Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

Porting della macchina virtuale UmView su sistema operativo Android ARM.

Tesi di Laurea in Architettura degli Elaboratori

Relatore:
Chiar.mo Prof.
Vittorio Ghini

Presentata da:
Davide Berardi

Sessione II
Anno Accademico 2012/2013

You know what the most complex piece of engineering known to man in the whole solar system is?
Guess what – it's not Linux, it's not Solaris, and it's not your car.
It's you.
And me.

...

I'm deadly serious: we humans have never been able to replicate something more complicated than what we ourselves are, yet natural selection did it without even thinking.

Linus Torvalds

Indice

Introduzione **5**

Paradigmi di virtualizzazione **6**
Virtualizzazione Totale . . . 7
Virtualizzazione legata all'esecuzione di processi . . . 9
Virtualizzazione Parziale . . . 11
Altre forme di virtualizzazione . . . 12
Paravirtualizzazione e hypervisors . . . 12

UmView **15**
All'interno di UmView . . . 15
ptrace . . . 15
UmNet . . . 16
Umnetnull . . . 16
La struttura umnet_operations . . . 18
Mstack . . . 19
Esempio: umnetnull . . . 19
Esempio: umdevnull . . . 20

Obiettivi e vantaggi del progetto **22**
Switch dell'interfacci dinamico . . . 23

Tools utilizzati al fine del porting **27**
Tools Compilativi . . . 27
ndk-build . . . 27
Tools Esecutivi . . . 29
AVD . . . 29
adb . . . 30
lo script setEnv.sh . . . 30
Tools relativi al Debug . . . 31

Porting **34**
Porting del core . . . 35
Modifiche alla bionic . . . 35
Modifiche al codice di UmView . . . 39
LibCap e LibMhash . . . 39
Il file defs_arm_android.h . . . 40
Il file defs.h . . . 41
Il problema delle system calls mancanti . . . 41
Porting delle utility (um_cmd) . . . 42
Porting di UmNet . . . 42
Il problema della DALVIK VM . . . 43

Sviluppi Futuri **45**

Conclusioni **47**

Introduzione

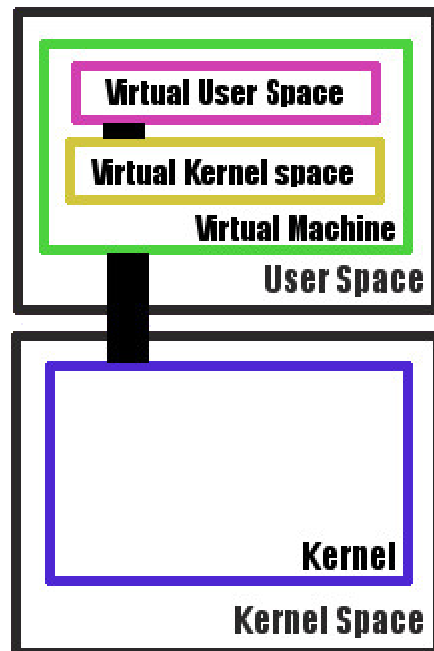
La libertà di azione sul proprio device personale è sempre stata un argomento delicato, sempre difesa da piccole cerchie di persone autodenominatesi *Hackers*.
Per questo motivo, soprattutto negli ultimi tempi, si è sviluppato sempre piú un discreto numero di espedienti per eludere le barriere piú o meno imposte dal device stesso.
Su questi presupposti, quindi, si basa questa tesi, presentando un metodo di *work-around* perlopiú mirato ad aggiungere alcune funzionalitá al sistema Android, sistema operativo sempre piú in espansione sul fronte dei device mobili.

A questo proposito verrà quindi introdotto, dopo una breve introduzione alle famiglie di macchine virtuali con le quali ci andremo a rapportare, il paradigma di virtualizzazione parziale e le sostanziali differenze con i classici paradigmi di virtualizzazione a cui siamo abituati, il progetto ViewOS, e nello specifico il programma UmView.

Inoltre verranno elencati alcuni dei problemi relativi alle, seppur piccole, differenze presenti tra il kernel Android e il kernel Linux, prestando particolare attenzione alle incongruenze relative alle varie librerie posix utilizzate da UmView.

Paradigmi di virtualizzazione

Virtualizzazione Totale



Con virtualizzazione totale si intende l'intera virtualizzazione dell'architettura della macchina, partendo quindi dall'emulazione del processore per quindi avere come risultato un laboratorio virtuale sul quale sperimentare.
L'esistenza di queste *macchine virtuali* é richiesta dal fatto che esse risultano estremamente comode (nonostante la normale perdita di prestazioni) in determinati frangenti.

Chiari esempi di macchine virtuali totali risultano essere qemu e AVD[1] che utilizzeremo in questa tesi.
Come si puó facilmente evincere da un minimo di configurazione di queste

[1] android virtual device

macchine virtuali, la maggior parte di esse é in grado di adattare alla nostra architettura, la quale prende il nome di macchina host, o ospite, un architettura eterogenea, é disponibile, ad esempio, un AVD ARM per le macchine x86, in modo da rendere piú comodo il debug e il test di applicazioni per questa architettura.

Una nota va invece al problema di effettuare un collegamento dalla macchina virtuale alla macchina host.
Essendo due entitá completamente separate, instaurare una connessione tra esse é sempre abbastanza difficile, normalmente, esistono vari programmi in grado di creare ponti tra la macchina virtuale e la macchina host, passando nella maggior parte dei casi per la rete della macchina host stessa.
Il programma specifico per il collegamento dell'Android Virtual Device alla macchina host prende il nome di Android Debug Bridge (o, in versione compatta, adb).

Come si evince facilmente dall'immagine sottostante, la macchina virtuale (che vediamo sulla destra), collegata all'host tramite l'android debug bridge, possiede un architettura eterogenea (nel caso specifico possiede un processore ARM, notare il terminale in basso), rispetto all'architettura della macchina ospite (piu' precisamente x86, la quale e' esplicitata nel terminale in alto), oltre a possedere un differente sistema operativo.

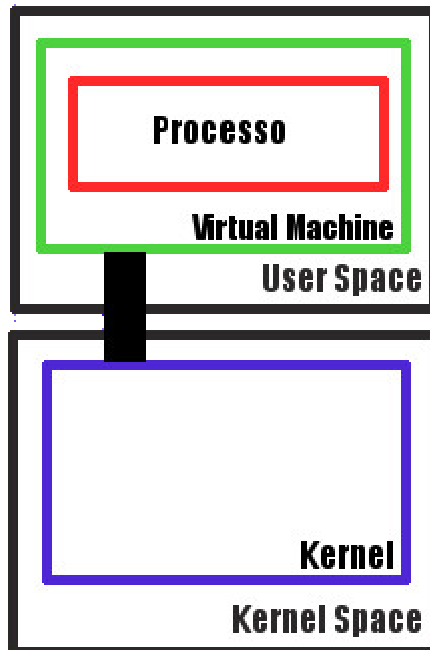
```
fpu             : yes
fpu_exception   : yes
cpuid level     : 10
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov c
lflush dts acpi mmx fxsr sse sse2 ss ht tm pbe nx constant_tsc arch_perfmon bts a
perfmperf pni monitor est tm2 xtpr pdcm dtherm
bogomips        : 3325.05
clflush size    : 64
cache_alignment : 64
address sizes   : 32 bits physical, 32 bits virtual
power management:

bera@White-Rabbit(0)~> cat /proc/cpuinfo|egrep "model name|cpu family"
cpu family      : 6
model name      : Genuine Intel(R) CPU           T2300  @ 1.66GHz
cpu family      : 6
model name      : Genuine Intel(R) CPU           T2300  @ 1.66GHz
bera@White-Rabbit(0)~>
```

```
bera@White-Rabbit(0)~/tesi4> ./adb shell
* daemon not running. starting it now on port 5037 *
* daemon started successfully *
# cat /proc/cpuinfo
Processor       : ARMv7 Processor rev 0 (v7l)
BogoMIPS        : 211.35
Features        : swp half thumb fastmult vfp edsp neon vfpv3
CPU implementer : 0x41
CPU architecture: 7
CPU variant     : 0x0
CPU part        : 0xc08
CPU revision    : 0

Hardware        : Goldfish
Revision        : 0000
Serial          : 0000000000000000
#
```

Virtualizzazione legata all'esecuzione di processi



É presente un'altra forma di virtualizzazione classica la quale è principalmente legata all'esecuzione di processi, questa forma è sviluppata per l'unificazione e l'eliminazione delle barriere presenti tra i vari sistemi operativi, un classico esempio risulta la java virtual machine[2].

