read(tempAudioData, 0, nOfSamples * 2);

        for (int i = 0, j = 0; i < nOfSamples; i++, j += 2) {
                // http://www.peterfranza.com/2008/09/26/little-endian-input-stream/
                audioData[i] = (short) ((tempAudioData[j + 1] & 0xff) << 8 |
(tempAudioData[j] & 0xff));
        }

        readSamplesCount += nOfSamples;
        return audioData;
}
```

4.1.3 Il metodo read() di MicDecoder

```
public short[] read(int nOfSamples) throws IOException, DataFormatException {
        if (isRecording == false) {
                recorder.startRecording();
                isRecording = true;
                audioData = new short[nOfSamples];
        }
```

```
	if(audioData.length!=nOfSamples)
		audioData = new short[nOfSamples];

	// read is a blocking method
	recorder.read(audioData, 0, nOfSamples);

	return audioData;
}
```

4.1.4 La classe FilterChain

```
package unipd.dei.filterPlayer;

import java.util.ArrayList;

public class FilterChain {
	private ArrayList<Filter> filters = new ArrayList<Filter>();

	public void addFilter(Filter filter) {
		filters.add(filter);
	}

	public short[] elaborate(short[] samples) {
		for (Filter filter : filters) {
			samples = filter.elaborate(samples);
		}
		return samples;
	}

	public ArrayList<Filter> getFilters() {
		return filters;
	}
}
```

4.1.5 Il metodo elaborate() di filter

```
jshortArray filter::elaborate(JNIEnv* env, jobject obj, jshortArray samples, jint
nOfSamples) {
	jboolean isCopy=false;
	jshort *samplesPointer = env->GetShortArrayElements(samples, &isCopy);
	short x;
	double y;
	int i, j;
	for (i = 0; i < nOfSamples; i++) {
		//memorize actual sample
		x = (short) samplesPointer[i];
		//calculate actual output
		y = localB[channelIndex][0] * x;

		//calculate memories
		if (calculateMemories) {
			y = (y + s[channelIndex][0]) * gain[channelIndex];
			for (j = 1; j < myNOfParameters - 1; j++) {
				s[channelIndex][j - 1] = localB[channelIndex][j] * x
					- localA[channelIndex][j] * y + s[channelIndex][j];
			}
			// Calculate last memory element, j points to last item
			s[channelIndex][j - 1] = localB[channelIndex][j] * x
```

```
                                                - localA[channelIndex][j] * y;
                } else {
                        y *= gain[channelIndex];
                }
                //write result
                samplesPointer[i] = (jshort) y;

                // Calculate channel index
                channelIndex++;
                if (channelIndex >= channels)
                        channelIndex = 0;
        }
        env->ReleaseShortArrayElements(samples, samplesPointer, 0);
        return samples;
}
```

4.1.6 I metodi onStart() e onStop() di FilterPlayerActivity

```
@Override
public void onStart() {
        super.onStart();
        // register broadcast receiver
        registerReceiver(myBroadcastReceiver, new
IntentFilter(PlayerService.BROADCAST_ACTION));
        // if service is started, bind to it, if not wait with the broadcast
        // receiver defined over there.
        if (PlayerService.getIsStarted()) {
                bindService(playerServiceIntent, myPlayerConnection,
Context.BIND_AUTO_CREATE);
        }
}

@Override
public void onStop() {
        super.onStop();
        unregisterReceiver(myBroadcastReceiver);
        // Unbind from the service.
        if (isBoundToService) {
                unbindService(myPlayerConnection);
                isBoundToService = false;
                if (isStopped && hasStartedService)
                        stopService(playerServiceIntent);
        }
}
```

4.1.7 Definizione dell'istanza myPlayerConnection in FilterPlayerActivity

```
private ServiceConnection myPlayerConnection = new ServiceConnection() {
        /**
         * this bounds to the service and is called by system when binding is
         * done The system calls this to deliver the IBinder returned by the
         * service's onBind() method.
         */
        @Override
        public void onServiceConnected(ComponentName name, IBinder service) {
                PlayerBinder binder = (PlayerBinder) service;
                myPlayerService = binder.getService();
                isBoundToService = true;

                // If it was playing, probably user changed file, and want to play
```

```
                // the new track
                if (isPlaying)
                        myPlayerService.play();

