Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

Esecuzione di applicazioni su macchina virtuale android con esperimenti su ART

Tesi di Laurea in Architettura degli Elaboratori

Relatore:
Chiar.mo Prof.
Vittorio Ghini

Presentata da:
Fabio Roscioli

Sessione I
Anno Accademico 2013/2014

Introduzione

Il recente sviluppo delle tecnologie di telefonia mobile e l'aumento incredibile di strumenti come *tablet* e *smartphone* e *smart TV* hanno avvicinato maggiormente le tecnologie informatiche e più precisamente quelle che fanno uso di *internet* all'utenza. La capacità di poter reperire qualunque tipo di informazione e notizie tramite *internet* è sempre più alla portata anche di chi non ha mai usato uno strumento informatico prima d'ora e questo ne ha aumentato a dismisura l'utenza. L'aumento di utilizzo dei programmi di VoIP mostra quanto sia cambiata l'importanza e la qualità dell'accesso alle risorse di rete da parte dell'utenza in quanto la tecnologia VoIP consente di instradare conversazioni vocali attraverso qualunque rete basata su controllo IP, comportando dunque un abbattimento dei costi rispetto ad un contratto telefonico tradizonale ed aprendo la possiblità di servizi avanzati come conversazioni video e teleconferenze. I limiti principali riscontrati attualmente in questo contesto riguardano la qualità del servizio, necessaria per una conversazione soddisfacente, e la mobilità del terminale.

All'interno del presente lavoro verrà mostrato un meccanismo di virtualizzazione che possa dare supporto alle applicazioni VoIP nelle operazioni di accesso alla rete continuando il porting di UMView. UMView permette di superare alcuni problemi della connessione di rete dei software VoIP consentendo l'utilizzo contemporaneo e personalizzabile, all'interno di un terminale

mobile, di differenti interfacce di rete; in questo modo sarà possibile ottenere, all'interno del dispositivo stesso, una specifica gestione del flusso di dati prodotto dall'applicazione.

Capitolo 1 si farà una panoramica di deversi modelli di virtualizzazione e dei criteri con i quali questi siano divisi per descrivere quelli utilizzati e si parlerà dell'ambiente di sviluppo dell'applicazione.

Capitolo 2 si parlerà degli strumenti utilizzati dando di loro una breve descrizione.

Capitolo 3 si descriverà il progetto ViewOS di cui UMView fa parte parlando dei lati comuni e dando una visione d'insieme del progetto.

Capitolo 4 si descriverà gli obiettivi della tesi.

Capitolo 5 si descriverà la progettazione del porting di UMView evidenziandone le problematicitità ed eventuali metodi per risolvere questi problemi.

Capitolo 6 si parlerà dei test effettuati, dei criteri e della valutazione del progetto.

Capitolo 7 si mostrerano le conclusioni a cui si è giunti al termine del lavoro.

Indice

Capitolo 1

Ambito di sviluppo

1.1 Macchine virtuali

1.1.1 Introduzione

In questo capitolo verrà esaminato il concetto di virtualizzazione in informatica e di come questo sia il riferimento alla possibilità di astrarre una risorsa, creando un'interfaccia esterna che nasconda la parte sottostante e ne permetta l'accesso concorrente da parte di pi istanze che funzionano in contemporanea. Dunque immaginando di aver uno strumento caratterizzato da un insieme di operazioni offerte, la sua virtualizzazione consiste nel fornire la sua interfaccia ma che potrebbe avere forma e funzionamento differenti.

1.1.2 Categorie e classificazione

Le macchine virtuali possono essere categorizzate rispetto a diverse esigenze che le distinguono le une dalle altre. Gli aspetti fondamentali di questa distinzione sono tre:

1. Come avviene comunicazione tra le macchine virtuali e le macchine reali sottostanti

   - **System call trapping**: Tutto il codice utente che viene eseguito sul processore reale ma le *System Call* sono reindirizzate alla macchina virtuale. Questa tipologia di macchina virtuale ha delle performance molto alte ma non è possibile eseguire codice compilato per architetture diverse da quella ospite.
   - **Processor Architetture Virtualization**: L'intera esecuzione è virtualizzata, è più lenta della precedente ma può eseguire codice compilato per diverse architetture Hardware.

2. Complessità della virtualizzazione, cioè quanto dell'*hardware* reale viene emulato dalla macchia virtuale. Le tipologie sotto elencate sono le varie possibilità che una macchina virtuale ci può presentare.

   - **Processor Virtualization only**: solo il processore è virtualizzato. Il sistema operativo *host* garantisce l'accesso a tutte le risorse del sistema.
   - **Partial Virtual Machine**: I programmi sono eseguiti sulla CPU fisica. Qualche parte del sistema è virtualizzato senza dover eseguire un sistema operativo *guest*.
   - **Complete System Virtualization**: Tutta l'architettura (processore e periferiche) sono virtuali. C'è la necessità di un sistema operativo *guest* per far eseguire la macchina virtuale.

3. "*Invasivity*" sul computer *Host*: quanto il sistema operativo reale sia influenzato dall'esecuzione di quello virtuale.

- **User-Level**: può essere implementata ed utilizzata usando i permessi standard dell'utente.
- **Root**: l'installazione necessita l'accesso a livello *root* sulla macchina *host*
- **Kernel Patch**: Il kernel deve essere modificato in qualche modo, sia tramite una patch che inserendo dei moduli.

In base alle categorie precedentemente elencate, a seconda del livello di astrazione, è possibile distinguere le macchine virtuali di processo e le macchine virtuali di sistema.

1.1.3 Macchine virtuali di Sistema o Macchine virtuali Totali

Questa tipologia di macchina virtuale si basa sull'emulazione totale di un sistema operativo e di un *hardware* da parte di un processo. Il suo scopo è eseguire un sistema operativo completo (detto sistema *guest*) e su di esso lanciare programmi applicativi. Questa forma di macchina virtuale solo in alcuni casi è necessaria, nella nostra tesi l'abbiamo usata per emulare il sistema *Android* e per l'esattezza un cellulare tramite AVD[1] che è una macchina virtuale fornita dalle Andoid *SDK* proprio per tale scopo. Tali macchine virtuali risultano essere molto comode in tali frangenti.

1.1.4 Macchine virtuali di processo o legate all'esecuzione di processi

Una macchina virtuale di processo, qualche volta chiamata Managed Runtime Environement (MRE), è eseguita come una normale applicazione all'interno del sistema operativo *host*. Viene creata quando il processo è creato e distrutta quando esce. Il suo scopo è di dare una piattaforma indipendente di ambiente di sviluppo che astragga dai dettagli dell'*hardware* o del sistema operativo e che permetta al programma di eseguire nello stesso modo su ogni piattaforma. Una macchina virtuale di processo fornisce un alto livello di astrazione e rende possibile programmare in linguaggi ad alto livello[2]. Le macchine virtuali sono implementate usando un interprete e il raggiungimento delle performance comparabile ai programmi in linguaggi nativi tramite l'utilizzo della compilazione Just-in-Time. L'esecuzione del codice quindi è

[1] *Android* Virtual Device

[2] Esempi : C# e Java

presa a carico dell'istanza della macchina virtuale, la quale si occuperà dell'interpretazione del codice e di interagire con il sistema operativo sottostante. Il sistema operativo *Android* si regge sull'utilizzo di due implementazioni della Java Virtual Machine, che è un classico esempio di questa tipologia di macchina virtual, quali sono la *Dalvik Virtual Machine* e *ART* che verranno trattate successivamente.

1.1.5 Macchine Virtuali a livello di processo e macchine virtuali parziali

Prima di descrivere direttamente questo tipo di macchine virtuali, è bene aprire una parentesi per descrivere la *System Call* sulla quale si basano, la comprensione de suo utilizzo è infatti importanti ai fini della comprensione dello svolgimento della tesi.

Syscall Ptrace [6]

Una *System Call* che si trova in molti sistemi operativi *unix* e *unix-like*. Usando la *ptrace* (il cui nome è un'abbreviazione di "process trace") un processo può controllarne un altro, abilitando il controllore ad ispezionare e manipolare lo stato interno del processo da lui controllato.la *System Call Ptrace* è usata spesso dai debuggers e altri tipi di strumenti per la code-analysis, la maggior parte sviluppati per aiutare la produzione di *software*.

Uso I debuggers (come *gdb* e *dbx*) la usano attraverso strumenti come *trace* e *itrace*, *ptrace* è anche utilizzata da programmi specializzati per patchare i programmi in esecuzione, per evitare bug non sistemati o per superare le feature di sicurezza. Può essere inoltre utilizzata come *sandbox*[4][5] e come ambiente di simulazione a runtime (come emulare accessi di root per *software* in user mode). Un processo controllandone un altro attraverso una chiamata di *ptrace*, ha un controllo esteso sulle operazioni svolte dal processo controllato tanto da poterne controllare:

- I *file descriptor*;
- La memoria;
- Le operazioni svolte;

- Le invocazioni delle *System Call* e il loro risultato;

- Gli *header* dei segnali;

La possibilità di scrivere nella memoria del processo controllato non solo permette di modificare i dati memorizzati nello stesso ma possono essere modificati anche i segmenti di codice del programma controllato. Il processo controllore di conseguenza è in grado di inserire dei *breakpoints* e modificare il codice in esecuzione del processo controllato. La possibilità di ispezionare e alterare gli altri processi è molto potente, *ptrace* è in grado di controllare un processo che può essere raggiunti dai segnali inviategli dal proprietario. Con il livello di accesso *superuser* si possono effettuare chiamate di *ptrace* per controllare quasi tutti gli altri processi presenti sul sistema.

Limitazioni Le comunicazioni tra il processo controllore e processo controllato avvengono tramite la ripetizione di chiamate di *ptrace*, passando un blocco di memoria di dimensione fissa tra i due; ciò è molto inefficiente quando si accede a un grande quantitativo di memoria del processo controllato. *Ptrace* provvede solo la pi basilare interfaccia necessaria a supportare i debuggers e strumenti simili. I programmi che ne fanno uso devono avere profonda conoscenza delle specifiche del sistema operativo e dell'architettura e sono responsabili a riguardo della comprensione e riassemblamento del codice macchina. Inoltre i programmi che inseriscono codice eseguibile nel processo controllato o (come *gdb*) permettono all'utente di inserire comandi che sono eseguiti nel contesto del processo controllato devono generare e caricare loro stessi il codice, generalmente senza l'aiuto del *program loader*.

Macchine virtuali a livello di processo

Una virtualizzazione è a livello di processo quando la macchina virtuale è direttamente interfacciata col processo. Questo significa che le richieste di servizi sono direttamente gestiti dalla macchina virtuale che implementa una interfaccia di alto livello.
Un esempio esaustivo di questa forma di macchina virtuale è **User-Mode Linux**[3]: L'obiettivo di questo programma è spesso introdurre nuovi elementi, genericamente non accessibili in un sistema reale. L'obiettivo di una macchina virtuale del genere, fattispecie UML[1][2], è quello di rendere possibile l'esecuzione di un kernel di Linux in user mode. Questo sistema è spesso usato per:

- il debug del kernel
- isolamento dei processi
- uso di diversi ambienti senza una macchina reale
- esperimenti in ambiente isolato dove l'utente può testare operazioni potenzialmente pericolose senza toccare il sistema operativo reale.

La virtualizzazione è ottenuta tramite la chiamata continua della *Syscall Ptrace*, che permette alla macchina virtuale di reindirizzare le invocazioni delle syscall da parte del processo ospite al fine di poterle eseguire tramite il kernel eseguito come processo in user-mode.

Macchine Virtuali parziali

Esistono anche delle macchine virtuali diverse da quelle precedentemente descritte, questa è una categoria di applicazione che non sono definibili come

[3]Indirizzo del sito web ufficiale : http://user-mode-linux.sourceforge.net

"*Macchine Virtuali*", ma comunque loro introducono un tipo di virtualizzazione. Questi sistemi non sono ambienti virtualizzati e non non si inseriscono completamente tra le applicazioni e il sistema operativo. L'obiettivo di questi strumenti è quello di introdurre nuovi elementi, genericamente non raggiungibili da un sistema operativo reale, per testare, *debuggare* e altri propositi. Come le macchine virtuali a livello di processo queste macchine virtuali utilizzano lo stesso meccanismo, cioè la chiamata alla *syscall ptrace* per dirottare le chiamate di sistema dei processi ospiti, però queste non emulano intero kernel bensì si limitano ad emulare solo alcune parti del sistema operativo come potrebbe essere la connessione di rete o l'accesso al *File System*.

Un esempio di tale macchina è quella che verrà utilizzata e spiegata in questa tesi, cioè Virtual Square's *ViewOS*.

1.2 *Android*

1.2.1 Introduzione

Android non è una piattaforma *hardware*, ma un ambiente *software* progettato appositamente per telefonini o pi in generale per dispositivi mobili. Basato sul kernel di linux con un'interfaccia utente basata sulla manipolazione diretta, disegnato principalmente per *device* mobili con *touchscreen* come *smartphone* e *tablet*, con varianti per televisioni e automobili. Nel 2011, *Android* ha il pi grande numero di sistemi installati di ogni sistema operativo per cellulari e nel 2013, i suoi device sono pi venduti di tutti i device Windows, iOS e Mac OS sommati tra loro [8, 9, 10, 11, 12]. Nel giungno 2013 il *Google Play Store* ha avuto pi di un milione di applicazioni pubblicate e pi di 50 miliardi di applicazioni scaricate[13]. Si attesta quindi come il sistema operativo pi utilizzato in assoluto nell'ambito dei *device* mobili e quindi come un ottimo settore di sviluppo del *software*.