L'esecuzione del codice quindi è presa in carico dall'istanza della macchina virtuale, la quale si occuperá dell'interpretazione del codice e di interagire con il sistema operativo sottostante.

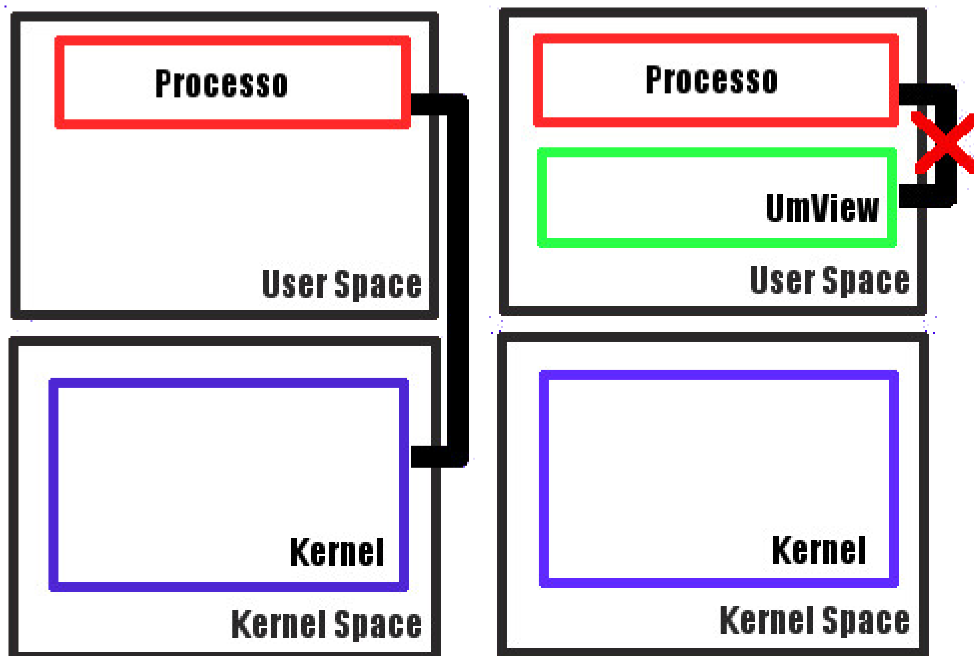
Il sistema operativo Android usa una particolare implementazione della java virtual machine, la quale prende il nome di *Dalvik Virtual Machine* sulla quale vengono eseguiti i programmi lanciati dal livello utente (come ad esempio, l'interfaccia dei messaggi o skype).

Vedremo successivamente i problemi che questa politica di virtualizzazione introduce, in Android, rapportandosi con le altre famiglie di macchine

[2]la macchina virtuale totale precedentemente citata qemu possiede una modalitá nella quale agisce in un modo molto simile a questa famiglia di macchine virtuali

virtuali.

Virtualizzazione Parziale



Con virtualizzazione parziale si intende la forma di virtualizzazione utilizzata dal progetto ViewOS, questa particolare forma di virtualizzazione si pone in una strada intermedia tra le precedenti, modificando quindi il comportamento del sistema operativo ospite senza intaccare la struttura dello stesso, risultando un ambiente separato a se stante.

Tramite questo meccanismo é possibile dirottare le chiamate ad un singolo file (sia che questo sia un file o un file speciale, descrittore di device) per poi processarle autonomamente.

Questo torna utile se si volesse, ad esempio, dirottare una chiamata *open* diretta al file '*/etc/passwd*' sul file '*/etc/host*' oppure creare un device driver a livello utente, o addirittura creare un file che descriva una rete[3].
Il lato interessante di queste specifiche macchine virtuali é la semplicitá con la

[3]vedremo piú avanti il comando mstack, un implementazione di questo concetto, in grado di controllare questo specifico comportamento della virtual machine

quale il processo riesce a coinvivere sia con l'ambiente emulato sia con l'ambiente ospite, aumentando, tramite semplici moduli, le potenzialitá del sistema operativo sottostante, risultando estremamente versatili nel caso l'implementazione diretta sul kernel del sistema operativo risulti troppo complessa.

Un implementazione di queste macchine virtuali risulta essere UmView, che sará proprio il fulcro di questa tesi nei prossimi capitoli.

Altre forme di virtualizzazione

Esistono varie altre forme di virtualizzazione, letteralmente, in informatica, ciò che è virtuale è una trasparenza di ciò che non esiste, come ad esempio una macchina virtuale non è realmente una macchina, ma bensì un programma che emula una macchina reale, allo stesso modo possono esistere file systems virtuali, devices virtuali, sistemi virtuali e addirittura, molto alimentate in letteratura, realtà virtuali.

Una grossa famiglia di macchine virtuali non trattate in questa tesi risultano le macchine virtuali integrate nel kernel, queste, ad esempio KMV o KmView risultano molto più performanti delle loro controparti user level, e adatte a emulazioni veloci.

Dall'altro lato della medaglia però si ha una difficoltà molto maggiore ad esempio a fornire un etereogenità delle architetture, per esempio, KVM funziona solo in modalità 1:1 con l'architettura su cui risiede, inoltre c'è una maggiore difficoltà di sviluppo legata a queste macchine, mentre la programmazione a livello utente è molto più user friendly, queste macchine risultano più difficili e impegnative da debuggare.

Paravirtualizzazione e hypervisors

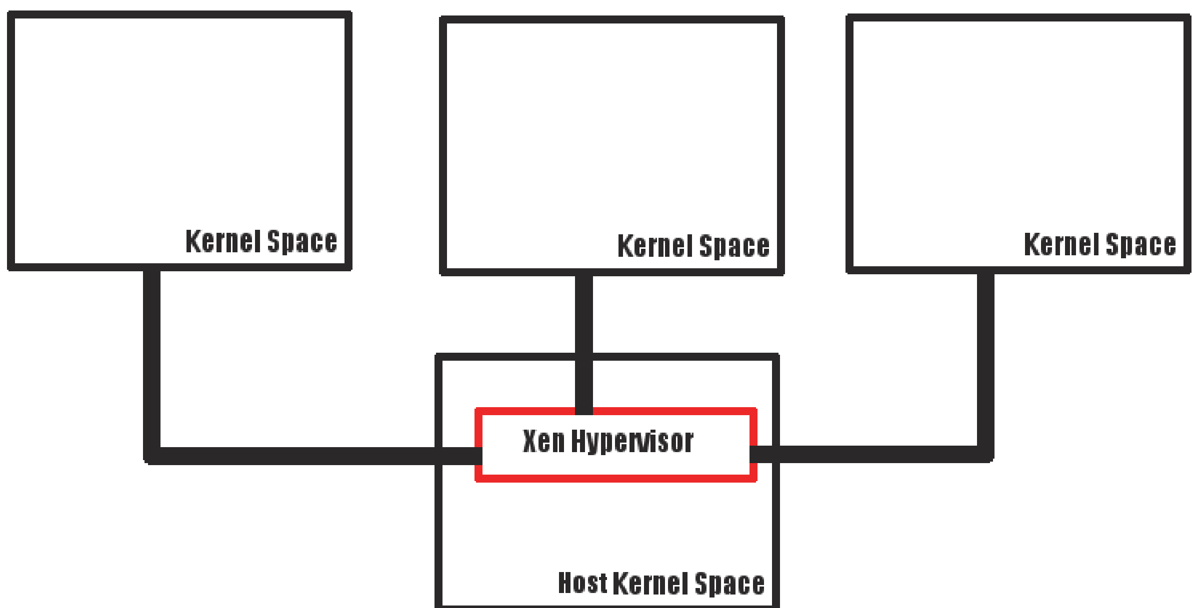
Esiste una forma di virtualizzazione chiamata paravirtualizzazione che occupa un ottica molto diversa dalle classiche introdotte, ma entra in un ottica molto simile in effetti alla virtualizzazione parziale.