                // get informations from service in order to update user interface
                filePath = myPlayerService.getPath();
                isPlaying = myPlayerService.getIsPlaying();
                isStopped = myPlayerService.getIsStopped();
                isLooping = myPlayerService.getIsLooping();
                isBandPassFilterOn = myPlayerService.getIsBandPassFilterOn();
                isPlayingMic = myPlayerService.getIsPlayingMic();

                updateUI();
                Log.d(TAG, "service bound");
        }

        /**
         * The Android system calls this when the connection to the service is
         * unexpectedly lost, such as when the service has crashed or has been
         * killed. This is not called when the client unbinds.
         */
        @Override
        public void onServiceDisconnected(ComponentName name) {
                isBoundToService = false;
        }
};
```

4.1.8 Parte del metodo onStartCommand di PlayerService

```
/**
 * Called when starting the service with "startService()" Receive as Extras:
 * path to file, bufferSize in seconds, volume in percent.
 */
@Override
public int onStartCommand(Intent intent, int flags, int startId) {
        // Save intent to use on looping
        savedIntent = intent;
        // intent could be null if there are no pending command for the
        // service after a system-kill
        if (intent != null) {
                // Check if it is a fresh start
                if (!isInitialized) {
                        [...]
                        // Initialize Decoder
                        if (!isPlayingMic) {
                                decoder = new WaveDecoder(path, readStreamBufferSize);
                                // Notify user for decoder parameters
                                Toast.makeText(getApplicationContext(), ((WaveDecoder)
decoder).fileInfo(), Toast.LENGTH_LONG).show();
                        } else {
                                decoder = new MicDecoder(1, readStreamBufferSize,
FilterPlayerActivity.SAMPLE_RATE);
                        }
                        [...]
                        / Initialize audio system
                        sampleRate = decoder.getFrequency();
                        outputChannels = decoder.getChannelsConfiguration();
                        outputEncoding = decoder.getBytesPerSample();
```

```
                    nOfChannels = decoder.getChannels();

                    // Erase filter chain
                    filterChain = new FilterChain();

                    // Volume filter
                    filterChain.addFilter(JniFilterWrapper.makeVolumeFilter(volume,
nOfChannels));
                    filterChain.addFilter(new
JniFilterWrapper(bandPassFilterBs.getParameters(), bandPassFilterAs.getParameters(),
nOfChannels));
                    // Initialize android's audio
                    // Get minimum buffer size
                    audioBufferInBytes = AudioTrack.getMinBufferSize(sampleRate,
outputChannels, outputEncoding);
                    // Initialize audio
                    androidPlayer = new AudioTrack(AudioManager.STREAM_MUSIC, sampleRate,
outputChannels, outputEncoding, audioBufferInBytes, AudioTrack.MODE_STREAM);
                    isInitialized = true;
                    }
             } else
                    isInitialized = false;

       // Other activities should Bind
       isStarted = true;
       Log.d(NAME, "Service started with onStartCommand()");
       // Broadcast message that I'm started
       Intent broadcastMessage = new Intent(BROADCAST_ACTION);
       broadcastMessage.putExtra(FilterPlayerActivity.SERVICE_IS_STARTED, true);
       sendBroadcast(broadcastMessage);
       return START_REDELIVER_INTENT;
}
```

4.1.9 Codice che permette il bind su PlayerService

```
private final IBinder myBinder = new PlayerBinder();
public class PlayerBinder extends Binder {
       PlayerService getService() {
              return PlayerService.this;
       }
}

@Override
public IBinder onBind(Intent intent) {
       return myBinder;
}
```

4.1.10 I metodi play(), stop() e pause() di PlayerService

```
// These methods are public methods accessible after binding
public void play() {
       if (!isInitialized) {
              onStartCommand(savedIntent, 0, 0);
       }
       if (isInitialized && !isPlaying) {
              isStopped = false;
              isPlaying = true;
              androidPlayer.play();
              Thread t = new Thread(new PlayAudio());
              t.start();
```