1.2.2 Linux Kernel

Android consiste in un *kernel* basato sul *branch Long term support* del *kernel* di *Linux*. A gennaio 2014, la versione corrente di Android è relativa alla 3.4 o pi recente[14, 15], ma la versione specifica del kernel dipende dal *device* e dal *chipset*. Il *kernel* di *Linux* ha utilizzato da Android altri cambi di architettura che sono stati implementati da *Google* al di fuori dalle tipici cicli di sviluppo di *Linux*, come l'inclusione di componenti come *Binder*, *ashmen*, *pmen*, *logger*, *wekelocks* e differenti gestioni di tipo *out-of-memory*[19, 20, 21].

1.2.3 Dalvik Virtual Machine

È una macchina virtuale *open source* con compliatore *just-in-time* per l'esecuzione di *Dalvik Dex-code*, il quale è solitamente tradotto da codice *bytecode Java*. Questo è il *software* che esegue le applicazioni sui *device* Android. *La Dalvik Virtual Machine* è anche una parte integrante di Android, che è tipicamente usato su *device* mobili come telefoni e tablet, di recente è stato implementato in *device embedded* come *smart TV* e *media streamers*. Il *Java bytecode* è anche convertito in un gruppo di istruzioni alternative utilizzate dalla *Dalvik Virtual Machine*. Dalla versione 2.2 di Android la *Dalvik* presenta pure un compilatore *JIT*. Essendo ottimizzata per richiedere poca memoria, la *Dalvik Virtual Machine* ha qualche specifica caratteristica che la differenziano dalle Java Virtual Machine standard . Più in particolare, secondo specifiche richieste di *Google*, il *design* della permette ad un device di eseguire pi istanze della *Virtual Machine* efficientemente .

Processo Zygote Lo *Zygote* è un processo che contiene una *Dalvik Virtual Machine* vergine con solamente alcune librerie precaricate tra cui la libreria C *Bionic libc*[4]. Per creare un nuovo processo lo *Zygote* fa una *fork* di se stesso e poi specializza tale nuovo processo in base alle richieste del sistema. Per ricevere le richieste da parte del sistema nella fase di *boot* lo *Zygote* registra un *socket* per comunicare. Tale *socket* si trova in */dev/socket* ed ha il nome *zygote*. Tale socket */dev/socket/zygote* ha le seguenti proprietà:

1. Unix Domain Socket;
2. Stream;
3. Proprietà del root;

[4]è una versione customizzata e pi leggera delle *libc standard*

4. Permessi Linux 666 (lettura e scrittura da parte di tutti gli utenti);

1.2.4 ART: *Android* Runtime

Dalla versione 4.4 di Android è stato introdotto una nuova macchina virtuale chiamata *ART* . Questa usa un processo *ahead-of-time (AOT)* nel quale il bytecode è ricompilato in linguaggio macchina al momento dell'installazione. Al momento le performance di questa macchina non sono significativamente migliori di *Dalvik*, e qualche volta persino peggiori, ma i risultati sembrano essere molto dipendenti dai *banchmark* (forse può essere spiegato dal fatto che il lavoro di ottimizzazione della nuova *runtime* ancora non è iniziato) . Ci sono alcuni svantaggi come un tempo più lungo per installare le applicazioni, più che altro le applicazioni occupano dal 10% al 20% di spazio in più a causa del bytecode precompilato.

Google dichiarato che il *garbage collector* è stato modificato e migliorato su *ART* rispetto alla *Dalvik* ottimizzando la parallelizzazione dei processi durante l'esecuzione dello stesso. Google ha ha inoltre dichiarato di aver creato dei nuovi strumenti utili per il *debug* delle applicazioni su *ART*.

Capitolo 2

Strumenti di sviluppo

2.1 Configurazione dell'ambiente di sviluppo

I componenti necessari al fine di compilare correttamente il progetto e poter utilizzare le *NDK* di *Google* su *Linux* sono GNU Make 3.81, una recente versione di *Nawk* (GNU *Nawk* o N*Nawk*). Nella tesi di Bergami Si suggerisce per distribuzioni linux che usano aptitude di debian per l'installazione ed aggiornamento dei pacchetti ma pi specificatamente per Ubuntu l'installazione di alcuni pacchetti da cui il seguente comando[33] :

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

È importante soprattutto per le distribuzioni di linux installare la JDK di Oracle, esattamente la JDK 6 e 7. Senza di questa non è possibile compilare correttamente il codice da eseguire poi sul dispositivo *Android*.

2.1.1 *Android SDK*

L'*SDK* di *Android* è un insieme modulare di tools per lo sviluppo ed il debug delle applicazione in ambiente *Android*. L'*SDK* contiene anche degli emulatori per avere versione emulata di *Android* sul proprio computer e testare comodamente le proprie applicazioni senza rischio di danneggiare un eventuale *device*. Tra tutti i pacchetti dell'*SDK*, vi parlerò di due di questi specificatamente *ADB* e *AVD*.

2.1.2 *ADB - Android* Debug Bridge

L'*Android* Debug Bridge è un tool di linea di comando versatile che permette di comunicare con un'istanza dell'emulatore o con un *device Android* supportato. È un programma di tipo client server che include tre componenti:

- **Un client** che viene eseguito sulla macchina che viene utilizzata per sviluppare il software. Questi può essere invocato dal terminale con una linea di comando. Altri strumenti di *Android* come il plugin *ADT* di *Eclipse*, di cui parleremo in seguito, possono creare un client *ADB*.
- **Un server** che viene eseguito come processo di background sull'emulatore o sul *device* usato per i test.
- **Un demone** che esegue come un processo di background su ogni istanza di emulatore o *device*.

ADB permette delle operazioni molto comode per lo sviluppo di applicazioni. Tramite esso è infatti possibile avviare un terminale sul *device* o macchina virtuale usata per i test, inviando ad essa tutti i comandi di una normale *shell* di tipo *bash*. È anche possibile utilizzarla al fine di copiare file dalla macchina virtuale a quella reale e viceversa, ovviamente è possibile utilizzare le stesse potenzialità anche per i *device*. Tramite la *shell* di *ADB* si possono anche avviare le applicazioni che abbiamo compilato in C tramite l'*NDK* e le si può gestire come se fossero dei normali programmi con interfaccia di console.

Alcuni esempi di comandi utili di *ADB*

Alcuni esempi di utilizzo di questo programma sono i comandi:

```
adb push file_macchina destinazione_device
```

che serve ad inviare un file dalla macchina reale al *device*

```
adb pull file_\textit{device} destinazione_macchina
```

che serve ad inviare un file dal *device* (emulato o no) alla macchina reale.

```
adb forward tcp:#porta_\textit{device} tcp:#porta_macchina
```

quest'ultimo serve a inoltrare le porte dal *device* alla macchina virtuale, operazione necessaria se si vuole utilizzare un gdb server.

il più utile di tutti resta il seguente:

```
adb shell
```

che serve appunto ad avviare la *shell* dell'*ADB*.

2.1.3 *AVD - Android* Virtual Device

Un Androd virtual *device* è un configuratore dell'emulatore, permette di modellare una versione del *device* definendo opzioni riguardo l'*hardware* ed il software che devono essere emulati dall'emulatore di *Android*.

Un *AVD* consiste in:

- Un profile *hardware*: definisce le feature *hardware* del *device* virtuale. Per esempio si può definire se il *device* ha la videocamera, se usa una tastiera QWERTY fisica e quanta memoria ha a disposizione.

- Una mappatura dell'immagine di Sistema: è possibile definire quale versione di *Android* la piattaforma eseguirà sul *device* virtuale. E' possibile scegliere la versione della piattaforma standard di *Android* o l'immagine di sistema data con l'aggiunta di un ADD-ON dell'*SDK*.

- Un'altra opzione: puoi specificare la *skin* dell'emulatore che si preferisce usare, il che permette di controllare le dimensioni dello schermo,

l'aspetto e così via. Si può anche specificare quale SD card emulare per essere usata dall'ADV.

- Un'area di conservazione dei dati dedicata nella macchina di sviluppo: I dati utenti del *device* emulato (applicazioni installate, impostazioni, e via dicendo) e l'sd card emulate sono conservati in quest'area.

Basandoci sui tipi di *device* che si vuole modellare, è possibile creare tutte le versioni di *AVD* che si necessita. Per testare nel dettaglio le applicazioni, si dovrebbe creare un *AVD* per ogni generica configurazione di *device* con cui l'applicazione dovrebbe essere compatibile e testarla l'applicazione su ognuna di queste.

2.1.4 *Eclipse ADT*

Eclipse ATD è un tool plugin di sviluppo fornito gratuitamente per *Eclipse*, questo plugin garantisce una serie di features al fine di sviluppare e debuggare con l'accesso ad un IDE grafico le applicazioni per *Android*. è già disponibile nelle *SDK* una versione di *Eclipse* coi plugin già installati per sviluppare per *Android*.

Tramite *Eclipse ADT* è anche possibile avere un accesso grafico a gdb, evitando del tutto il dover utilizzare i comandi da console per controllarlo. Al fine di debuggare applicazioni *Android* è quindi opportuno configurare *Eclipse* per risparmiarsi del lavoro extra successivo che può far perdere tempo inutilmente.

Per fortuna l'*SDK* di *Android* già fornisce una versione già curata di *Eclipse* con tutti i plugin installati, alla quale nel nostro caso si dovrebbe solo installare i plugin riguardanti l'*NDK* che l'*SDK* non fornisce.

2.2 *NDK* - Native Development Kit

L'*NDK* è un *toolset* che permette di implementare parti delle proprie applicazioni usando il codice nativo come C e C++. Per certi tipi di applicazioni, questo dovrebbe essere utile così da poter riutilizzare il codice esistente e librerie scritte in questi linguaggi. *Google* stessa sconsiglia di scrivere applicazioni direttamente in codice nativo altresì ne consiglia il minimo uso necessario. I tipici buoni candidati consigliati da *Google* per l'*NDK* sono processi che chiedono intensivamente l'utilizzo della CPU come

- motori di videogiochi;
- emulazioni fisiche;
- elaborazione di segnali;

Nella pagina di presentazione stessa si sottolinea che i vantaggi per quanto riguarda le performance non sono così evidenti da poter considerare l'*NDK* un'opzione da prendere in considerazione senza aver valutato i pro e i contro e senza aver controllato che non ci sia già una API che potrebbe effettuare lo stesso lavoro.

2.2.1 Il Builder : Ndk-Build

ndk-build è un wrapper del comando *GNU Make* sviluppato per selezionare con le informazioni contenute nel file *Android.mk* e *Application.mk*, invocando lo script ndk-build corretto e aggiungendo i vari flag necessari per la compilazione. Il comando ndk-build dovrà essere lanciato da terminale direttamente nella cartella del progetto. La classica struttura di un progetto adattato per la compilazione con l'ndk-build è la seguente:

```
jni/     # contenente i file di configurazione
         # per ndk-build e i file per la compilazione
libs/    # conterra' le librerie compilate
obj/     # conterra' i vari eseguibili compilati
src/     # contiene i source file java
```

All'interno della cartella *jni* dovranno essere posizionati vari file quali Android.*mk*, *Application.mk* e i vari file sorgenti in linguaggio nativo del nostro progetto. Nello specifico, nel file *Application.mk* sarà esplicitata l'architettura per la quale sarà compilato il progetto e e nel file Android.*mk* l'ordine di compilazione dei file e le varie specifiche per essi.

2.2.2 Esempio di compilazione in *NDK*

Eseguibile scritto totalmente in C

Vi sono due modi per scrivere codice nativo per *Android*, il primo è quello di compilare codice nativo direttamente e renderlo eseguibile da *shell* tramite *ADB*, il secondo è quello consigliato da *Google* stessa: compilare il codice nativo come libreria dinamica e poi renderlo eseguibile tramite oppurtune chiamate *Java*. In questa tesi ho scelto di illustrare entrambi i metodi.

Inizio da un esempio semplice che riguarda un piccolo client che ho utilizzato per i test:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
```

```
#include <string.h>

#define SERVERPORT 3002
#define SERVERADDRESS "127.0.0.1"

int main(int argc, char *argv[]) {
int client_socket;
struct sockaddr_in server_addr;
char *msg;
size_t msglen;
int retcode;

// verifico la correttezza dei parametri
if (argc != 2) {
printf("Usage: %s \"message to send\"\n", argv[0]);
exit(-1);
}

// preparo la variabile msg in base alla dimensione del messaggio inserito dall'ut
msg = malloc(strlen(argv[1]) * sizeof(char));

// apertura socket del client
if ((client_socket = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
perror("Error in socket()");
exit(-1);
}
```

```
// preparazione dell'indirizzo del server con cui connettere il socket
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(SERVERPORT);
server_addr.sin_addr.s_addr = inet_addr(SERVERADDRESS);

// connessione del socket al server
if ((connect(client_socket, (struct sockaddr *) &server_addr,
sizeof(server_addr))) == -1) {
perror("Error in connect()");
exit(-1);
}

printf("Client connected to %s\n", SERVERADDRESS);

// invio messaggio al server
strcpy(msg, argv[1]);
msglen = strlen(msg) + 1;
retcode = write(client_socket, msg, msglen);

printf("Sent %d (%u request) bytes on socket %d\n", retcode,
(unsigned) msglen, client_socket);
}
```

Come si può vedere, è un semplice client *tcp/ip* che attende un segnale da parte dell'utente, e questo sarà utile per farlo controllare da *UMView* prima di partire e poi esegue la connessione al client.