Prendiamo d'esempio quindi il software capostipite di questa forma di virtualizzazione, che viene distribuito sotto il nome di *XEN*.

La paravirtualizzazione è considerata come una virtualizzazione di diversi sistemi operativi sulla stessa macchina, i quali prendono il nome di *domini*[4] essi risultano essere le istanze dei sistemi operativi virtualizzati.

La paravirtualizzazione quindi risulta essere più che una virtualizzazione intesa in senso lato, ma bensì più incentrata verso la gestione delle risorse presenti nella macchina a bassissimo livello, un concetto simile a quello di UmView, piuttosto che ad una qualsiasi virtualizzazione totale.

L'*hypervisor*[5] è invece il modulo back-end della nostra ParaVirtual Machine implementato a livello kernel che si occupa quindi di dialogare con il front-end in modo da gestire efficentemente le risorse messe a disposizione dal kernel sottostante.



Tra i pro di questa virtualizzazione possiamo senza ombra di dubbio elencare le prestazioni risultanti da questo approccio, senza costi di overhead dovuti ai vari approcci di virtualizzazione, costituendo un sistema virtualizzato estremamente performante.

[4]il termine originale risulta domains

[5]traducibile in italiano come supervisore, ma questa definizione risulterebbe ambigua per alcuni aspetti

I contro di questo approccio sono invece considerevoli, da un lato troviamo la necessità dell hypervisor nel kernel del sistema operativo, cosa in alcuni casi assolutamente non banale, e dall'altro lato, la necessità del supporto da parte del sistema operativo virtualizzato delle procedure di paravirtualizzazione, dovendo quindi implementare su ogni sistema che si desidera virtualizzare le suddette procedure nel caso non fossero disponibili.

Esiste tuttavia un metodo per far scomparire la necessità di queste procedure al sistema virtualizzato utilizzando un supporto da parte della *CPU* che prende il nome di *istruzioni di virtualizzazione*[6].

Queste istruzioni creano una vera e propria gestione della virtualizzazione ad un livello estremamente basso, in modo da avere prestazioni estremamente elevate in ogni contesto.

[6]I nomi proprietari risultano essere VT-X (Intel) e AMD-V (AMD)

Umview

Come visto, quindi, UmView rientra nella categoria delle macchine virtuali parziali, l'obiettivo di UmView è quello di offrire una gestione delle system calls a livello utente.

Esistono diversi metodi differenti dal progetto UmView, ma sempre all'interno del progetto ViewOS per implementare questa gestione, per esempio KmView o Vloader, mentre il primo è un implementazione a livello kernel il secondo è un metodo molto giovane presente nei meccanismi di umview, per questo si rimanda alla tesi di Federico Pareschi, presente nelle note bibliografiche.

All'interno di UmView

Introduciamo come lavora internamente quindi la versione classica di UmView, abbiamo già introdotto che la macchina virtuale in questione *dirotta* le system calls lanciate dai suoi figli per gestirle in modo autonomo.

Questo particolare comportamento è possibile grazie alla presenza di una particolare system call presente nei sistemi posix chiamata ptrace.

ptrace

```
#include <sys/ptrace.h>
long ptrace (enum __ptrace_request request,
        pid_t pid,
        void *addr,
        void *data);
```

Ptrace è una system call nata principalmente per il debugging, ma nasconde in se uno dei più potenti metodi di modifica dei processi esistenti nei sistemi operativi posix.

Questa potentissima possibilità è data dal fatto di poter scrivere (tramite le richieste *PTRACE_SETREGS* e *PTRACE_POKETEXT*)
nelle aree di memoria riservate al nostro processo tracciato (identificabile tramite il campo *pid*), costruendo un complesso ambiente attorno a questa semplice logica l'algoritmo del nucleo di UmView si riduce a questo piccolo schema:

- Lancia una fork() e prepara l'ambiente del figlio per fargli eseguire il comando passato come parametro.

- Traccia il figlio appena creato (il figlio quindi lancerà una *PTRACE_TRACEME*).

- all'arrivo di una system call da parte del figlio tracciato, controlla all'interno della tabella interna se presente un riferimento a quel numero di system call.

- Nel caso il punto precedente non restituisca funzioni allora la system call non viene processata da UmView, altrimenti viene lanciata la funzione associata.

Le funzioni associate inoltre possono essere *caricate dinamicamente* tramite il comando *um_add_service*, potendo avere quindi *moduli* intercambiabili per configurare il sistema dinamicamente.

Ovviamente questa gestione delle system calls appesantisce notevolmente il sistema (principalmente per l'utilizzo della system call ptrace, implementata in modo molto macchinoso), questo problema è uno dei punti più delicati dello sviluppo della macchina virtuale citata, per il quale rimandiamo a Vloader o al wiki ufficiale del progetto.

UmNet

UmNet è il modulo presente in UmView riguardante le interfacce di rete, è in grado (tramite il meccanismo sopra descritto) di catturare le system calls di rete e di associarle quindi ad un singolo file (che sarà quindi quello dove andremo a montare il sotto-modulo relativo alla nostra gestione.

Umnetnull

Il modulo che andremo ad utilizzare come esempio nei prossimi capitoli e che prenderemo come base per i moduli è umdevnull, come facilmente intuibile dal nome, questo modulo ha come unico comportamento quello di far fallire ogni system call dirottata.

Vediamo ora nel dettaglio il codice sorgente.

umnetnull.c

```
int umnetnull_msocket (...){
        /* questa printf e' stata aggiunta per
         * rendere l'esempio successivo
         * piu' chiaro
         */
        printf("Hijacked␣msocket␣syscall\n");
        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,
};
```

Analizzando riga per riga, la prima cosa che si dovrebbe notare è il fatto che la struttura *umnet_operations* ricorda molto quella di una *classe* nel senso informatico del termine, difatti racchiude come campi le 3 funzioni dichiarate subito sopra, queste sono difatti:

- Il costruttore del nostro modulo (*.init*) che viene lanciato una volta che viene fatto *mount* dello stesso.

- Il distruttore del nostro modulo (*.fini*) che viene lanciato una volta che viene fatto *umount* dello stesso.

- Il gestore delle nostre system calls di rete (*.msocket*) gestite tramite il comando *mstack* che introdurremo tra poco.

Una volta che una sistem call sarà dirottata dalla macchina virtuale verso il nostro modulo vi seguirà un esecuzione della funzione associata.

umnet_operations

Come già introdotto la struttura umnet_operations è la destinazione delle system calls dirottate associate al modulo, questa funzione è molto più versatile del semplice utilizzo che se ne fa in umnetnull, difatti osservando la dichiarazione (presente in *include/umnet.h*) troviamo la seguente struttura[7] contenente quindi tutte le system call di rete *dirottabili*.

```
struct umnet_operations {
        int (*msocket) (...);
        int (*bind) (...);
        int (*connect) (...);
        int (*listen) (...);
        int (*accept) (...);
        int (*getsockname) (...);
        int (*getpeername) (...);
        ssize_t (*send) (...) ;
        ssize_t (*recv) (...);
        ssize_t (*sendto) (...);
        ssize_t (*recvfrom) (...);
        ssize_t (*recvmsg)(...);
        ssize_t (*sendmsg)(...);
        int (*setsockopt) (...);
        int (*getsockopt) (...);
        int (*shutdown) (...);
        ssize_t (*read) (...);
        ssize_t (*write) (...);
        int (*ioctl) (...);
        int (*close) (...);
        int (*fcntl) (...);

        int (*supported_domain) (...);
        int (*event_subscribe) (...);
```

[7] sono stati commentati opportunamente i parametri per motivi di spazio

```
        int (*ioctlparms) (...);
        int (*init) (...);
        int (*fini) (...);
};
```

Mstack

Se UmNet può essere considerato il *gestore* delle nostre system calls di rete, Mstack può essere considerato il *selettore* dello stack personalizzato che andiamo ad usare, tramite il l'esecuzione del comando seguente.

```
mstack <percorso del file relativo alla rete> comando
```

Il nostro comando lanciato verrà eseguito nell'ambiente virtuale avente come gestore delle system calls di rete il gestore montato sul percorso specificato come primo parametro.