```
	}
}

public void pause() {
	if (isInitialized && isPlaying && !isStopped) {
		androidPlayer.pause();
		isPlaying = false;
	}
}

public void stop() {
	if (isInitialized && !isStopped) {
		androidPlayer.stop();
		isStopped = true;
		isPlaying = false;
		// stop
		try {
			decoder.stop();
			// if recording stop encoder
			if (isRecOn)
				encoder.finish();
			totalPlayedSamples = 0;
			isInitialized = false;
			androidPlayer.release();
		} catch (IOException e) {
			Toast.makeText(this, FilterPlayerActivity.FILE_NOT_FOUND,
Toast.LENGTH_SHORT).show();
		}
	}
}
```

4.1.11 Il Runnable PlayAudio in PlayerService

```
private class PlayAudio implements Runnable {
	@Override
	public void run() {
android.os.Process.setThreadPriority(android.os.Process.THREAD_PRIORITY_URGENT_AUDIO);
		int samplesToGet = audioBufferInBytes / outputEncoding;
		short[] audioData;
		while (!isStopped && isPlaying) {
			audioData = new short[samplesToGet];
			// read samples
			try {
				audioData = decoder.read(samplesToGet);
				// Check if we are at the end of the track, so we
				// have less samples to play
			} catch (IOException e) {
				e.printStackTrace();
			} catch (DataFormatException e) {
				e.printStackTrace();
			}
			// elaborate samples
			audioData = filterChain.elaborate(audioData);
			// write data
			int writedSamplesCount = androidPlayer.write(audioData, 0,
audioData.length);
			// if rec is on Encode audio to output
			if (isRecOn && !isPlayingMic) {
```

```
                        try {
                                encoder.write(audioData);
                        } catch (IOException e1) {
                                e1.printStackTrace();
                        }
                }
                totalPlayedSamples += writedSamplesCount;
                if (!isPlayingMic) {
                        // check if we are at the end of the track, so we have to
                        // stop or to loop.
                        if (totalPlayedSamples >=
((WaveDecoder)decoder).getTotalNumberOfSamples()) {
                                if (isLooping) {
                                        // reset decoder and restart
                                        try {
                                                decoder.stop();
                                                Log.d(NAME, "Stop decoder");
                                                totalPlayedSamples = 0;
                                        } catch (IOException e) {
                                                e.printStackTrace();
                                        }
                                        onStartCommand(savedIntent, 0, 0);
                                        Log.d(NAME, "Reload decoder");
                                } else {
                                        // stop
                                        stop();
                                }
                        }
                }
        }
    }
}
```

Bibliografia

Bornstein, 2008: Dan Bornstein, 2008 Google I/O Session Videos and Slides, 2008, https://sites.google.com/site/io/dalvik-vm-internals/

Fantozzi, 2011: Prof. Carlo Fantozzi, Slide del corso di Sistemi Embedded, 2011, http://es1011.wikispaces.com/

Android Guide: The Developer's Guide, http://developer.android.com/guide

Android SDK - NDK: What is the NDK?, http://developer.android.com/tools/sdk/ndk/overview.html

Sheng Liang, 2002: Sheng Liang, The Java(TM) Native Interface Programmer's Guide and Specification, 2002, http://java.sun.com/docs/books/jni/html/titlepage.html

Kreidler, 2009: Johannes Kreidler, Programming Electronic Music in Pd, 2009, http://www.pd-tutorial.com/english/

libpd Wiki: libpd Wiki, https://github.com/libpd/libpd/wiki/libpd

Brinkmann e altri, 2011: Peter Brinkmann, Peter Kirn, Richard Lawler, Chris McCormick, Martin Roth, Hans-Christoph Steiner, Embedding Pure Data with libpd, 2011

Brinkmann, 2012: Peter Brinkmann, Making Musical Apps, 2012

PD for Android Wiki: PD for Android Wiki, https://github.com/libpd/pd-for-android/wiki

Wiki openFrameworks: Wiki openFrameworks, http://wiki.openframeworks.cc/index.php?title=Main_Page

Dan Wilcox, 2011: Dan Wilcox, ofxPd readme, 2011, https://github.com/danomatika/ofxPd/blob/master/README.markdown

Smith, 2007: Introduction to Digital Filters with Audio Applications, http://ccrma.stanford.edu/~jos/filters/

Blog "Mind The Robot": Android Audio: play a WAV file on an AudioTrack, http://mindtherobot.com/blog/580/android-audio-play-a-wav-file-on-an-audiotrack/

Blog "Software Evolved": Little Endian Input Stream, http://www.peterfranza.com/2008/09/26/little-endian-input-stream/