```
LOCAL_PATH :=$(call my-dir)
```

```
# dichiara che stiamo entrando
# in una nuova compilazione , pulisce l' ambiente
include $(CLEAR_VARS)
# nome del file di output
LOCAL_MODULE := client
# files da compilare per ottenere il file di output
LOCAL_SRC_FILES := Client.c
# flag da passare al compilatore
LOCAL_CFLAGS :=-g
# dichiara l' output come eseguibile ,
# prepara i giusti CFLAGS
# poi compila con il compilatore
# assegnato prima nel file Application.mk
include $(BUILD_EXECUTABLE)
```

per farlo compilare ho scritto il seguente file Android.*mk*. È importante considerare che la linea di codice

```
include $(BUILD_EXECUTABLE)
```

serve per informare il compilatore, *ndk-build*, di compilare un'eseguibile. A seconda che lì sia presente l'opzione $BUILD_EXECUTABLE$ o $BUILD_SHARED_LIBRARY$, verrà compilato un eseguibile oppure una libreria condivisa.

Per questo genere di programma, che serve per essere esclusivamente scritto in C ed eseguito solo da *shell*, è necessaria che il file *Application.mk* contenga le specifiche di architettura e la versione dell'API di *Android* di riferimento.

```
# specifica che stiamo
```

```
# sviluppando per processori ARMs
APP_ABI := armeabi-v7a
# specifica che stiamo
# sviluppando per la versione 19 di android
APP_PLATFORM := android-19
```

In questo caso specifico, è stato sceltoo di compilare per la versione 19 dell'API di *Android* che corrisponde alla versione 4.4.2 di *Android* in quanto è la prima a dare la possibilità di usare anche ART e quindi è sembrata una scelta opportuna per evitare problemi di compatibilità.

Esempio di applicazione *Android* con integrazione di libreria in linguaggio nativo

Come già detto questo vale per quanto riguarda il codice nativo che deve essere eseguito direttamente da *shell* per *Android*, ma l'*Android NDK* non è stato progettato per questo espressamente. È stato invece progettato per integrare piccole librerie in linguaggio nativo nei programmi scritti in JAVA e compilati per *Android* quindi vi propongo il seguente esempio pubblicato ufficialmente dall'*Android* Open Source Project:

```
/*
 * Copyright (C) 2009 The \emph{Android} Open Source Project
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
```

```
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 *
 */
#include <string.h>
#include <jni.h>

/* This is a trivial JNI example where we use a native method
 * to return a new VM String. See the corresponding Java source
 * file located at:
 *
 *   apps/samples/hello-jni/project/src/com/example/HelloJni/HelloJni.java
 */
jstring
Java_com_example_hellojni_HelloJni_stringFromJNI( JNIEnv* env,
                                                  jobject thiz )
{
    return (*env)->NewStringUTF(env, "Hello from JNI !");
}
```

Questo è un codice fornito dagli esempi di *Android* che mostra esattamente come può una libreria scritta in linguaggio nativo interfacciarsi con un programma scritto in alto livello. In questo caso abbiamo necessità di Android*Manifest.xml* nella directory principale nel quale saranno scritti i

permessi che *Android* garantirà al processo e l'architettura di riferimento oltre che la versione delle API.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
package="com.example.hellojni"
android:versionCode="1"
android:versionName="1.0">
<uses-sdk android:minSdkVersion="3" />
<application android:label="@string/app_name"
android:debuggable="true">
<activity android:name=".HelloJni"
android:label="@string/app_name">
<intent-filter>
<action android:name="android.intent.action.MAIN" />
<category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
</activity>
</application>
</manifest>
```

in questo caso il file Android.*mk* sarà scritto nella seguente maniera:

```
# Copyright (C) 2009 The \emph{Android} Open Source Project
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
```

```
# http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
#
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE := hello-jni
LOCAL_SRC_FILES := hello-jni.c
include $(BUILD_SHARED_LIBRARY)
```

Al fine di richiamare la libreria scritta in C, saranno necessarie alcune specifiche nell'activity scritta in java:

ed il file *Application.mk* sarà semplicemente:

```
APP_ABI := all
```

per indicare che la libreria dovrà essere compatibile con qualunque architettura.

Mentre in Java il file avrà alcune direttive specifiche:

```
/*
* Copyright (C) 2009 The \emph{Android} Open Source Project
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
```

```
* You may obtain a copy of the License at
*
* http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*/
package com.example.hellojni;
import android.app.Activity;
import android.widget.TextView;
import android.os.Bundle;
public class HelloJni extends Activity
{
/** Called when the activity is first created. */
@Override
public void onCreate(Bundle savedInstanceState)
{
super.onCreate(savedInstanceState);

TextView tv = new TextView(this);
tv.setText( stringFromJNI() );
setContentView(tv);
}
```

```
public native String stringFromJNI();

public native String unimplementedStringFromJNI();

static {System.loadLibrary("hello-jni");}
}
```

Come vediamo esiste una chiamata specifica per caricare la libreria

```
System.loadLibrary("hello-jni");
```

Il metodo

```
 static {System.loadLibrary("hello-jni");
```

serve a caricarela libreria hello-jni all'avvio dell'applicazione. La libreria è stata già stata spacchettata dentro la cartella adatta dal packet manager. Il metodo

```
public native String stringFromJNI()
```

serve invece per richiamare la funzione scritta in linguaggio nativo. I metodi di questo tipo si possono riconoscere dalla parola chiave che serve per identificarli *native* e come vuole dimostrare l'autore dell'esempio se ne possono creare quanti se ne vuole basta che facciano riferimento ad un metodo implementato nella libreria.

Con questi sistemi è possibile fare in modo di scrivere alcune parti delle applicazioni direttamente in linguaggio nativo e richiamarle comodamente da Java qualora fossero necessarie. Per fortuna molta parte di queste operazioni é già svolta automaticamente dall'*ADT* di *Eclipse* e quindi per il programmatore non dovrà essere necessario imparare buona parte di questi procedimenti.

2.3 Script setEnv.sh

Per l'esecuzione degli eseguibili come utente e dislocati in una cartella differente dalle classiche cartelle designate ai binari, è necessario impostare le varie variabili d?ambiente. Per semplicità Davide Berardi aveva scritto un semplice script di *bash* che ora ripropongo al lettore:

```
# !/ system / bin / sh
export LD_LIBRARY_PATH =/data/local/bin
export HOME =/data/local/bin
export PATH = $PATH :/data/local/bin
```

Lo script deve essere eseguito all'interno della *shell* desiderata specificandola come destinazione nel seguente modo:

```
. ./setEnv
```

Capitolo 3

ViewOS

ViewOS parte dall'idea che ogni processo dovrebbe poter avere la sua visione dell'ambiente di esecuzione: questo non significa che ogni processo deve essere eseguito in ambienti totalmente diversi, ma può essere utile mantenre nella loro visione un sottoinsieme del sistema reale, mentre parte di questo è virtualizzato. In particolare, un processo di *ViewOS* può ridefinire ogni comportamento delle *System Call* e definire nuove *System Call* e allo stesso tempo mantenere la sintassi originale. Ciò è possibile per eseguire eseguibili esistenti in diversi scenari, possibilmente gestire le loro *features*. In ultimo *ViewOS* dovrebbe essere visto come una macchina virtuale di processo configurabile e per utilizzo generico. *ViewOS* può essere implementato come modulo del *kernel* oppure come una *System Call* Virtual machine a livello utente. Un software per virtualizzare può dare al processo differenti viste intercettando le *System Call*s generate da ogni processo. Ogni *System Call* può essere valutata ed eventualmente gestita nel contesto in cui viene invocata. Tutti i metodi di interposizione delle *System Call* dovrebbero come essere visti come "applicazioni" della *ViewOS*. Una implementazione *System Call* virtual machine di *ViewOS* ha performance minori rispetto ad una im-

plementazione *kernel* ma una *System Call* virtual machine può essere essere eseguita a livello utente con permessi utente, e non necessita una completa riconfigurazione del sistem operativo reale per essere eseguita.

3.1 Obiettivi e applicazioni di *ViewOS*

Le possibili applicazioni di questa progetto, comode anche nello svolgimento di questa tesi, si basano sul suo concetto di fondo : agguiungere possibilità e flessibilità al sistema operativo sul quale questa viene lanciata. I seguenti punti servono per indicare esplicitamente a cosa ci si riferisce:

- **Ridefinizione delle *System Call***: in modo più generico cambiare la visione del sistema per un processo è modificare la sematica delle *System Call* che invoca. *ViewOS* permette di ridefinire totalmente il comportamento delle *System Call* e fornire nuovi risultati al processo.

- **Creazione di nuove *System Call***: in aggiunta alle *System Call* standard, è possibile definire nuove *System Call* per molte ragioni:

  - permettere ai processi di interagire con le nuove feature fornite dalla macchina virtuale;
  - un utente potrebbe voler usare *ViewOS* per ottenere la compatibilità con un software che gira su altre architetture e che usano un altro insieme di *System Call*;

  Lo stesso meccanismo usato per la ridefinizione delle *System Call* può essere usato per crearne nuove;

- **Compatiblità dei binari**: un impotante obiettivo di *ViewOS* è di raggiungere la compatibilità binaria con i software esistenti, quindi non

rendere necessario ricompilare il programma per riutilizzarlo dentro *ViewOS*, ma la virtualizzazione deve essere il più trasparente possibile. E questo è stato raggiunto tenendo la sintassi delle *System Call* intatta;

- **Utilizzo di servizi senza privilegi di amministratore**: *ViewOS* permette di superare il tradizionale limite imposto nei sistemi Unix-like riguardo i privilegi da garantire agli utenti. Grazie a *ViewOS* molte operazioni (come montare un *File System* da un disco) possono essere eseguite dagli utenti nella loro visione del mondo, senza mettere a rischio la stabilità del sistema e la sua sicurezza. *ViewOS* non esegue codice privilegiato: nel caso della mount, l'unico requisito è che l'utente abbia i diritti corretti per leggere il file che contiene l'immagine;

- **Modularità e componibilità**: *ViewOS* punta ad essere generica ed estensibile, è composta da un core centrale e molti moduli, ognuno dei quali implementa una virtualizzazione specifica. *ViewOS* è applicabile in molti contesti e casi. Alcune applicazioni di *ViewOS* sono:

  - Emulazione di operazioni che richiedono accessi privilegiati;
  - Sicurezza: l'isolamento di un processo server è una problematica nota per evitare di accedere ai dati protetti. *ViewOS* può sovrascrivere la visione del processo facendo in modo di nascondere alcune parti del *File System* e nel tempo stesso farne vedere altre. Un'altra applicazione interessante per quanto riguarda la sicurezza può essere l'esecuzione di software sconosciuto in un ambiente protetto ed isolato proteggendo le parti critiche del sistema;
  - Mobilità grazie ad IPv6 è possibile assegnare un indirizzo ip ad ogni processo oltre che alla macchina reae. Questo, combinato con l'uso dei programmi per la creazione di reti virtuali o altre reti

virtuali, può rendere il processo indipendente dalla configurazione di rete. Questo modo è posibile per bloccare i processi e farli riprendere successivamente, mantenendo la loro visione della rete intatta;

– Prototipi: Con *ViewOS* è possibile definire un inseme di network virtuali e visioni virtuali che permettono ai processi di comunicare che siano nella stessa macchina reale o in macchine reali differenti;

3.2 Implementazioni di *ViewOS*

L'idea di *ViewOS* è stata implementata più volte a seconda delle necessità incontrate dai programmatori e come questi volessero impostare il progetto stesso. Quindi abbiamo tre diversi progetti che condividono la stessa idea e buona parte della stessa logica:

1. *pure_libc* : questa non è un'implementazione di ViewOs specificatamente ma una libreria di supporto che è necessaria per implementare un'efficiente sovrascrittura delle *System Call*, perchè l'implementazione corrente delle lib C GNU non lo permettono.

2. *UMView*: questa è un'implementazione di ViewOs come una System Call Virtual Machine, più specificatamente una Macchina virtuale Parziale. *UMView* è implementata interamente nello user-space e non necessita di alcuna modifica al *kernel* in esecuzione nella macchina host. È basata sul meccanismo della pthrace() per tracciare le *System Call* invocate dai processi.

3. *KMView*: come la precedente, è una implementazione di *ViewOS* come una System Call Virtual Machine e più specificatamente una Macchina

Virtuale Parziale. Diversamente da *UMView*, dipende da un modulo del *kernel* di linux.

KMView e *UMView* condividono gli stessi principi, concetti e moduli ma differiscono solamente nel meccanismo ad alto livello.

3.2.1 *UMView*

UMView è un'implementazione di *ViewOS* come System-Call Virtual Machine o per essere più precisi come una Macchina Virtuale Parziale che fornisce lo stesso insieme di System Call della macchina ospite.