Il funzionamento di *Mstack* si può riassumere in un semplice passo, quale associare tramite una, come viene chiamata nel progetto View-OS, *Virtual System Call* questa system call non esistente a livello kernel ma solo a livello utente viene dirottata da *UmView* verso la gestione del modulo montato.

Esempio: umnetnull

Vediamo ora l'esempio base del nostro modulo.

Innanzitutto servirà eseguire un istanza della nostra macchina virtuale, tracciando, per esempio una shell.

```
$ umview bash
umview init
```

a questo punto l'esecuzione all'interno della nostra shell sarà interamente tracciata, carichiamo quindi il nostro modulo d'interesse, UmNet, questo viene fatto istruendo UmView con una *Virtual System Call*, cioè una system call non esistente nel kernel, utile solo per comunicare alla macchina virtuale di caricare il nostro modulo.

```
$ 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*.

In questo momento le system calls sono tracciate ma non ancora *funzionanti* poichè non abbiamo specificato nessun comportamento non avendo caricato nessun modulo, ciò è possibile tramite la system call (e il relativo comando) *mount*.

```
$ mount -t umnetnull none /dev/net/null
```

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

Il passo successivo quindi risulta eseguire un qualsiasi comando con il nostro modulo

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

questo comando verrà quindi eseguito con il nostro stack di rete montato.

Esempio: umdevnull

Allo stesso modo delle reti è possibile virtualizzare singoli descrittori di devices associati quindi ai nostri driver user space,
Mentre il metodo di programmazione è analogo[8] l'interfaccia di esecuzione differisce principalmente nell'omissione del comando mstack, non essendo necessari metodi di selezione di uno stack di rete.

L'esempio inizia quindi sempre con l'instanziazione di una macchina virtuale

```
$ umview bash
umview init
```

dopodichè ci sarà quindi necessità di comunicare a UmView la nostra intenzione di caricare il modulo relativo ai device

```
$ um_add_service umdev
umdev init
```

a questo punto si procederà con il mount del sottomodulo selezionato[9] sul percorso desiderato.

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

A questo punto le system calls relative ai files (open, write, read, ioctl, lseek, close) saranno dirottate quindi al nostro sottomodulo.

[8] ovviamente cambiano le system calls dirottate

[9] è stato scelto il sottomodulo di test umdevnull per l'analogia con il modulo umnetnull

```
$ cat /dev/test
null_open        c 0 0 flag 8000
null_read        c 0 0 len 65536
null_release     c 0 0 flag 0
```

Obiettivi del progetto

Il progetto mira al porting della macchina virtuale UmView e dei suoi moduli al sistema Android, risultando, per svariati motivi, una scelta molto versatile per alcune espansioni a basso livello di questo giovane sistema operativo.

Il vero obiettivo del progetto é quello di riuscire a unificare le interfacce di rete presenti sul device in un unico file, in modo da avere un controllo completo sulla rete da un unico *net device driver* che scelga secondo vari criteri l'interfaccia da usare al momento opportuno.

I vari vantaggi del porting della macchina virtuale (che saranno ulteriormente approfonditi nella sezione futuri sviluppi) risiedono sia nella possibilità di elusione di vari limiti imposti dalle applicazioni di default presenti su Android o da vari limiti sottostanti alle applicazioni.
Alcuni esempi che sarebbe possibile realizzare tramite UmView:

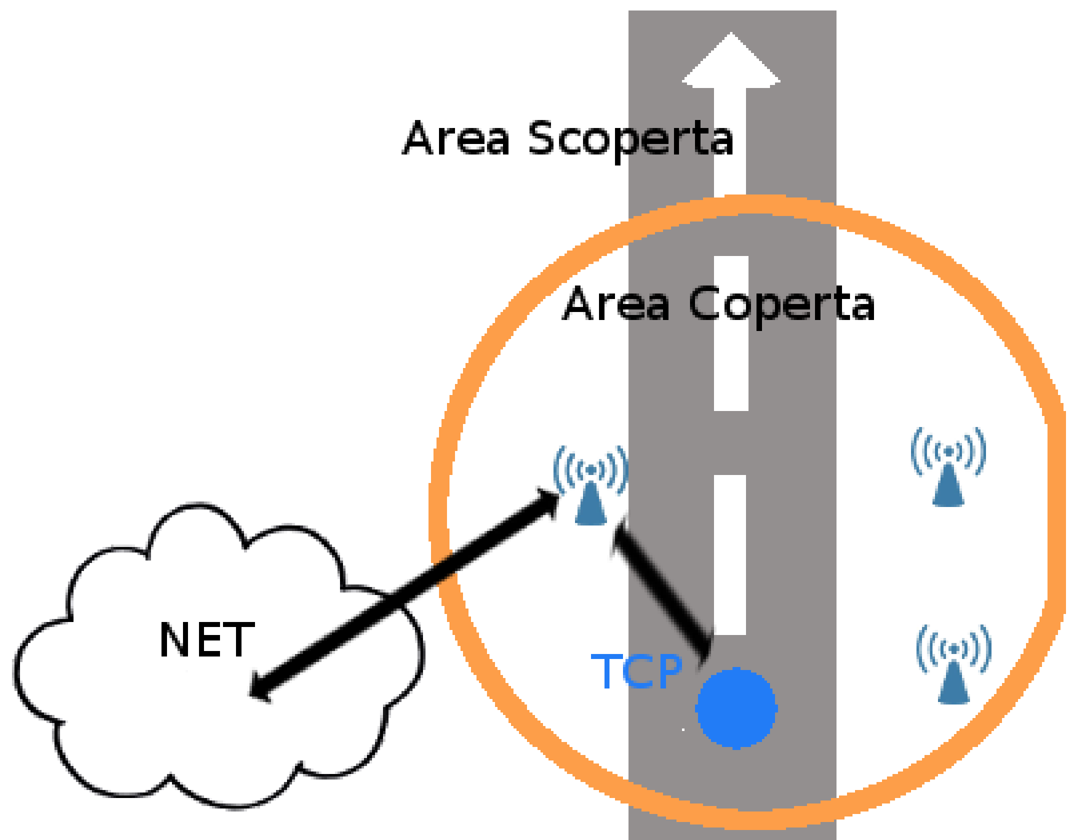
- un filtro *anti-spam* per gli sms
- un applicazione di registrazione delle chiamate
- un applicazione di broadcast di uno stream audio (tutti i device potrebbero riprodurre lo stesso stream audio)
- un file system virtualizzato a livello utente, contenente ad esempio la struttura classica del file system linux (/etc /home /dev ...)

Switch dell'interfaccia dinamico

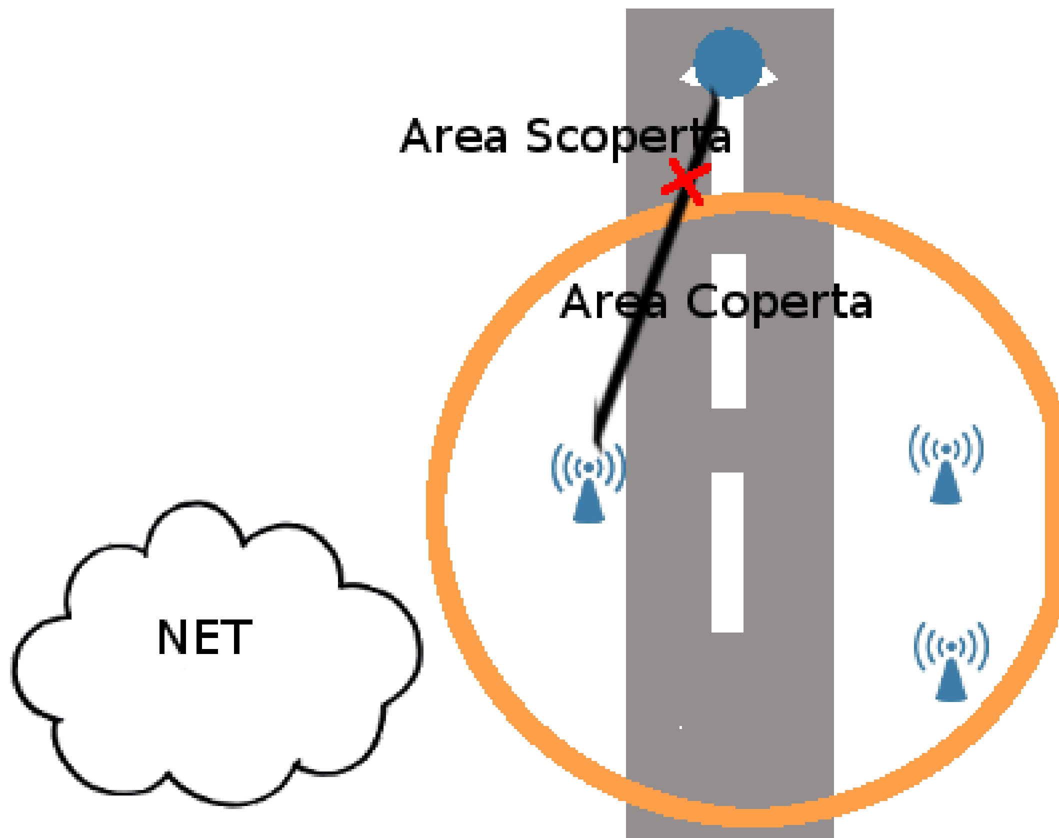
Supponiamo questo scenario, stiamo girando per una via minata di access points, dei quali abbiamo permessi di accesso, con il nostro device mobile, a questo punto avviamo, ad esempio una connessione TCP verso un server remoto, e supponiamo che questa resti attiva per un tempo considerevole

(ad esempio pensiamo di stare scaricando con *wget* un immagine di un disco della dimensione di 8 GB tramite una classica connessione 8Mb).

Ci troviamo quindi nella situazione descritta dalla seguente immagine

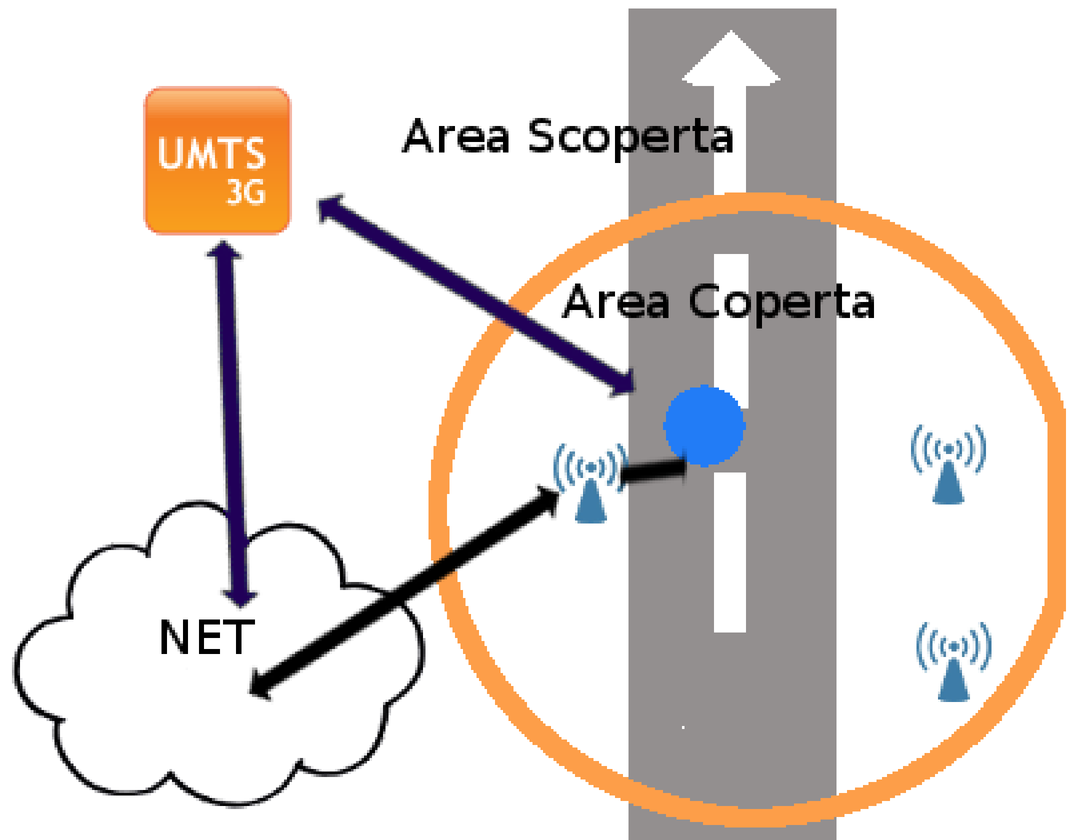


A questo punto supponiamo di uscire dalla zona coperta dalla connessione ai vari access points.



La connessione verso l'access point di conseguenza cade, interrompendo quindi il canale con l'applicazione e costringendo il device a passare sull'interfaccia UMTS e (se non sono disponibili metodi a livello applicativo) ristabilire la connessione e riabilitare lo stream di dati.

Utilizzando al suo posto invece un interfaccia virtuale, vista dall'applicazione, mentre nell'astrazione sottostante lasciamo il nostro modulo a dialogare con entrambe le interfacce, il nostro sistema non lascerebbe cadere la connessione TCP ma continuerebbe non accorgendosi dello switch dell'interfaccia e quindi mantenendo aperto lo stream.



Questo obiettivo sarà, però raggiungibile solamente una volta considerati anche i meccanismi che stanno alla base della riconsiderazione dell'indirizzo *IP* differente delle due interfacce, sia da parte della gestione dei pacchetti relativa al nostro sottomodulo di UmNet, sia da parte del *proxy* o *server*; in modo da gestire questa operazione di *smistamento* e rendere la comunicazione sottostante trasparente alla nostra applicazione.

Queste operazioni saranno quindi analizzate una volta portato con successo sul sistema la macchina virtuale UmView.

Tools utilizzati al fine del porting

Il porting del codice di UmView può essere diviso in 3 fasi salienti, *Compilazione*, *Esecuzione*, *Debugging*.
Ognuna delle fasi citate ha quindi i suoi programmi specifici, perlopiù disponibili per la maggiorparte dei sistemi operativi.

Tools Compilativi

per la compilazione dell'intero codice (scritto in C) è stato utilizzato l'Android Native Development Kit (ndk) questo kit è una collezione di toolchains per la compilazione sul sistema Android, contenente molti programmi utili alla programmazione in C su questo sistema operativo; nel nostro caso ci interesseremo maggiormente al tool ndk-build.

Questa toolchain porta con se l'intera libreria C di Android (denominata *Bionic*) che possiede alcune differenze sostanziali con la classica *glibc* linux, per svariati motivi[10].

ndk-build

NDK-build è un utile script sviluppato per selezionare automaticamente, dalla cartella del progetto, la toolchain adatta per la cross-compilazione che si va ad effettuare, e per il settaggio automatico dei vari flag specificati per la compilazione dell'applicazione stessa.

[10] i quali saranno esplicati nel capitolo apposito

La classica struttura di un progetto adattato per la compilazione con l'ndk-build e' 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 del nostro progetto.

Nello specifico, nel file *Application.mk* sarà esplicitata l'architettura per la quale andremo a sviluppare e nel file *Android.mk* i file che andremo a compilare e le varie specifiche per essi.

Vediamo un piccolo esempio su che configurazione utilizzare per compilare il seguente file per Android ARMv7:

jni/test.c

```c
#include <stdio.h>
int main(int argc, char **argv)
{
        printf("Hello␣World\n");
        return 0;
}
```

jni/Application.mk

```
# specifica che stiamo
# sviluppando per processori ARM
APP_ABI:=armeabi-v7a
# specifica che stiamo
# sviluppando per la versione 14 di android
APP_PLATFORM:=android-14
```

jni/Android.mk