UMView, come già visto nel capitolo riguardo le macchine virtuali, utilizza la System Call pthrace per seguire e controllare il processo a seconda dei moduli caricati. Le System Call invocate dal processo verranno ridirette, qualora necessario, ai moduli di *UMView* ed eseguite secondo il moduli implementati nella macchina virtuale.

Modularità di *UMView*

UMView è divisa in due parti

1. il core
2. moduli

Il core si prende cura della cattura delle *System Call* e nel caso le ridireziona al modulo appropriato. Ogni modulo implementa un sottoinsieme di *System Call*, basate sulla virtualizzazione del servizio offerto. I moduli sono implementati nella forma di librerie dinamiche, caricate dall'utente al momento al momento dell'esecuzione. Ogni modulo definisce una scelta particolare di funzioni, usate per determinare se il modulo deve essere utilizzato

o no. In questo modo, dopo l'intercettazione di una *System Call*, il core deve decidere se la *System Call* deve essere gestita dalla macchina virtuale o ridirezionata al *kernel* reale e se nel caso in cui questa debba essere gestita dalla macchina virtuale, il core deve scegliere anche da qualche modulo farla gestire.

Utilizzo base di *UMView*

Ora mostreremo una serie di esempi per l'utilizzo base di *UMView*:

```
$umview bash
```

una volta che la macchina virtuale è stata eseguita, se non ci sono moduli caricati, il comando si comporta come se fosse stato eseguito fuori da *UMView*. Quando si carica una shell non interattiva all-interno di *UMView*, potrebbe essere utile precaricare i moduli (con l'opzione - *p*).

```
$umview -p umnet bash
```

un altro modo per caricare i moduli è il comando

```
$um_add_service nome_modulo
```

Questo comando, che deve essere lanciato quando il processo di *UMView* è già stato avviato, carica uno spefifico modulo nella catena dei servizi di *UMView*.

```
$um_ls_service
```

Questo comando ritorna uan lista dei moduli correntemente caricati nella macchina virtuale.

```
$um_del_service
```

Questo comando rimuove il servizio specificato dalla lista dei servizi: il modulo da rimuovere deve essere specificato dal suo nome come specificato da *um_ls_service* citato precedentemente.

```
$umshutdown
```

Questo comando è usato per chiudere la macchina virtuale corrente. Quando chiamato senza argomenti, invia un segnale di tipo TERM a tutti i processi in esecuzione nella macchina virtuale corrente, aspetta 30 secondi e allora invia il segnale di KILL a tutti i processi sopravvissuti.

```
$viewname
```

Questo comando configura e mostra il nome della corrente virtual machine, come nell'esempio qui sotto riportato.

```
$ viewname
$ viewname myview
$ viewname
myview
```

Il comando più interessante ai fini di questa tesi è

```
$vuname
```

vuname può essere usato invece di emphuname, ha la stessa opzione standard del comando unix. Nello specifico emphvuname -a mostra tre altri campi quando eseguito nella macchina virtuale *UMView*.

```
 $ uname -a
Linux v2host 2.6.22-viewos \#5 SMP Tue Jul 31 22:29:46 CEST 20
07 i686 GNU/Linux
$ vuname -a
Linux v2host 2.6.22-viewos #5 SMP Tue Jul 31 22:29:46 CEST 20
07 i686 GNU/Linux/View-OS 24410 0
$ viewname myview
$ vuname -a
```

```
Linux v2host 2.6.22-viewos #5 SMP Tue Jul 31 22:29:46 CEST 20
07 i686 GNU/Linux/View-OS 24410 0 myview
$ UMView bash
\textit{UMView}: nested invocation

$ vuname -a
Linux v2host 2.6.22-viewos #5 SMP Tue Jul 31 22:29:46 CEST 20
07 i686 GNU/Linux/View-OS 24410 1 myview
$ exit
$ vuname -a
Linux v2host 2.6.22-viewos #5 SMP Tue Jul 31 22:29:46 CEST 20
07 i686 GNU/Linux/View-OS 24410 0 myvie
```

Questo di sopra è un esempio del suo utilizzo.

Il nome del sistema operativo è cambiato da "GNU/Linux" a "GNU/Linux/View-OS", quindi c'è un un server id, cioè un pid del processo server di *UMView*, il nome della view e il numero della view.

3.3 Considerazioni

I vantaggi che il progetto *ViewOS* fornisce a chi vuole alterare il comportamento di programmi già esistenti senza doverne modificare il codice sorgente sono evidenti e come spiegato un sistema come quello di *ViewOS* permette un'ampia personalizzazione delle caratteristiche della macchina virtuale. La possibilità poi che si ha tramite *UMView* di avere a disposizione questa potente macchina virtuale tramite la quale è possibile gestire a livello utente le chiamate alla *System Call* rendono la scelta di questa implementazione obbligata. Avendo di fronte a noi la necessità progettuale di lavorare su

Android è necessario avere la possibilità di gestire i processi in modo tale che non sia necessario compilare diversamente le applicazioni che si vogliono ingabbiare nella macchina virtuale nè essere obbligati in qualche modo a dover caricare una macchina virtuale totale con tutto so spreco di risorse e quindi batteria che una scelta del genere porterebbe. Invece tramite *ViewOS* e *UMView* questi sprechi sono evitabili e anche facilmente superabili in quanto l'idea di fare in modo di non dover per forza gestire tutte le chiamate delle *System Call* ma soltanto quelle desiderate ne velocizzano la gestione.

Capitolo 4

Obiettivi

4.1 Preambolo : Porting di *UMView* su *Android* con processore *ARM*

Il lavoro di questa tesi si basa sulla tesi di *Davide Berardi* nella sua tesi "Porting della macchina virtuale *UMView* su sistema operativo *Android ARM*". In quella tesi veniva spiegato il funzionamento base di *UMView* e cosa è stato modificato per permetterne la compilazione sui processori *ARM*. Grazie a questo lavoro è stato possibile avere una versione stabile anche se non perfetta di *UMView* sulla quale si basa questa tesi. Questa versione è stata ottenuta tramite la modifica di *Bionic* da parte di Berardi e l'aggiunta di alcune modifiche parziali per arginare il problema della tracciabilità delle *System Call*. Il grande problema che Berardi non è riuscito a risolvere è quello relativo all'ingabbiamento delle applicazioni non lanciate direttamente dalla *shell* della macchina virtuale.

4.2 Obiettivi del progetto

L'obiettivo del progetto consiste nel continuare il lavoro di Davide Berardi nel porting della macchina virtual *UMView*. Questo perché? Perché il vero obiettivo del progetto è quello di riuscire a creare un gestore della rete al quale un processo possa far riferimento al fine di connettersi, in modo da avere un controllo completo sulla rete visibile da quel processo e poter programmare di conseguenza unico net device driver che scelga secondo alcuni criteri se sia più opportuno utilizzare un'interfaccia di rete oppure un'altra. I vari vantaggi del porting di questa macchina virtuale sta nel poter superare alcuni dei limiti imposti da *Android* alle sue applicazioni.

Alcuni esempi che sarebbe possibile realizzare tramite questo sistema:

1. un filtro anti-spam
2. un applicazione di registrazione delle chiamate
3. un *firewall*
4. switch dinamico dell'interfaccia di rete

Con l'avvento di *ART*, si è posto il problema se il porting precedentemente effettuato fosse anche compatibile con la nuova runtime di *Android* o se invece il cambio di tecnologie avesse reso nullo il lavoro precedentemente fatto. Per questo motivo è stato necessario effettuare test di utilizzo della nuova runtime con il porting.

4.3 Switch dell'interfaccia di rete dinamico

Un altra importantissima applicazione di *UMView* ad *Android* è lo switch dinamico dell'interfaccia di rete. Supponiamo di star camminando per strada

in una zona con diverse reti *wireless* alle quali il nostro device può avere accesso e supponiamo di star effettuando una telefonata tramite un programma di VoIP come possono essere *Skype* o *Viber*. Ogni qual volta ci si allontana dall'access point e si cambia connessione, la chiamata cade e non si è più in grado di continuare la conversazione senza rilanciare nuovamente la chiamata una volta che il programma si sarà riconnesso alla nuova rete wireless.

Tramite *UMView* potremmo impostare il processo di telefonia al fine di farlo connettere ad un gestore virtuale delle interfacce di rete e quindi fare in modo che l'applicazione VoIP non debba riconnettersi per continuare la telefonata, questo mentre il sistema operativo rieffettuarà la connessione al nuovo access point senza dare alcun tipo di problema di linea.

Con la versione "L"[1] di *Android*, pare che una delle possibilità nascoste sia il multi-networking[2] [3] per cui una chiamata *Skype* non dovrebbe interrompersi la differenza sarebbe che mentre questa di *UMView* sarebbe gestita dall'utente e quindi personalizzabile, invece quella di *Android* sarebbe gestita dal sistema operativo e quindi generica.

Lo *switch* dinamico tramite *UMView* sarà, però raggiungibile solamente una volta considerati anche i meccanismi riguardanti la perdita di connessione e riconnessione di *Android* e i meccanismi che regolano la gestione dei pacchetti che andremo ad introdurre e la gestione delle connessioni da parte dall'applicazione VoIP. Tutto ciò al fine di garantire l'operazione di smistamento dei pacchetti e delle interfacce di rete e rendere la comunicazione gestibile efficientemente da parte della macchina virtuale. Per questo motivo

[1] cioè la versione 4.5/5.0 di *Android*

[2] http://www.*Android*police.com/2014/07/01/*Android*-l-feature-spotlight-multi-networking-offers-seamless-network-switching-easy-wi-fi-accessories/

[3] http://www.tutto*Android*.net/*Android*/*Android*-l-visualizzera-la-frequenza-delle-reti-wifi-ed-avra-il-multi-networking-202960/

questa parte sarà gestita una volta che *UMView* sarà portata con successo su *Android*.

4.4 Problema riguardo l'intercettazione dei programmi lanciati da interfaccia grafica

L'obiettivo specifico di questa tesi è quello di considerare ed eventualmente superare il problema che Davide Berardi aveva trovato nel porting di *UMView*:

> "La DALVIK virtual machine 'e come gi'a introdotto una macchina virtuale legata all'esecuzione di processi che ha in carico l'intero front-end grafico di *Android*. Esiste quindi un problema relativo all'utilizzo che si vuole fare di *UMView* con il sistema di gestione della virtual machine java implementata in *Android*: Volendo, per esempio, lanciare un applicazione *Android*, la dalvik virtual machine sottostante lancia una piccola procedura interna chiamata zygote, la quale ha effetto solo di creare una nuova istanza della dalvik come nuovo thread, nascondendo l'esecuzione del nuovo thread al sistema operativo, essa entra quindi in conflitto con *UMView*, non essendo in grado di tracciare il suo tentativo di clonazione e non avendo quindi possibilit'a di tracciare qualsiasi programma presente all'interno dell'interfaccia grafica. L'unica plausibile soluzione 'e quindi riuscire a tracciare la Dalvik Virtual Machine principale, lanciandola quindi come processo figlio di *UMView*. Essa per'o 'e lanciata al boot del sistema operativo, quindi risulta l'unico metodo plausibile la modifica del file init.rc specificando quindi il lancio della Dalvik come figlia di *UMView*,

> e quindi ottenendo come risultato che ogni *System Call* lanciata dal processo venga dirottata dalla sottostante instanza di *UMView*. Questa soluzione per'o porta ancora 25 problemi dal punto di vista implementativo, pensiamo infatti uno scenario composto da un instanza di *Skype* ed un instanza del gestore dei messaggi, a questo punto vogliamo virtualizzare per *Skype* un ambiente virtuale differente da quello del gestore dei messaggi,dovremmo quindi formalizzare un metodo con il quale *UMView* sappia riconoscere i due thread particolari, visto che a questo punto ogni instanza della Dalvik 'e tracciata allo stesso modo."[4]

Come indicato da Berardi, durante il porting era stato trovato un problema riguardo l'ingabbiamento dei processi avviati dall'interfaccia grafica, come ad esempio poteva essere *Skype*. E l'obiettivo principale di questa tesi è stato trovare un modo per superare tali problemi e testare i risultati su entrambe le runtime[5] studiandone le cause ed eventualmente soluzioni.

[4]Davide Berardi - Porting della macchina virtuale *UMView* su sistema operativo *Android ARM*: pag.43

[5]Dalvik e ART

Capitolo 5

Progettazione

In questo capitolo ci occuperemo di come è stato effettuato il *porting*, di *UMView*, della struttura dell'applicazione e delle problematiche relative al suo sviluppo su *Android* oltre che ad alcune modalità suggerite per ovviare a tali limitazioni. Per testare il *porting*, è stata utilizzata l'*NDK* versione r9d rilasciata nel Marzo 2014, le prove sperimentali sono state effettuate sia attraverso Android *Virtual Device* sia direttamente su *Google Nexus 4* tramite Android *Debug Bridge*. La versione di *Android* è la Kitkat(versione 4.4.3) mentre l'architettura del processore è *ARM*.

5.1 Struttura dell'applicazione

La struttura del *porting*, di *UMView* è praticamente la stessa della versione originale per *Linux*,:

1. un *core* che si occupa della parte di cattura e ridirezionamento delle *System Call*

2. dei semplici programmi compilati in C che servono per essere utilizzati all'interno della macchina virtuale

3. una serie di librerie dinamiche che sono compilate separatamente dal progetto ma che condividono degli header con esso al fine di poter essere caricate dinamicamente, tali librerie si chiamano "moduli".