```
# cambia la variabile con il path locale
# in modo che i riferimenti che facciamo
# siano legati alla cartella in cui risiede il file
# ( jni )
```

```
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:=hello
# files da compilare per ottenere il file di output
LOCAL_SRC_FILES:=test.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)
```

una nota va alla direttiva *$(BUILD_EXECUTABLE)* la quale è il fulcro della compilazione, modificandola ad esempio in *$(BUILD_SHARED_LIBRARY)* si avrà quindi la compilazione di una libreria coindivisa, invece che un eseguibile.

Tools Esecutivi

Per il test del nostro programma è necessaria quindi l'esecuzione su un device ospite, sia questo reale o virtuale, in tal caso l'Android Software Development Kit (sdk)ci viene in aiuto con vari software, quelli di maggior interesse per il nostro progetto risultano essere l'Android Virtual Device (avd) e l'Android Debug Bridge (adb)

AVD

L'Android Virtual Device (AVD) è una macchina virtuale totale disponibile all'interno dell'Android SDK, è un tool estremamente utile nel caso non sia possibile per qualche motivo il test su un vero device.

La macchina virtuale è disponibile precompilata per le architetture ospite appartenenti alla famiglia x86 e x86_64, mentre, nella lista di architetture virtualizzate troviamo le principali architetture su cui Android è ufficialmente supportato, quali MIPS, ARM e x86.

Assieme alla grande lista di pro legati a questa virtualizzazione (quali ad esempio il test dell'applicazione con device speciali virtuali), è ovviamente presente una diminuzione considerevole delle prestazioni e una cospiqua richiesta di risorse dell'host.

adb

L'Android Debug Bridge è invece una suite di connessione tra la macchina sulla quale viene eseguita e il device (sia esso virtuale o reale).
È presente qui a seguito le operazioni più usate per la realizzazione del porting qui descritto[11].

- *push* - copia file sul device
- *pull* - copia file dal device
- *shell* - lancia la shell di default */sbin/sh*
- *forward* - esegue un forward del socket passato come primo parametro (per esempio tcp:55555) su di un socket del device[12].

lo script setEnv.sh

Per l'esecuzione degli eseguibili come utente e dislocati in una cartella differente dalle classiche cartelle designate ai binari, è necessario settare varie variabili d'ambiente.

Per un utilizzo più agevole quindi è stato scritto uno script bash per l'assegnamento automatico delle giuste variabili d'ambiente.

```
#!/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.sh
```

[11] per la lista completa si rimanda al comando *man adb*

[12] L'utilizzo di quest'ultimo comando, poco chiaro, è da ricercarsi nella sezione *Tools relativi al debug*

difatti il primo punto indicherà di utilizzare la shell locale come destinazione dei comandi.

Tools relativi al Debug

Per l'intero debug del codice portato su Android è stato utilizzato gdb, essendo uno dei più diffusi e completi debugger disponibili per il mondo GNU scrivere una guida esaustiva su questo tool sarebbe estremamente prolisso e complesso, per quanto rimandiamo alla documentazione ufficiale, esponendo un semplicissimo esempio ed elencando solo le differenze incontrate con il normale utilizzo di gdb sulla macchina ospite (avente quindi, oltretutto, la stessa architettura base).

Per utilizzare quindi il debugger nel nostro ambiente di sviluppo innanzitutto bisognerà utilizzare sulla macchina host (la quale si occuperà del debug remoto dell'applicazione) una differente versione di gdb in possesso della tabella di conversione dei simboli di debug relativi alla versione scelta di android[13].

In questo caso particolare si è scelto di utilizzare il debugger precompilato presente nell'Android NDK, *arm-linux-androideabi-gdb*.
Purtroppo la versione base di Android è sprovvista di default dell'intera suite gdb, ad eccezione di gdbserver, server grazie al quale è possibile effettuare il debug remoto dell'applicazione.

Vediamo quindi un piccolo esempio pratico[14] riprendiamo ora il nostro codice precendete *jni/test.c* il quale conteneva questa piccola riga:

jni/test.c

```
/*...*/
char *w="World";
/*...*/
```

il primo passo è quindi eseguire il nostro server gdb sulla macchina target (sia essa virtuale o emulata) su di una porta disponibile, passandogli come primo parametro il programma da eseguire[15].

[13] È possibile utilizzare un pacchetto disponibili per la maggior parte delle distribuzioni linux chiamato gdb-multiarch

[14] per comodità la shell della nostra macchina host sarà indicata con il seguente prompt $ e il nostro sistema target con prompt %

[15] ovviamente sarà necessario eseguire un *adb push src dst* prima di ciò!

```
$ adb shell
 # ora siamo nella macchina target
% gdbserver :55555 hello
process hello created; pid=xxx
Listening on port 55555
```

il nostro processo sarà quindi bloccato in attesa del debugger remoto dal quale ricevere istruzioni.
Torniamo quindi nella macchina host per il debugging vero e proprio, avviando *arm-linux-androideabi-gdb*.

```
 # e' necessario fare forward del socket tcp dalla
 # macchina virtuale alla reale.
$ adb forward tcp:55555 tcp:55555
 # si e' aggiunto hello come parametro per
 # avere i simboli di debug.
$ arm-linux-androideabi-gdb hello
(...)
(gdb)
```

A questo punto dovremo istruire gdb comunicandogli di porsi in modalità client.

```
(gdb) target remote :55555
Remote debugging using :55555
(gdb)
```

Ora, essendo all'interno del nostro debugger ed essendo legati all'applicazione remota, possiamo porre un breakpoint alla linea 5 *printf(...)*; [16]

```
(gdb) b 5
Breakpoint 1 at 0xxxxxxxxx: file test.c, line 5.
(gdb) r
Starting Program hello

Breakpoint 1 main() at test.c:5
5        char *w="World";
(gdb)
```

A questo punto è, per esempio, possibile stampare il valore assunto dalla variabile *w*.

[16] è possibile che le stampe di debug non siano presenti, ciò è dovuto al fatto di non aver compilato il programma con il flag *-g*

```
(gdb) print w
$1 = 0xxxxxxxxx "World"
```

Porting

Basandosi quindi sulla tesi di Alessio Siravo si è continuato il porting e adattato esso alla compilazione per architettura Android e processore ARM; ciò è stato possibile grazie a vari passi, qui di seguito elencati:

Porting del core

Il core è stato quindi *cross-compilato* per Android settando i vari flag di compilazione (come visto prima nelle sezioni relative all'Android.mk e all'Application.mk) e poi testato; si è però riscontrato un problema nella mancanza di varie strutture e direttive presenti nella glibc linux non presenti nella libc Android[17], queste, anche per scelta implementativa ereditata dal progetto originale sono state implementate in una nuova libreria chiamata *bionic_addon* dislocata dalla bionic, in modo che, nel caso che in un prossimo futuro la *bionic* venga aggiornata ufficialmente ciò non comporti problemi al nostro eseguibile e possa semplicemente essere rimossa la dichiarazione dalla libreria aggiuntiva e ricompilata solo quest'ultima, mantenendo così il sistema allineato.

Modifiche alla bionic

Le modifiche come già detto quindi sono state effettuate come libreria separata, difatti, osservando il file *Android.mk* si incontra la seguente direttiva di compilazione:

```
#bionic_addon
include $(CLEAR_VARS)
include $(LOCAL_PATH)/bionic_addon\
        /bionic_clone/syscalls.mk

LOCAL_MODULE:=bionic_addon
```

[17]Bionic

```
libc_common_src_files := bionic_addon\
        /bionic_clone/bionic_clone.c
# caused some problem with the arm port,
#       because it's x86 assembly
libc_common_src_files += bionic_addon\
        /bionic_clone/clone.S
libc_common_src_files += bionic_addon\
        /fmemopen/fmemopen.c
libc_common_src_files += bionic_addon\
        /mempcpy/mempcpy.c
libc_common_src_files += bionic_addon\
        /pwd/getpwuid_r.c
libc_common_src_files += bionic_addon\
        /backtraces/backtrace.c
LOCAL_SRC_FILES := $(libc_common_src_files)

libc_common_cflags := \
        ...

LOCAL_CFLAGS := $(libc_common_cflags)

LOCAL_PRELINK_MODULE := false
include $(BUILD_SHARED_LIBRARY)
```

oltre ai vari include specifici dell'Android.mk notiamo un inclusione del file relativo alle system calls che aggiunge la compilazione di un file assembly relativo alla system call *clone()*, questo codice è stato quindi inserito in versione ARM, trovato dai sorgeti ufficiali di Linux.
Notiamo infatti nella variabile *libc_common_src_file* l'inclusione della *bionic_clone.c* la quale fa da wrapper per la system call dichiarata nel file *clone.S*. oltre alla suddetta system call sono stati aggiunti varie chiamate che non erano presenti quali la *fmemopen*, la *mempcpy*, la backtrace e la *getpwuid_r*.
Mentre se ciò è tutto per quanto riguarda le chiamate a funzioni (quindi dichiarate nei file *.c*) è tutto un altro discorso quello riguardante le strutture dati non presenti nella libc e non utilizzate a livello utente.
Queste difatti sono state dichiarate nella cartella *include_adds* nella quale sono presenti i vari files *.h* che fanno da sostituti ai vari files reali della bionic. Questi difatti contengono le strutture non dichiarate nella bionic, le quali non sono utilizzate da system calls a livello utente verso il kernel, ma che, i programmi che vengono tracciati all'interno di UmView ovviamente possono

usare, per questo UmView si preoccupa di utilizzarle, anche se non vengono utilizzate dal sistema operativo[18].

Lista delle aggiunte alla Bionic

Funzioni aggiunte:

- Aggiunta una suite *fasulla* per compatibilità con *backtrace.c*, quali le funzioni:

```
int backtrace(void **buffer, int size)
{
        return -ENOSYS;
}
char **backtrace_symbols(void * const *buffer,
        int size)
{
        return NULL;
}
```

- Ereditate le funzioni *fmemopen* dal precedente porting.
- Ereditate la funzione *mempcpy* dal precedente porting.
- Ereditata la funzione *clone* dal precedente porting, riscritta in assembly ARMv7.
- Implementata la funzione *getpwuid_r* senza supporto LDAP[19].

Headers adattati/aggiunti:

- Aggiunti le definizioni dei limiti del sistema ereditati dal sistema linux nel file *limits.h*

```
#define NR_OPEN 1024

#define PATH_MAX        4096
#define XATTR_* ...
#define IOV_MAX 16
```

[18] e quindi giaceranno inutilizzate, almeno fino a implementazioni future della Bionic
[19] questo meccanismo di login non è implementato in Android

```
#ifdef __LP64__
        #define LONG_BIT 64
#else
        #define LONG_BIT 32
#endif
```

le ultime due definizioni sono relative invece al semplice fatto che la libreria andrebbe a sovrascrivere la libreria di default della bionic, quindi non definerebbe la costante *LONG_BIT*.

- Aggiunto l'header *dirent.h* che semplicemente fa da link verso il file *linux/dirent.h*.

- Aggiunto l'header *pwd.h* che contiene la struttura passwd utilizzata dalla funzione sopracitata *getpwuid_r*.

- Creato un header aggiuntivo con un enum *fasullo*, questo enum dovrebbe contenere le richieste ptrace, che in Android sono comunque contenute in costanti definite dal preprocessore, risulta quindi inutile avere questo enum in questo sistema, ma per compatibilità con UmView si utilizza per non modificare il codice interno.

```
enum __ptrace_request
{
        ptrace_dummy_request,
};
```

- Ridefinito il tipo *__fd_mask* come *long int* non essendo presente in Bionic nell'header *select.h* (ed utilizzato quest'ultimo come link a *sys/select.h*.

- Aggiunto l'header *sys/utsname.h* nel quale sono è stata inserita la struttura *utsname* e dichiarata la corrispondente costante:

```
#define _UTSNAME_LENGTH 64
struct utsname
{
        ...
}
```

- Aggiunto l'header *linux/utsname.h* nel quale sono state inserite le funzioni mancanti relative alle strutture **_utsname*

- Definita una funzione di minimo tra due variabili nell'header *sys/add.h*

```c
#define MIN(a,b) ({a<b?a:b;})
```

- Riscritta la struttura *timex* nel relativo header.
- Aggiunto l'header *param.h* con al suo interno le costanti non definite da bionic:

```c
#define NOFILE          256
#define MAXSYMLINKS     256
```

- Dichiarate all'interno del file socket.h le costanti:

```c
#define SOCK_CLOEXEC    0x0
#define SOCK_NONBLOCK   0x0
```

  non supportate da UmNet

- Dichiarati all'interno del file *types.h*:

```c
#define __off64_t size_t
#define off_t size_t
struct iovec
{
        void *iov_base;
        size_t iov_len;
};
```

Modifiche al codice di UmView

Sono quindi stati modificati tutti i riferimenti alla gerarchia del file system classica presenti nel codice di UmView, ad esempio i riferimenti alla cartella classica *tmp* sono stati modificati in */data/local/tmp*.

È stato inoltre rilevato un problema relativo ad una non inizializzazione di un semaforo, difatti mentre, probabilmente, l'implementazione linux della funzione posix *pthread_rwlock_** ammette l'assegnamento del valore al semaforo alla creazione della variabile associata, mentre l'implementazione Android necessita invece della funzione *pthread_rwlock_init(...)* la quale è stata quindi modificata per ammettere quindi maggiore portabilità in UmView.

LibCap e LibMhash

Si è riscontrato necessario il porting delle librerie dinamiche *libcap* e *libmhash*, rispettivamente le librerie relative alle capability Linux e all'utilizzo di varie tabelle di hash, si è scritto quindi un piccolo Android.mk per rendere i tre progetti a se stanti, per utilizzarle come porting separato nel caso ulteriori progetti abbiano bisogno delle suddette librerie.

Sono state quindi inserite nel Makefile principale del progetto con una direttiva chiamata *PREBUILT_SHARED_LIBRARY* la quale non ha altri effetti se non quello di copiare automaticamente il file indicato nella cartella output del progetto.

Riportiamo qui quindi lo spezzone relativo del Makefile:

```
#mhash
include $(CLEAR_VARS)
LOCAL_MODULE:=libmhash
LOCAL_SRC_FILE:mhash/libmhash.so
include $(PREBUILT\_SHARED\_LIBRARY)

#capability
include $(CLEAR_VARS)
LOCAL_MODULE:=libcap
LOCAL_SRC_FILES:=capability/libcap.so
include $(PREBUILT_SHARED_LIBRARY)
```

Il file defs_arm_android.h

È stato invece necessaria la scrittura di un nuovo header, i files *arch_defs.h* sono infatti i files relativi alle system calls mancanti, il file *defs_arm_um.h* è sì presente nel core del programma, ma esso è relativo alla classica infrastruttura linux, difatti si preoccupa di dichiarare come non esistenti le system calls non presenti per l'implementazione del kernel linux per la suddetta infrastruttura di cpu.

Il discorso per Android in questo caso è molto diverso, non avendo, per scelta implementativa, ¨ ereditato¨ dal kernel linux varie[20] system calls, que-

[20]un numero considerevole, circa un centinaio!

ste devono essere segnalate a UmView come non esistenti[21], semplicemente dichiarando a *__NR_doesnotexist* il valore della system call; vediamo ora un piccolo esempio:
Nel caso io non abbia la system call *nice* dovrò dichiarare all'interno del suddetto file di dichiarazione:

```
#define __NR_nice          __NR_doesnotexist
```

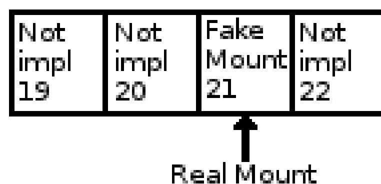
ciò consentirà a UmView di comprendere che la system call non è installata nel sistema.

Il file defs.h

A questo punto si è quindi dovuta quindi abilitare la selezione del suddetto file, chiamato in un file *wrapper* più grande, è stato quindi aggiunto un *#ifdef* all'interno del suddetto file wrapper chiamato *defs.h*

Il problema delle system calls mancanti

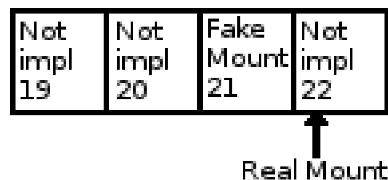
A questo punto sorge quasi di rito una domanda, ma perchè tutto questo?
Semplicemente ciò è dovuto al fatto di come *ptrace* e di conseguenza UmView gestisce il dirottamento delle system calls, ptrace semplicemente utilizza il numero della system call chiamata come identificativo di essa.
Possiamo quindi immaginare che per varie ragioni (come ad esempio, il fatto di non avere una libreria allineata o non raggiungibile a lato utente con i giusti identificativi delle system calls) UmView usi quindi un dizionario le cui chiavi risultano proprio questi valori.
Possiamo infatti immaginare questo dizionario come un grande array dove ad ogni numero di system call è associata la funzione relativa da chiamare al suo posto, o nel caso, lasciar passare la vera system call.
Facciamo un piccolo esempio grafico:



[21] ad esempio non sono implementate la maggior parte delle system calls sys-V

in questo caso notiamo che la system call mount è indirizzata in modo giusto alla sua controparte tracciata, in un certo senso il sistema si comporta bene poichè è allineato al vero sistema operativo, e quindi riesce a tracciare la system call che lui sa essere in posizione 21.

Consideriamo ora il caso in cui il, nella nostra implementazione di umview non abbiamo dichiarato che la system call in posizione 17, che per comodità chiameremo *mysystemcall*, non è presente utilizzando il metodo descritto precedentemente, in questo modo il vettore si troverebbe shiftato avanti di una posizione, nella quale, le system call seguenti si troverebbero in posizione sbagliata, senza venir quindi reinderizzate alle controparti virtuali corrette[22].



Questo appunto è proprio il nostro caso, ed è proprio questo il motivo della scrittura di un nuovo file di allineamento per le system calls.

Porting delle utility (um_cmd)

Il porting delle utility è stato estremamente semplice, scrivendo semplicemente le opportune direttive di compilazione dentro al makefile Android.mk Difatti la maggior parte di essi risultano semplici programmi applicativi di qualche centinaia di righe, il quale unico scopo è quello di fare parsing di vari parametri a linea di comando e lanciare una system call [23] verso il descrittore (file) specificato.

Porting di UmNet

Durante la compilazione del core di UmNet e dei suoi sottomoduli è risultato necessario il linking tra esso e le principali funzioni di UmView.
Non essendo il nostro sistema di sviluppo (basato su make) basato su un tool come *./configure* è risultato molto più semplice dal punto di vista pratico

[22]Nella figura verrebbe fatta passare la vera mount, senza poter in nessun modo tracciarla.

[23]sia questa reale o virtuale, vedi il capitolo su umview

di adattamento del codice e della compilazione la creazione di una libreria dinamica chiamata *libumcore* contenente al suo interno le funzioni comuni a tutti i vari moduli, essa poi è stata specificata come linkata sia all'eseguibile UmView, sia alle varie compilazioni dei moduli interni.
È inoltre stato creato un modulo AndroidNetTest che contiene al suo interno (oltre alla classica definizione del modulo UmNetNull) una piccola printf di test, da utilizzare affiancata al debug con gdb.

Il problema della DALVIK VM

La DALVIK virtual machine è come già 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[24], 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 possibilità di tracciare qualsiasi programma presente all'interno dell'interfaccia grafica.

L'unica plausibile soluzione è quindi riuscire a tracciare la Dalvik Virtual Machine principale, lanciandola quindi come processo figlio di UmView.

essa però è 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 però 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,

[24] una sorta di *clone()*
[25] per fortuna piccoli stavolta!

dovremmo quindi formalizzare un metodo con il quale UmView sappia riconoscere i due thread particolari, visto che a questo punto ogni instanza della Dalvik è tracciata allo stesso modo.

Sviluppi Futuri

Rimanendo nell'ambito del porting il prossimo passo è sicuramente quello di portare i moduli *umdev* e *ummisc* all'architettura Android.

Un passo molto proficuo invece risulterebbe quello di portare il progetto Vloader (e quindi la modifica delle librerie linkate dinamicamente) su questo porting, in modo da migliorare le prestazioni offerte dalla macchina virtuale parziale.

Riguardo al refactoring e alla riconcezione di alcuni punti del funzionamento di UmView per adattarlo ad Android invece sono punti focali i vari problemi elencati in precedenza, quindi considerare il fatto di eseguire il programma come utente senza i privilegi di root e quindi l'installazione quasi *trasparente* per l'utente medio, e, forse ancor più difficoltoso, il fatto di rieseguire la Dalvik VM in modo che venga tracciata dalla nostra macchina virtuale e legarla ad un ambiente dinamico dove avere più libertà rispetto ai vincoli imposti dal sistema.

Un altro passo per niente banale è legato al fatto di avere un sistema operativo principalmente legato all'interfaccia grafica; sviluppare un semplice front-end che ci permetta di escludere il terminale per eseguire applicazioni tracciate direttamente dall'interfaccia grafica può risultare estremamente difficoltoso, dovuto alla necessità che la nostra applicazione sia figlia di un istanza *UmView*. Altrimenti fornire un meccanismo che *faccia adottare suo figlio* ad UmView[26] non è immediatamente ovvio su sistemi Unix-like come Android.

Entrando invece nell'ottica dell'obiettivo finale dell'intero progetto, il primo passo logico incontrato è quindi relativo alla creazione del modulo relativo alla gestione delle interfacce di rete, dopo un accurata analisi dello scenario

[26]in un modo simile con cui il processo init adotta i processi orfani su linux

e dei problemi relativi alle comunicazioni di rete non considerati in questa tesi.

Conclusioni

Il porting è risultato quantomeno efficace come *proof of concept* del possibile utilizzo di progetti C complessi sul sistema Android e della sua[27] compatibilità con il suo ¨ capostipite ¨ Linux.
La mancanza di moduli specifici per Android ora è un passo evolutivo direttamente diramato da questa tesi, dovendo studiare quindi caso per caso la compatibilità del modulo con il sistema Android e i suoi problemi del non essere pensato per l'esecuzione di programmi sistemistici scritti in linguaggio C.

I problemi quindi sono un punto focale di questo aspetto, risultando in un progetto estremanente efficace e utile nel caso i nostri gradi di libertà coincidano con quelli che il sistema impone alla macchina virtuale.

Si spera quindi in un sollecitamento da parte degli sviluppatori Android, verso maggiori gradi di libertà a livello di sistema, dovuto all'espansione a macchia d'olio dei porting/progetti nativi in C su Android.

[27]seppur in qualche caso lieve

Bibliografia

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

Relativi alla virtualizzazione parziale

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

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

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

[5] View-OS source code: http://sourceforge.net/projects/view-os/

Relativi all'architettura Android e alla programmazione in C su di essa

[6] Giacomo Bergami: Pjproject su Android: uno scontro su più livelli.

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

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

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

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

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

Altri riferimenti

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

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

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

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

[16] XEN: http://www.xenproject.com

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

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

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

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

Ringraziamenti

Volevo ringraziare tutte le persone che sono rimaste al mio fianco in questo percorso,

Ringrazio innannzitutto i miei genitori e la mia famiglia per avermi sostenuto in ogni modo fino a questo momento.

Ringrazio il professor. Vittorio Ghini per la sua infinita pazienza per ogni quesito da me postogli e per la sua disponibilità, oltre all'importantissima preparazione dovuta ai suoi insegnamenti/consulti.

Ringrazio il professor. Renzo Davoli per l'avermi appassionato all'ambito della programmazione relativa ai sistemi operativi.

Ringrazio il docente Piero Pasotti per l'avermi insegnato a programmare e l'avermi indirizzato verso questa disciplina.

Ringrazio infine tutti i miei amici/colleghi di studi che per un motivo o per l'altro ci sono sempre stati.

Quindi grazie, a chiunque abbia mai creduto anche un po in me, chiunque leggerà questa tesi, e anche a chi non ci ha mai creduto.

Dedico questa tesi a tutte queste persone, sperando che questa bellissima serie di esperienze passata in questi anni non termini solo con questo documento.

Davide Berardi