L'applicazione è incentrata sulla modularità e quindi anche la sua struttura ne risente, il suo *core* è dato da un eseguibile *umview* che contiene le procedure base della macchina virtuale.

5.1.1 Il comando "*mount*" in *UMView* e gestione dei File System

UMView utilizza la *System Call mount* per aggiungere sotto alberi virtuali al *File System* grazie al sistema di nomenclatura. Infatti, il comando *UMView mount* ha la stessa semantica della *mount* standard. Quando un *File System* F viene montato su una directory D, la directory radice di F diventa D e il contenuto di F sovrascrive il contenuto di D. Se D non è vuota, allora D e tutti i suoi sotto alberi del *File System* diventano inaccessibili, nascosti da F. L'operazione *UMView mount* può essere vista come un operatore overlay: F è sovraimposto a D. *UMView* applica ed estende questa idea: la *System Call* virtuale "*mount*" è utilizzata da *UMView* per definire una nuova vista per i processi: quando un modulo dentro *UMView* gestisce una chiamata di *mount*, crea un'overlay. Genericamente la scelta del modulo dipende dal parametro filesystemtype. *UMView mount* ha due differenze fondamentali rispetto al *mount* del kernel:

1. gli effetti della *System Call mount* in *UMView* sono limitati ad un parziale SCVM che è eseguito dalla chiamata. Dopo che *UMView* ha eseguito un'operazione di *mount* sulla directory obiettivo /mnt, il co-

mando *ls /mount*, se eseguito da una shell all'esterno della macchina virtuale, continuerà a mostrare il precedente contenuto di /mnt;

2. La chiamata standard del kernel di "*mount*" ha delle restrizioni d'uso: l'utilizzo è limitato all'amministratore del sistema, mentre la *mount* in *UMView* può essere invocata da qualunque processo.

UMView estende l'idea della *System Call* "*mount*" come segue:

- Ogni *mount* può ridefinire completamente la visione del processo del *File System*. Questo include le ridefinizioni del *mountpoint* o dell'hiding di un'immagine montata da una *mount* precedente. Ovviamente una *mount* è legale solo quando non ci sono altre *mount* che dipendono da da qualche *pathname* fornito da essa.

- La *mount* di *UMView* permette di montare file, mentre le specifiche di POSIX limitano l'uso ad una directory come *mountpoint*.

- *UMView* permette istanze ricorsive. Dal punto di vista di un utente di *UMView*, quando una macchina virtuale di *UMView* è eseguite dentro un'altra macchina virtuale di *UMView*, la corrente vista è condivisa da entrambe.

L'implementazione di *UMView* è in grado di gestire efficientemente le *mount* annidate come è in grado di gestire le esecuzioni annidate. In ultimo, *UMView* non crea un altro processo per gestire *mount* multiple o chiamate ricorsive. Usando purelibc, *UMView* cattura le *System Call* per eseguire con l'efficienza di una chiamata a funzione.

Sfortunatamente, il *File System* non ha una nomenclatura per tutto; infatti, il supporto di rete ha uno spazio dei nomi separato. Le interfacce, gli stack, le famiglie di protocollo non hanno delle entità del *File System* per

dar loro un nome. L'intero supporto di rete è stato disegnato come globale e condiviso da tutti processi. Senza uno spazio dei nomi gestibile l'API fornita dalle librerie non da l'astrazione necessaria alla virtualizzazione delle entità di rete (stack, interfacce, etc..) o l'accesso a diversi stack nello stesso momento.Per evitare tale problema *UMView* supporta l'estensione *msockets* della *Berkeley sockets interface*, attraverso la quale è possibile mappare diversi stack sul *File System*.

5.1.2 Algoritmo di ingabbiamento di *UMView*

UMView per prendere il controllo di un processo effettua il seguente schema:

1. Effettua una fork ed prepara l'ambiente per il figlio
2. Traccia il figlio appena creato
3. Quando il figlio invoca una *System Call*, il nucleo cercherà nelle *System Call* dei moduli registrati se questa è da gestire o meno
4. Se questa è da gestire allora sarà gestita dal modulo apposito altrimenti sarà ridirezionata al kernel

Questo avviene sia per i programmi lanciati direttamente dalla shell lanciata dal nucleo che per i programmi lanciati esternamente e successivamente ingabbiati nella macchina virtuale.

5.1.3 Tavola Hash di *UMView* e problema delle *System Call* mancanti

La struttura cardine per gestire le *System Call* e ridirezionarle verso i moduli è una *hash table* globale,che conserva tutte le strutture dati di tutti

i servizi forniti dai moduli (e dai loro sotto-moduli) e che permette un approccio veloce e scalabile per inoltrare tutte le *System Call* chiamate dal sistema al modulo giusto. Molti tipi di oggetti possono essere conservati nella *hash table*:

- moduli
- pathname
- addess families
- *System Call*
- ecc..

Ogni oggetto ha la sua propria *sum hash* che è un intero (long), la chiave *hash* è il modulo del numero degli elementi della *hash table*. Ogni oggetto è conservato nella *hash* table in una lista di corrispondente alla sua chiave hash. Segue la struttura dati associata ad ogni oggetto:

Listing 5.1: codice di ht_elem

```
struct ht_elem {
void *obj;
char *mtabline;
struct timestamp tst;
unsigned char type;
unsigned char trailingnumbers;
unsigned char invalid;
struct service *service;
struct ht_elem *service_hte;
void *private_data;
int objlen;
long hashsum;
int count;
confirmfun_t confirmfun;
struct ht_elem *prev,*next,**pprevhash,*nexthash;
};
```

- obj è l'oggetto (la cui lenght è contenuta in object lengt;
- type è la tag che si riferisce al tipo dell'oggetto;
- hashsum è la sum hash, permette una veloce selezione all'interno della lista di collisioni, se la somma *hash* non coincide, l'oggetto non è quello voluto;
- tst è il *timestamp*;
- *service* e *service hte* sono link veloci al servizio che possiede questo elemento;

- *private data* è un dato opaco dove il modulo può conservare le sue informazioni riguardo questo oggetto;
- *count* è il numero di istanze correntemente usate per la pulizia della memoria;
- *confirmfun* è funzione di conferma per gestire le eccezioni;
- *mtabline* è la tab line della *mount* (quella mostrata da umproc in /proc/*mount*s)
- *prev*, *next*, *pprevhash*, *nexthas*, collegamenti per la lista delle collisioni e per lo scan lineare di tutti gli elementi dello stesso tipo.

Il tipo di oggetto è utilizzato nella computazione somma e nella chiave hash, per cui gli oggetti che hanno lo stesso valore ma diverso tipologia sono conservati indipendentemente. Quando un processo richiede una *System Call*, *UMView*, cerca nell'*hash table* quale è l'oggetto responsabile della sua gestione. L'elemento dell'*hash table* (spesso riferito nel codice come *hte*) è utilizzato da tutta la macchina virtuale e dai moduli come chiave per trovare la virtualizzazione adatta.

5.2 Moduli

UMView e il progetto *ViewOS* si incentrano sull'espandibilità del progetto attraverso dei moduli. Questi sono delle piccole librerie condivise compilate separatamente dal codice e sono da integrare nella macchina virtuale una volta che questa è stata eseguita tramite comando esterno. Questi moduli aggiungono potenzialità alla macchina virtuale che può essere facilmente espansa a seconda delle necessità riscontrate.

5.2.1 UmNet

Umnet è il modulo di *ViewOS* per il multi-networking. Infatti supporta l'estensione multi stack della *Berkeley Socket API* chiamata *msockets*. Tramite questo modulo è possibile intercettare le chiamate di rete e renderle disponibili alla macchina virtuale. È ancora migliore il fatto che la modularità di *UMView* permetta ai programmatori di programmare degli altri sotto moduli che possono essere successivamente montati nella macchina virtuale.

Funzionamento di Umnet

Utilizzando il servizio di sottoscrizione degli eventi e quello di notifica fornito da *UMView*, *umnet* contiene un vettore di puntatori (uno per ogni processo). Ogni elemento di questo vettore (defnet), si riferisce ad una struttura *struct umnetdefault*, condivisa tra tutti i processi che hanno la stessa configurazione di generica. Infatti una struttura di tipo *struct umnetdefault* ha un contatore e un vettore di riferimenti ad una struttura *struct umnet*, una per ogni famiglia. Il contatore è impostato a zero per il primo processo. L'informazione sugli stack generici è ereditata durante le fork/clonazioni, il contatore incrementa costantemente e decresce quando un processo termina o cambia il la sua configurazione generica degli stack. Quando un contatore di processi diventa negativo la struttura *umnetdefault* viene rilasciata. La funzione di controllo trova il punto di *mount* per le m*socket* e per il file dell'operazione sul punto di *mount*. Se il codice utente chiama le *socket*, la funzione di controllo usa il tag *CHECKSOCKET* e attraverso *defnet* è possibile trovare lo stack generico per la famiglia di richieste. La funzione di controllo riconosce anche tutti i tipi di *File System* con il prefisso "*umnet*" dati alla *System Call* "*mount*".

I sotto moduli attualmente implementati e che sono stato utilizzato in questa tesi sono :

- umnetnull
- androidumnetnull

Sono i sotto moduli che sono statoi utilizzati per testare il comportamento della macchina virtuale su *Android*. Questo sotto modulo manda in errore *EAFNOSUPPORT* tutte le richieste di accesso alla rete che gli vengono fornite.

Listing 5.2: codice di umnetnull

```
int umnetnull_msocket (...){
errno = EAFNOSUPPORT ;
return -1;
}
int umnetnull_init (...) {
return 0;
}
int umnetnull_fini (...){
return 0;
}
struct umnet_operations umnet_ops ={
. msocket = umnetnull_msocket ,
. init = umnetnull_init ,
. fini = umnetnull_fini ,
};
```

La prima cosa che si nota di questo modulo è l'estrema semplicità. Questo infatti gestisce una struttura dati che al suo interno ha solo puntatori a

funzione e sembra emulare il comportamento di una classe nel senso informatico del termine. La struct infatti definisce come funzione di inizializzazione la funzione *umnetnull_init* e la funzione *umnetnull_fini* come distruttore. La funzione umnetnull_msocket invece viene impostata come principale funzione del modulo. La struttura umnet_operations è definita nell'header incluso nel modulo sopra citato *include/umnet.h*, tale struttura ha con sè i puntatori alle funzioni definite nel modulo *umnet* al fine di controllare e intercettare le operazioni di rete.

Nella normale esecuzione di *umnet*, tale modulo dovrebbe essere utilizzato attraverso la chiamata dei seguenti comandi che fungono anche da esempio di loro utilizzo in un ambiente linux:

1. Si avvia la macchina virtuale da shell

```
$ umview bash
umview init
```

2. si aggiunge il servizio per la gestione dei *socket*: umnet

```
$ um_add_service umnet
umnet init
```

   Il comando quindi andrà a cercare nel path DEFAULT MODULES PATH il file chiamato umnet.so per poi caricarlo con la Virtual System Call.

3. si monta ora il sotto modulo per intercettare correttamente le *System Call*

```
$ \textit{mount} -t umnetnull none /dev/net/null
```

   il comando quindi monterà il nostro modulo umnetnull sul path specificato (/dev/net/null).

4. il passo successivo sarà eseguire un comando col nostro modulo

```
$ mstack /dev/net/null ip addr
```

per far sì che le operazioni di rete effettuate dal programma virtualizzato vengano inoltrate attraverso lo stack virtuale nullo.

5.2.2 UmDev

È altresì presente un modulo che serve a gestire i descrittori dei device associati ai nostri driver user space. Il l'implementazione è praticamente identica a quella vista precedentemente mentre l'interfaccia di esecuzione invece non avendo bisogno di un'interfaccia di rete non ha bisogno del comando *mstac* essere provata.

Prova di utilizzo di umdevnull

Si inizia eseguendo la macchina virtuale

```
$ umview bash
umview init
```

Quindi dovremo comunicare alla macchina virtuale di caricare il modulo umdev

```
$ um_add_service umdev
umdev init
```

A questo punto dovremo montare il sottomodulo tramite il comando *mount*

```
$mount -t umdevnull none /dev/umnull
```

In questo modo faremo fallire tutte le *System Call* che si riferiranno al *File System* e ai vari file

- open

- write
- ioctl
- lseek
- close

in quanto saranno gestite dal nostro modulo.

5.3 Limitazioni del progetto e problematiche di sviluppo

A questo punto è necessario analizzare le limitazioni che il *porting*, così fatto di questo progetto può avere e cosa invece è da cambiare soprattutto alla luce delle problematiche relative alla gestione di alcune differenze tecniche che attualmente *Android* ha rispetto ad un sistema *Linux*,.

5.3.1 Problematica riguardo la gestione dell'ambiente di esecuzione

Il primo grande problema è evidente nella natura stessa del progetto *UM-View*. Esso infatti è un programma fatto per essere eseguito da shell in un *File System* adatto a gestire programmi del genere. Il *File System* di *Android* invece non permette nemmeno attraverso la shell di *Android* Debug Brige di eseguire programmi che non siano nella directory *sh/bin* senza che sia necessario ogni volta riconfigurare la directory di ambiente. Per questo motivo ho aggiunto lo script di configurazione dell'ambiente tra gli strumenti utilizzati nella tesi. Questa sebbene sembri poco importante è una limitazione non leggera.

Esempio:

```
$ ./umview
CANNOT LINK EXECUTABLE:
could not load library "libum\textit{core}.so" needed by "./umview";
caused by library "libum\textit{core}.so" not found
```

Questo significa che per accedere ai comandi della macchina virtuale è sempre necessario impostare l'LD_LIBRARY_PATH che viene resettata ad ogni esecuzione della shell di ADB. Questo ne rende difficoltosa l'esecuzione e la gestione anche per programmi esterni se questi hanno necessità di accedere ai comandi della macchina virtuale.

5.3.2 Problematica riguardo la possibilità di controllare processi non eseguiti dalla shell principale di *UMView*

Questa è stata l'attività principale della tesi cioè cercare di superare il problema che Davide Berardi aveva trovato nello sviluppo della sua tesi: controllare all'interno di *UMView* programmi lanciati da interfaccia grafica;

Per risolvere questo problema ho utilizzato il comando um_attach presente nelle utility di *UMView*.

um_attach UmView presenta al suo interno un comando che le permette di agganciarsi a programmi che non vengono eseguiti direttamente dalla shell, comando che andremo a trattare è per l'appunto

```
$um_attach process_pid
```

tale comando consente alla macchina virtuale di ingabbiare programmi lanciati esternamente ad essa, tale programma infatti avvia forzatamente

l'algoritmo con il quale *UMView* effettua la ptrace sui processi per controllarli. Questo comando è stato assolutamente vitale per lo sviluppo di questa tesi in quanto consente di lanciare un'applicazione dall'interfaccia grafica di *Android* e poterla agganciare dalla macchina virtuale senza essere costretti a lanciare il programma direttamente dalla shell controllata da *UMView*.

5.3.3 Mancata gestione di alcune System Call

Nei capitoli precedenti ho descritto più volte come è nessaria la gestione delle *System Call* e in particolare della *System Call mount* all'interno della macchina virtuale. Soprattutto la *System Call mount* è vitale per poter caricare di volta in volta i sottomoduli per gestire le altre *System Call*. Il problema nel quale mi sono imbattuto è stato appunto la mancata gestione della *System Call mount*, questa infatti è del tutto ignorata:

```
$mount -t umdevnull none /dev/umnull
mount: No such file or directory
```

Come si vede il risultato ottenuto su adb shell è ben diverso da quello che ci si aspetta invece su un sistema operativo *Linux*,.

Questo perchè?

Come scritto da Davide Berardi nella sua tesi, il *porting*, da lui programmato per *UMView* presenta dei problemi riguardo la cattura di alcune *System Call*. Ciò è dovuto a come *UMView* gestisce le chiavi delle tabelle di riferimento delle *System Call*, cioè legandole il calcolo della loro *hash* sum al loro numero della chiamata di *System Call* come identificativo per esse. Le cause per cui questo calcolo non avviene come dovrebbe possono essere:

- non avere una libreria allineata

- una libreria potrebbe non essere raggiungibile lato utente con i giusti identificativi delle *System Call*
- il fatto che Bionic comunque differisce dalla standard libc

UMView usa quindi un dizionario le cui chiavi vengono calcolate a partire proprio da questi valori e quindi risulta che se per qualche ragione, questo valore viene intercettato male nel programma, la chiamata alla syscall non è riconosciuta e quindi reinoltrata al kernel.

Tale problematica è avvenuta soprattutto per quanto riguarda le *System Call* relative ad umnet. È stato visto che la maggior parte di esse non tiene traccia delle invocazioni delle *System Call* da parte dei programmi che fanno accesso alla rete: le chiamate Write e read per esempio passano del tutto inosservate.

Superamento dell'ostacolo dato dalla mancata intercettazione della syscall *mount*

Nel cercare di effettuare i test per capire se i processi venivano veramente tracciati anche per le chiamate di rete, sono stato costretto ad utilizzare *umnet* e nello specifico cercare di utilizzare il suo sottomodulo *umnetnull*. L'unico metodo per aggirare il problema che ho trovato per superare tale problematica è stata una modifica del codice sorgente di *umnet* e più imparticolare della funzione di init:

```
   static void
__attribute__ ((constructor))
init ( void )
{
printk ( KERN_NOTICE "umnet init\n" ) ;
```

```
s.name = "umnet" ;
s.description = "virtual (multi-stack) networking" ;
s.destructor = umnet_destructor ;
s.ioctlparms = umnet_ioctlparms ;
s.syscall = ( sysfun * ) calloc ( scmap_scmapsize , sizeof(sysfun) ) ;
s.socket = ( sysfun * ) calloc ( scmap_sockmapsize , sizeof(sysfun) ) ;
s.virsc = ( sysfun * ) calloc ( scmap_virscmapsize , sizeof(sysfun) ) ;
s.ctl = umnet_ctl ;
MCH_ZERO(&(s.ctlhs)) ;
MCH_SET(MC_PROC, &(s.ctlhs)) ;
SERVICESYSCALL(s, \textit{mount}, umnet_\textit{mount}) ;
SERVICESYSCALL(s, u\textit{mount}2, umnet_u\textit{mount}2) ;
SERVICEVIRSYSCALL(s, msocket, umnetnull_msocketA)
SERVICESYSCALL(s, lstat64, umnet_lstat64) ;
SERVICESYSCALL(s, fcntl, umnet_fcntl64) ;
SERVICESYSCALL(s, access, umnet_access) ;
SERVICESYSCALL(s, chmod, umnet_chmod) ;
SERVICESYSCALL(s, lchown, umnet_lchown) ;
SERVICESYSCALL(s, ioctl, umnet_ioctl) ;

s.event_subscribe = umnet_event_subscribe ;

}
```

aggiungendo

```
SERVICEVIRSYSCALL(s, msocket, umnetnull_msocketA);
```

e la relativa funzione:

```
umnetnull_msocketA ( int domain , int type , int protocol ,
    struct\textit{ umnet }*nethandle )
{
printk ( "recived msocket on umnetandroid null net, failing\n" ) ;
errno = EAFNOSUPPORT ;
return - 1 ;
}
```

Dopo alcuni esperimenti, notavo che nulla cambiava nell'esecuzione dei programmi quindi ho cercato di far fallire la chiamata di inserendo un valore di ritorno volutamente fallace per cercare di comprendere se la connessione delle varie applicazioni usate per i test fallisse o no.

```
static int checksocket
  ( int type , void *arg , int arglen , struct ht_elem *ht )
```

Il risultato è stato che dopo il ritorno del *socket* al valore -1, veniva chiamata la *System Call* virtuale introdotta.

```
    printk ( "umnet.c = checksocket\n" ) ;
    //errore introdotto per testing
    return - 1 ;
```

Questo è successo perchè *umnet* ha richiamato mstack per gestire la chiamata di rete del processo e quindi questo l'ha fatta cadere; cosa che altrimenti non avveniva in quanto non è stato caricato il modulo tramite l'operazione di *mount* che dovrebbe impostare correttamente il modulo all'interno della macchina virtuale quindi l'operazione è risultata comunque forzata.

5.4 Riorganizzazione del programma e sviluppi futuri

Al fine di avere uno sviluppo futuro del programma purtroppo si deve scegliere di non seguire le linee guida originali, facendo in modo che *UMView* lasci la sua natura di programma per shell di *Linux*, e quindi segua le linee guida tracciate dall'*NDK* facendo in modo di integrarsi col codice *Java*. Questo vorrebbe dire

1. Cambiare la struttura del programma passando dall'essere un programma modulare ad diventare un programma statico in cui i moduli di umview base sarebbero preconfigurati al momento della compilazione;

2. Cambiare la sua struttura di eseguibile con molti eseguibili esterni per essere invece compilato come una grande libreria dinamica ed essere quindi integrato in un ambiente java interattivo che possa accedere alle sue funzionalità interne direttamente dal codice java;

3. Vedere quali funzionalità ora scritte in C potrebbero essere tradotte in codice *Java* senza perderne la potenzialità, così da evitare problemi futuri di compatibilità con le prossime versioni di *Android*;

Queste sono le modalità con le quali è possibile continuare il *porting*, del progetto per renderlo efficente ed efficace nel futuro oltre che utilizzabile attraverso l'interfaccia grafica e senza la necessità di avere a disposizione una *shell* come quella fornita da *Android Debug Bridge*.

Avendo tutto preconfigurato e precaricato al momento della compilazione sarebbe anche possibile saltare eventuali operazioni di *mount* e di configurazione a runtime dei moduli che potrebbero non andare a buon fine oltre al fatto che comunque renderebbero il programma più facilmente accedibile da

programmi esterni che non avrebbero bisogno di dover indicare l'ambiente per il linking dinamico delle librerie.

5.4.1 Alternative alla gestione diretta della pthrace

Un'alternativa valida all'uso di Pthread potrebbe essere quella dell'utilizzo del *socket* di zygote presente in */dev/socket/zygote* che potrebbe essere utilizzata (in quanto con permessi di tipo 666) dal programma al fine di gestire in modo più elegante e in modo granulare le chiamate effettuate da parte dei processi che si vuole ingabbiare. In questo modo probabilmente si potrebbe correggere agevolmente il problema della mancata gestione delle *System Call* invotate dai processi in quanto, essendo un'operazione di listening, il codice relativo non dovrebbe cambiare a seconda delle implementazioni di *UMView* rispetto alla Bionic.

Capitolo 6

Valutazione

Sono state effettuate delle prove al fine di valutare sperimentalmente che lo stato attuale del porting permettesse un vero ingabbiamento delle applicazioni, anche quando queste non fossero state lanciate direttamente dalla *shell* legata a *UMView*. Per fare queste prove sono stati utilizzati:

- Google Nexus 4
  - *Sistema operativo*: Android 4.4.2 (*API 19*)
  - *Runtime*: *Dalvik Virtual Machine* e *ART*;
- Android AVD
  - *Sistema Operativo* : Android 4.4.2 (*API 19*)
  - *Runtime*: *Dalvik Virtual Machine*;

Tramite *Android Debug Bridge* è stato possibile trasferire ed eseguire i file sviluppati su emulatore e su device fisico, mentre lo sviluppo dei test è stato effettuato tramite Eclipse *ADT*.

6.1 Criteri

I test sono stati svolti per dimostrare se era possibile o meno controllare i programmi non lanciati dalla *shell* di *UMView* ma dall'interfaccia grafica di Android ed in caso studiarli. Per questo motivo i criteri per i quali si può valutare se lo stato attuale del programma è in grado o meno di controllare processi esterni non possono che essere i seguenti:

1. **Controllare applicazioni a prescindere dal linguaggio in cui queste siano state sviluppati**: L'ingabbiamento deve funzionare a prescindere dal linguaggio stesso di sviluppo dell'applicazione, soprattutto deve funzionare con le applicazioni scritte in Java.

2. **Modifica del comportamento dei processi controllati**: Una volta che il processo è stato controllato, è importantissimo analizzare se è possibile o meno intercettare le sue chiamate alle *syscall* ed eventualmente modificarne l'esito.

3. **Controllare se la modifica del comportamento dei processi controllati cambia seconda del linguaggio sorgente**: È importante che a prescindere del linguaggio e del metodo di compilazione dei programmi, l'ingabbiamento da parte di *UMView* non cambi e che quindi si possa pensare che *UMView* non risenta di queste differenze.

4. **Cambio di comportamento a seconda della Runtime**: Il criterio più importante in assoluto è stato quello di verificare che la nuova runtime di Android *ART* non avesse cambiato il comportamento di *UMView* ma soprattutto è necessario considerare il fatto che anche l'ingabbiamento potesse essere modificato o meno dalla nuova runtime.

6.2 Come si è scelto di verificare i criteri indicati

Sono stati sviluppati una serie di mini programmi del tipo:

- Un programma scritto in *C* che esegue il comando "*vuname -a*[1]" una volta controllato, lanciato da *shell* diversa da quella controllata da *UMView*;
- Un programma scritto in JAVA che esegue il comando "*vuname -a*" una volta controllato, questo lanciato da interfaccia grafica;
- una coppia di programmi scritti in *C* per effettuare un sistema di client/server in TCP/IP;
- una coppia di applicazioni scritti in *Java* con interfaccia grafica per effettuare un sistema *client/server* TCP/IP.

6.2.1 Programmi che eseguono il comando "*vuname -a*"

Questi due programmi in *Java* ed in *C* eseguono il comando "*vuname -a*", questo è il modo più semplice e veloce per sapere se un programma è stato controllato o meno dalla macchina virtuale. Questo metodo prescinde dai livelli di astrazione superiore che possono essere quelli relativi all'utilizzo di moduli e servizi di *UMView* per l'intercettazione delle *syscall*. Con questi due programmi si può verificare facilmente se:

1. È possibile controllare programmi lanciati da interfaccia grafica o da *shell* esterne ;

[1] descritto nel capitolo riguardo ViewOS

2. È possibile controllare programmi a prescindere del linguaggio usato per il loro sviluppo;

3. È possibile controllare programmi a prescindere dalla Runtime di esecuzione e se il controllo cambia comportamento;

Purtroppo questo tipo di test non consente di controllare se il comportamento dei programmi controllati cambia in quanto non vengono richieste particolari *syscall* che possano essere tracciate.

6.2.2 Programmi Server e Client scritti in *C*

Questa coppia di programmi effettuano una semplice trasmissione di dati tramite socket tcp/ip tra un programma server ed un programma client. Tramite questo tipo di test molto semplice e veloce è facile per verificare alcuni criteri che ci siamo dati:

1. È possibile verificare che siano controllati programmi non lanciati dalla *shell* di *UMView*;

2. È possibile verificare che sia possibile intercettare e cambiare il funzionamento delle *syscall* invocate dal programma;

Essendo comunque due programmi scritti in *C* ed eseguiti da Shell non è possibile verificare sono tramite questo test che *UMView* supporti parimenti *Java* e *C* per quanto riguarda le chiamate di rete. Non è possibile tramite solo questo test neanche sapere se *UMView* possa o meno intercettare le *syscall* invocate da un programma *Java*.

6.2.3 Programmi Server e Client scritti in *Java*

Questa coppia di programmi non sono altro che due semplici applicazioni con interfacce grafiche che effettuano una connessione tra server e client del tipo *TCP/IP*. Tramite questo test è possibile verificare i criteri:

1. È possibile verificare se *UMView* riesca a controllare le applicazioni lanciate da interfaccia grafica;

2. È possibile verificare se *UMView* riesca a intercettare le *syscall* invocate da applicazioni *Java*;

6.3 Commenti

Sebbene presi singolarmente questi test non bastano a dare una valutazione oggettiva, insieme sono più che soddisfacenti per dare una stima secondo i criteri scelti e a comprendere quanto questi siano stati raggiunti o meno. Infatti nella loro totalità questi test toccano tutti i criteri necessari per comprendere se *UMView* riesce o meno a controllare i processi e ad alterare i risultati delle *syscall* invocate. Nella fattispecie essendo tra gli obiettivi ultimi del progetto quello di poter controllare e reindirizzare le chiamate di rete effettuate da un programma di *Voip* i test riguardo il sistema client-server in *Java* sono i più importanti.

6.4 Esempi e Risultati

È ora necessario a questo punto descrivere nel dettaglio i test effettuati ed esporne gli esiti al fine di poter dare una valutazione del progetto.

6.4.1 Esecuzione del comando *vuname -a*

Ora passiamo a descrivere i codici ed i risultati dei due programmi sviluppati per eseguire il comando "*vuname -a*"

Il programma scritto in *C* per eseguire il comando "vuname -a"

Questo è il primo dei due programmi che effettuano la chiamata del comando "*vuname -a*" e ne spiego qui di seguito il codice :

```
static void catch_function(int signo) {
printf("ricevuto segnale = %d\n\n", signo);
fflush(stdout);
}

static void test_function() {


if (execl("vuname", "vuname", "-a", (char*)0) < 0)
perror("execl:");
}

int main() {

if (signal(SIGINT, catch_function) == SIG_ERR) {
perror("An error occurred
while setting a signal handler.\n");
return EXIT_FAILURE;
    }
```

```
pause();
test_function();


  }
```

Come si nota facilmente questo l'algoritmo di questo programma è il seguente:

1. Aspetta un segnale e si mette in pausa fin quando non l'ha ricevuto:

Listing 6.1: gestione dei segnali

```
if (signal(SIGINT, catch_function) == SIG_ERR)
{
perror("An error occurred
while setting a signal handler.\n");
return EXIT_FAILURE;
}
```

Questa linea di codice permette di nominare una funzione quale gestore dei segnali. Questo serve per dare il tempo all'utente di dare il comando *um_attach* alla macchina virtuale per controllare il processo.

2. Il codice qui di seguito invece esegue il comando

Listing 6.2: esecuzione del comando vuname

```
if (execl("vuname", "vuname", "-a", (char*)0) < 0)
                perror("execl:");
```

Il risultato cambia a seconda che la variabile di ambiente $LD_LIBRARY_PATH sia stata impostata o meno. Quindi il risultato può essere questo:

```
CANNOT LINK EXECUTABLE:
could not load library "libumlib.so" needed by "vuname";
caused by library "libumlib.so" not found
```

Oppure questo:

```
Linux      GNU/Linux/View-OS
```

E quindi riscontrare il fatto che il processo è sotto il controllo di *UMView*.

Il programma scritto in *Java* per eseguire *vuname -a*

In *Java* invece ho scritto una piccola applicazione che ha legato l'esecuzione del metodo *ExecuteCommand* alla pressione di un bottone su interfaccia grafica.

```
private String ExecuteCommand(String command) {
 String[] envp = { "LD_LIBRARY_PATH=/data/local/bin:$LD_LIBRARY_PATH" };
 StringBuffer output = new StringBuffer();

 Process p;
 try {
   p = Runtime.getRuntime().exec(command, envp);
```

```
    p.waitFor();
    BufferedReader reader = new BufferedReader(new InputStreamReader(
    p.getInputStream()));
    String line = "";
    while ((line = reader.readLine()) != null) {
output.append(line + "\n");
}
      } catch (Exception e) {
    e.printStackTrace();
    }
    return output.toString();
    }
```

Listing 6.3: stringa dell'ambiente

```
String[] envp =
{ "LD_LIBRARY_PATH=/data/local/bin:$LD_LIBRARY_PATH" };
```

Serve al programma per impostare le variabili di ambiente per il processo che andrà ad eseguire il comando. Come detto in precedenza senza impostare la $LD_LIBRARY_PATH Android non è in grado di trovare le librerie nella cartella dove sono stati copiati i binari di *UMView*.

Listing 6.4: Esecuzione di vuname in Java

```
p = Runtime.getRuntime().exec(command, envp);
 p.waitFor();
```

Questo pezzo di codice *Java* serve per far eseguire il comando da un processo figlio e aspettare che esso abbia finito di eseguire una volta lanciato.

Listing 6.5: gestione dell'output in Java

```
BufferedReader reader =
  new BufferedReader(new InputStreamReader(
    p.getInputStream()));
String line = "";
while ((line = reader.readLine()) != null) {
output.append(line + "\n");
```

Questo pezzo di codice invece serve per permettere a *Java* di prendere l'output dell'esecuzione del comando e tradurlo in stringa da poter utilizzare all'interno per debug. Questo pezzo di codice è stato introdotto per avere il responso del test.

Listing 6.6: Override evento OnClick

```
@Override
public void onClick(View v) {
text.setText(
ExecuteCommand("/data/local/bin/vuname -a"));
}
```

Queste poche righe di codice sono quelle necessarie per legare l'esecuzione del metodo *ExecuteCommand* all'utilizzo del bottone dell'interfaccia grafica. Il resto dell codice serve invece impostare il risultato di *ExecuteCommand*, cioè la stringa di output, come testo interno di una casella di testo che è stata creata opportunamente per mostrare all'utente il risultato.

Il risultato di questo test è positivo, dimostrando che *UMView* riesce a prendere il controllo del processo.

Conclusioni e commenti sui test dell'esecuzione di *vuname -a*

Il risultato di questi test è positivo e dimostra che *UMView* su Android è in grado di tracciare programmi lanciati da interfaccia grafica contrariamente a quanto affermato nella tesi di Davide Berardi.

Lo stesso test è stato effettuato sia utilizzando la runtime *Dalvik Virtual Machine* che *ART* dando i medesimi risultati senza in alcun modo modificarne l'esito.

6.4.2 Test a riguardo Server/Client TCP scritti in *C*

Questo è un semplice test di comunicazione tra server e client sviluppato:

Il Server TCP/IP scritto in C

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>
#include <signal.h>

#define SERVERPORT 3002
#define SERVERADDRESS "127.0.0.1"
#define MAXBUF 32

static void catch_function(int signo) {
```

```
printf("ricevuto segnale = %d\n\n", signo);
fflush(stdout);
}

int main(int argc, char * argv[])
{
  if (signal(SIGINT, catch_function) == SIG_ERR) {
    perror("An error occurred
  while setting a signal handler.\n");
  return EXIT_FAILURE;
  }

  pause();

  int server_socket, connect_socket, retcode, nw;
  socklen_t client_addr_len;
  struct sockaddr_in server_addr, client_addr;
  char buffer[MAXBUF];
  char * client_address; // lo useremo come indirizzo del client in formato string

  // apertura socket del server
  if (((server_socket = socket(AF_INET,SOCK_STREAM,0))) == -1)
  {
  perror("Error in server socket()");
  exit(-1);
  }
```

```
// preparazione dell'indirizzo locale del server
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(SERVERPORT);
server_addr.sin_addr.s_addr = inet_addr(SERVERADDRESS);

// bind del socket
if ((retcode = bind (server_socket,
  (struct sockaddr*)  &server_addr,
  sizeof(server_addr))) == -1)
{
perror("Error in server socket bind()");
exit(-1);
}

// listen del socket
if ((retcode = listen(server_socket, 1)) == -1)
{
perror("Error in server socket listen()");
exit(-1);
}

printf("Server ready (CTRL-C quits)\n");
client_addr_len = sizeof(client_addr);

// cicla in attesa di connessione
while (1)
{
```

```
    // accetta le connessioni
    if ((connect_socket = accept(server_socket,
        (struct sockaddr *)&client_addr,
        &client_addr_len)) == -1)
    {
    perror("Error in accept()");
    close(server_socket);
    exit(-1);
    }

    // memorizzo l'indirizzo del client in formato stringa
    client_address = inet_ntoa(client_addr.sin_addr);

    printf("\nClient @ %s connects on socket %d\n",
    client_address,connect_socket);

    // riceve i messaggi e li stampa in output
    while ((retcode = read(connect_socket, buffer, MAXBUF-1)) > 0)
    {
    buffer[retcode] = '\0'; // buffer dev'essere una stringa
    printf(">> %s: %s\n", client_address, buffer);
    }

    // chiudo la socket
    close(connect_socket);
    }
  }
```

Come è facile vedere questo è un server il cui algoritmo è:

1. Attende che gli venga mandato un segnale (per dare eventualmente ad *UMView* la possibilità di controllarlo prima che lui esegua le *syscall* di rete;

Listing 6.7: gestore dei segnali

```
if (signal(SIGINT, catch_function) == SIG_ERR) {
        perror("An error occurred
        while setting a signal handler.\n");
        return EXIT_FAILURE;
    }
```

2. Attende che un client si connetta;
3. Accetta il pacchetto inviatogli dal client;
4. Stampa il pacchetto inviatogli dal client;
5. Attende una nuova connessione;

Il Client

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
```

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

#define SERVERPORT 3002
#define SERVERADDRESS "127.0.0.1"

static void catch_function(int signo) {
printf("ricevuto segnale = %d\n\n", signo);
fflush(stdout);
}

int main(int argc, char *argv[]) {
  if (signal(SIGINT, catch_function) == SIG_ERR) {
  perror("An error occurred
  while setting a signal handler.\n");
  return EXIT_FAILURE;
  }

  pause();

  int client_socket;
  struct sockaddr_in server_addr;
  char *msg;
  size_t msglen;
  int retcode;

  // verifico la correttezza dei parametri
```

```
if (argc != 2) {
printf("Usage: %s \"message to send\"\n", argv[0]);
exit(-1);
}

// preparo la variabile
//msg in base alla dimensione del messaggio inserito dall'utente
msg = malloc(strlen(argv[1]) * sizeof(char));

// apertura socket del client
if ((client_socket = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
perror("Error in socket()");
exit(-1);
}

// preparazione dell'indirizzo del server con cui connettere il socket
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(SERVERPORT);
server_addr.sin_addr.s_addr = inet_addr(SERVERADDRESS);

// connessione del socket al server
if ((connect(client_socket, (struct sockaddr *) &server_addr,
sizeof(server_addr))) == -1) {
perror("Error in connect()");
exit(-1);
}
```

```
  printf("Client connected to %s\n", SERVERADDRESS);

  // invio messaggio al server
  strcpy(msg, argv[1]);
  msglen = strlen(msg) + 1;
  retcode = write(client_socket, msg, msglen);

  printf("Sent %d (%u request) bytes on socket %d\n", retcode,
    (unsigned) msglen, client_socket);
}
```

Come è facile vedere questo è un client il cui algoritmo è:

1. Attende che gli venga mandato un segnale (per dare eventualmente ad *UMView* la possibilità di controllarlo prima che lui esegua le *syscall* di rete;

Listing 6.8: gestore dei segnali

```
if (signal(SIGINT, catch_function) == SIG_ERR) {
  perror("An error
  occurred while setting a signal handler.\n");
  return EXIT_FAILURE;
  }
```

2. Si connette al server;
3. Invia la stringa passatagli come parametro;
4. Stampa alcuni dati riguardo la connessione;

Il risultato del test sulla coppia server/client scritta in C

Per intercettare le chiamate di rete è stato necessario aggiungere il servizio *umnet* alla macchina virtuale, con le modalità descritte nei capitoli precedenti quindi senza poter caricare normalmente il modulo *umnetnull* e quindi ho dovuto utilizzare il metodo descritto nel capitolo sulla Progettazione. È stato riscontrato che *UMView* non intercetta tutte le *syscall* di rete, la maggior parte vengono ignorate. Infatti se *UMView* inizia a controllare il server una volta che questo ha lanciato la *bind*, nessuna altra syscall viene intercettata da parte di quest'ultimo e il comportamento del programma risulta invariato. L'esito risulta positivo invece sia nel server che nel client se *UMView* ingabbia il processo prima che questi possa fare alcun tipo di operazione di rete, in quanto la connessione di rete viene intercettata da *UMView* e quindi bloccata dimostrando che il processo era stato controllato e che la *syscal* è stata correttamente intercettata.

6.4.3 La coppia di programmi server/client TCP/IP scritta in JAVA

Al fine di fare questa prova ho creato due semplici classi *Java* che funzionassero in modo asincrono così da permettere ad Android di creare un *thread* che eseguisse il server o il client di turno.

La classe JavaServer

```
import java.lang.*;
import java.io.*;
import java.net.*;

import android.os.AsyncTask;

public class JavaServer extends AsyncTask<String, Void, String> {
  String data = "stringa di testo";

  protected String Run() {
  String result = "";
  try {
    ServerSocket srvr = new ServerSocket(3500);
    Socket skt = srvr.accept();
    result = "Server has connected!\n";
    PrintWriter out = new PrintWriter(skt.getOutputStream(), true);
    result += "Sending string: '" + data + "'\n";
    out.write(data);
    out.close();
```

```
    skt.close();
    srvr.close();
  }
  catch (Exception e) {
      result += "Problema " + e.getMessage();
      }
    return result;
}

@Override
protected String doInBackground(String... params) {
  return Run();
  }
}
```

Il metodo *Run* della classe JavaServer segue il seguente algoritmo:

1. Avvia il server;
2. Attende la connessione del client;
3. Invia al client la stringa “stringa di testo”;

Listing 6.9: Override dell'evento OnClick

```
@Override
public void onClick(View v) {
  Test js = new Test();
  AsyncTask<String, Void, String> k = js.execute();
  String result = "";
  try {
        result = k.get();

    } catch (InterruptedException e) {
        result = e.getLocalizedMessage();
    } catch (ExecutionException e) {
        result = e.getLocalizedMessage();
        }
  text.setText(result);
        };
});
```

Questo codice è necessario per legare l'utilizzo della classe *JavaServer* all'utilizzo di un bottone in interfaccia grafica.

Il metodo *doInBackground* è quello che viene automaticamente invocato dalla classe superiore *AsyncTask* ed è sovrascritto per eseguire il metodo *Run*.

La classe JavaClient

```
 package com.example.androidclienttest;


import java.io.*;
import java.net.*;
```

```
import android.os.AsyncTask;

public class JavaClient extends AsyncTask<String, Void, String> {

  protected String Run() {
  String result = "";
  try {
  Socket skt = new Socket("localhost", 3500);
  BufferedReader in = new BufferedReader
    (new InputStreamReader(skt.getInputStream()));
  result += "Received string: '";
  while (!in.ready()) {
  }
  result += in.readLine(); // Read one line and output
  result += "'\n";
  in.close();
  skt.close();
  } catch (Exception e) {
  result += e.getMessage();
  }
  return result;
  }


  @Override
  protected String doInBackground(String... params) {
```

```
  return Run();
  }

}
```

L'algoritmo del metodo *Run*

1. Si connette al server;
2. Riceve una stringa dal server;
3. ritorna la stringa ricevuta a chi l'ha invocato;

Il metodo *doInBackground* è quello che viene automaticamente invocato dalla classe superiore *AsyncTask* ed è sovrascritto per eseguire il metodo *Run*.

Listing 6.10: Override dell'evento OnClick

```
public void onClick(View v) {
  JavaClient jc = new JavaClient();
  AsyncTask<String, Void, String> k = jc.execute();
  String result = "";
  try {
        result = k.get();
        text.setText(result);
  } catch (InterruptedException e) {
        result = e.getLocalizedMessage();
  } catch (ExecutionException e) {
        result = e.getLocalizedMessage();
  }
```

Questa è la stringa necessaria per chiamare ed eseguire il metodo *Run* da java, come detto per quanto riguarda la classe *JavaServer* presentata prima è stato necessario utilizzare una classe propria di Android per gestire le operazioni asincrone.

Risultati e commenti del test server/client in Java

Per permettere ad un'applicazione *Java* eseguita su Android di usare i socket internet è stato necessario aggiungere nel *ApplicationManifest.xml* le seguenti linee che sbloccavano i permessi di rete:

Listing 6.11: modifica dei permessi nell'ApplicationManifest.xml

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

I risultati di questi test sono stati gli stessi che hanno riguardato i test della coppia server/client scritta in C:

1. Se la richiesta di rete viene effettuata quando il server è in attesa, allora non vi è alcun'altra *syscall* che viene tracciata;

2. Se invece il processo client o server vengono controllati da *UMView* prima che questi possano effettuare una richiesta di rete allora questa viene bloccata;

6.4.4 Commento ai risultati dei test

Tutti i test presentati e tutti i risultati degli stessi sono stati identici sia utilizzando la *Dalvik Virtual Machine* che *ART* dimostrando che il comportamento di *UMView* non cambia a seconda delle runtime.

A partire dai dati raccolti, dai criteri dati per la valutazione è possibile affermare che:

1. È possibile controllare programmi scritti sia in *Java* che in linguaggio nativo;

2. È possibile intercettare le *syscall* da esso invocate anche se in questo caso molte, per il problema descritto nella progettazione, vengono del tutto ignorate;

3. Il dirottamento delle *syscall* avviene in modo identico a prescindere dal linguaggio in cui l'applicazione è stata sviluppata;

4. *UMView* su Android non cambia il proprio comportamento in base a quale runtime è stata utilizzata;

Sebbene questi criteri siano stati soddisfatti quasi totalmente, il problema riguardo il mancato dirottamento delle *syscall* è tanto pesante da rendere vano il progetto se ciò non sarà risolto quanto prima.

Capitolo 7

Conclusioni

Sebbene questi criteri siano stati quasi tutti pienamente soddisfatti non è possibile pensare ad uno sviluppo successivo di UMView che non ne modifichi totalmente la struttura e la logica di base. Questo dimostra che purtroppo anche se è possibile trasportare linguaggi programmati per *Linux* su *Android*, questi hanno ancora molte problematiche riguardo l'utilizzo di alcune feature base della *libc* ce non sono state integrate in modo identico sulla Bionic. I problemi avuti ne sono una dimostrazione pratica e è stato dimostrato che l'utilizzo dell'NDK non è nel tutto consigliabile nel caso in cui si vogliano trasformare sistemi complessi a meno che non si utilizzino delle alternative atte ad aggirare i problemi tipici causati da queste differenze a basso livello. Per quanto riguarda invece l'idea di poter supportare i programmi VoIP tramite multistack delle connessioni TCP-IP, dato il recente annuncio di *Android* per quanto riguarda il futuro supporto del Multi-Networking, avrà interesse solo ed esclusivamente nell'ambito della personalizzazione del servizio offerto all'utente. Si spera altresì in un aumento di compatiblità di Bionic con la libc standard da parte degli sviluppatori di *Android* in modo da poter garantire maggiore libertà di sviluppo da parte dei programmatori e poter quindi ot-

tenere risultati migliori nel porting delle applicazioni in codice nativo C sui *device* portatili.

Bibliografia

[1] J. D. Dike. A user-mode port of the linux kernel. In Proc. of 2000 Linux Showcase and Conference, 2000.

[2] : J. D. Dike. User-mode linux. In Proc. of 2001 Ottawa Linux Symposium (OLS), Ottawa, 2001.

[3] Renzo Davoli, Square Website - http://virtualsquare.org/vm.html - 2004 *Relativi alla virtualizzazione parziale*

[4] sydbox : http://freecode.com/projects/sydbox

[5] PRoot : http://proot.me/

[6] Wikipedia - http://en.wikipedia.org/wiki/Ptrace

[7] Renzo Davoli, Michael Goldweber: Virtual SquareUsers,Programmers & Developers Guide

[8] Mahapatra, Lisa (November 11, 2013). "Android Vs. iOS: What's The Most Popular Mobile Operating System In Your Country?". Retrieved January 30, 2014.

[9] Elmer-DeWitt, Philip (January 10, 2014). Don't mistake Apple's market share for its installed base. CNN. Retrieved January 30, 2014.

[10] Yarow, Jay (March 28, 2014). This Chart Shows Google's Incredible Domination Of The World's Computing Platforms. Retrieved April 23, 2014.

[11] Samsung sells more smartphones than all major manufacturers combined in Q1. Retrieved May 12, 2014.

[12] Android's Google Play beats App Store with over 1 billion apps, now officially largest. Phonearena.com. Retrieved 2013-08-28.

[13] Android 4.4.2 KitKat running Kernel 3.10 on the Samsung Galaxy Ace Style. gsmarena.com. 2013-04-03. Retrieved 2014-04-11.

[14] Android 4.4.2 KitKat running Kernel 3.10 on the Exynos variant of the Samsung Galaxy S5 (SM-G900H). gsmkhmer.com. 2014-05-05. Retrieved 2014-05-05.

[15] Ryan Whitwam (November 25, 2013). HTC Posts Android 4.4 Kernel Source And Framework Files For One Google Play Edition, OTA Update Can't Be Far Off. androidpolice.com. Retrieved 2013-12-02.

[16] McPherson, Amanda (December 13, 2012). What a Year for Linux: Please Join us in Celebration. Linux Foundation. Retrieved April 16, 2014.

[17] Proschofsky, Andreas (July 10, 2011). Google: Android is the Linux desktop dream come true. derStandard.at. Retrieved March 14, 2013.

[18] Androidology -h Part 1 of 3 - Architecture Overview (Video). YouTube. September 6, 2008. Retrieved 2007-11-07. - https://www.youtube.com/watch?v=QBGfUs9mQYY

[19] Android Anatomy and Physiology Google I/O. 2008-05-28. Retrieved 2014-05-23. - http://androidteam.googlecode.com/files/Anatomy-Physiology-of-an-Android.pdf

[20] Android Kernel Features. - http://elinux.org/Android_Kernel_Features

[21] : Bornstein, Dan (2008-05-29). Presentation of Dalvik VM Internals (PDF). Google. p. 22. Retrieved 2010-08-16. [http://sites.google.com/site/io/dalvik-vm-internals/2008-05-29-Presentation-Of-Dalvik-VM-Internals.pdf?attredirects=0]

[22] Nexus One Is Running Android 2.2 Froyo. How Fast Is It Compared To 2.1? Oh, Only About 450% Faster. 2010-05-13. Retrieved 2010-05-21. [http://www.androidpolice.com/2010/05/11/exclusive-androidpolice-coms-nexus-one-is-running-android-2-2-froyo-how-fast-is-it-compared-to-2-1-oh-only-about-450-faster/]

[23] Rose, John (2008-05-31). with Android and Dalvik at Google I/O. Retrieved 2008-06-08. [http://blogs.sun.com/jrose/entry/with_android_and_dalvik_at]

[24] Google (2009-04-13). What is Android?. Retrieved 2009-04-19. [http://developer.android.com/guide/basics/what-is-android.html]

[25] Introducing ART. android.com. Retrieved 2013-11-02 [http://source.android.com/devices/tech/dalvik/art.html]

[26] Meet ART, Part 2: Benchmarks - Performance Won't Blow You Away Today, But It Will Get Better. androidpolice.com. 2013-11-12. Retrieved 2014-01-02. The numbers and the videos together paint a picture

of where ART stands today. It will definitely make a difference, but its current incarnation just hasn't matured enough to deliver significant gains [http://www.androidpolice.com/2013/11/12/meet-art-part-2-benchmarks-performance-wont-blow-away-today-will-get-better/]

[27] : Google introduces ART (Android Runtime) in KitKat. androidaio.com. Retrieved 2013-11-08. [http://androidaio.com/google-introduces-artandroid-runtime-in-kitkat/]

[28] : Google introduces ART (Android Runtime) in KitKat. androidaio.com. Retrieved 2013-11-08. [http://androidaio.com/google-introduces-artandroid-runtime-in-kitkat/]

[29] Renzo Davoli, Michael Goldweber: View-OS: Change your View on Virtualization.

[30] Federico Pareschi: Applying partial virtualization on ELF binaries through dynamic loaders

[31] Virtual Square: http://wiki.v2.cs.unibo.it

[32] View-OS source code: http://sourceforge.net/projects/view-os/ *Relativi all'architettura Android e alla programmazione in C su di essa*

[33] Giacomo Bergami: Pjproject su Android: uno scontro su pi livelli.

[34] Android NDK: http://developer.android.com/tools/sdk/ndk/index.html

[35] Android source code: https://github.com/android

[36] Linux syscall reference: http://syscalls.kernelgrok.com/

[37] Android Debug Bridge: http://developer.android.com/tools/help/adb.html

[38] Android initrc: https://android.googlesource.com/platform/system/core/+/master/rootdir/

Altri riferimenti

[39] Brian W. Kernighan, Dennis M. Ritchie: The C Programming Language.

[40] Andrew S. Tanenbaum: Structured Computer Organization.

[41] GNU 'make': http://www.gnu.org/software/make/manual/make.html

[42] ARM: http://www.arm.com

[43] GDB: http://www.sourceware.org/gdb

[44] Intel VT-x: http://ark.intel.com/it/Products/VirtualizationTechnology

[45] AMD AMD-V: http://www.amd.com/virtualization

[46] Qemu: http://wiki.qemu.org

[47] Davide Berardi : Porting della macchina virtuale UmView su sistema operativo Android ARM.

[48] Alessio Siravo: Esecuzione di applicazioni all'interno di una macchina virtuale su Android.

[49] Luca Verderame : Analisi di sicurezza del sistema operativo Android