Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea Triennale in Informatica

Esecuzione di applicazioni all'interno di una macchina virtuale su Android.

Tesi di Laurea in Architettura degli Elaboratori

Relatore:
Chiar.mo Prof.
Ghini Vittorio

Presentata da:
Siravo Alessio

Sessione III
Anno Accademico 2011/2012

Imparare è un'esperienza,
tutto il resto è solo informazione.

Albert Einstein

Indice

1 Introduzione **9**

2 Prerequisiti **13**

2.1 Metodi di virtualizzazione 13

2.1.1 Introduzione 13

2.1.2 Classificazione 14

2.1.3 Macchine virtuali di sistema 15

2.1.4 Macchine virtuali di processo 17

2.1.5 Considerazioni 21

3 Scenario **23**

3.1 Livello fisico e data-link 24

3.1.1 Wi-Fi 24

3.1.2 IEEE 802.11 24

3.1.3 Architettura 25

3.1.4 Livello MAC 25

3.2 Livello Rete 28

3.2.1 IPv4 28

3.2.2 IPv6 34

3.3 Livello Trasporto 39

3.3.1 Protocollo UDP 40

3.4 Livello Applicazioni 41

3.4.1 VoIP 42

4 Architettura 47

4.1 VoWIFI 47

4.1.1 802.11e 48

4.1.2 802.11r 50

4.1.3 Considerazioni 50

4.2 Always Best Packet Switching 51

4.3 Architettura 52

4.3.1 Algoritmo di selezione delle interfacce Wi-Fi 56

5 Obiettivo 59

5.1 Scenario Nodo Mobile 59

5.1.1 Android 61

5.1.2 Linux Network Stack 61

5.2 Considerazioni 65

6 Strumenti 67

6.1 umView 67

6.1.1 Struttura 67

6.1.2 Moduli 68

6.2 Android Native Development Kit 72

6.2.1 Utilizzo dell'NDK 74

6.2.2 Bionic 78

7 Deployment 79

7.1 Esempio 79

7.2 Considerazioni 82

8 Progettazione 83

8.1 Porting 83

8.1.1 Modifiche al codice sorgente 84

8.1.2 Libreria implementata 93

9 Valutazione **97**

9.1 Bionic_addon 98

9.1.1 Fmemopen test 98

9.1.2 Clone test 99

9.2 umView 101

9.2.1 Stato attuale del porting 102

10 Conclusioni e sviluppi futuri **105**

Capitolo 1

Introduzione

Nel corso degli ultimi anni l'esplosione del fenomeno Internet ha rivoluzionato numerosi aspetti della società, portando ad un'evoluzione senza precedenti delle tecnologie di comunicazione. Oggi è infatti possibile reperire qualsiasi tipo di notizia o dato attraverso la rete, chiunque può rendere pubblici i propri contenuti e le proprie idee utilizzando semplici strumenti come blog e social network ed il ruolo svolto dall'indiviuo all'interno dell'informazione è dunque cambiato radicalmente, passando dall'essere spettatore passivo ad utente attivo coinvolto in prima persona. Internet ha inoltre permesso di superare i limiti di comunicazione imposti dalle distanze geografiche, introducendo scenari e classi di servizi altrimenti impensabili come email, chat e comunicazioni VoIP (Voice over Internet Protocol).
In particolar modo, quest'ultima tecnologia consente di instradare una conversazione vocale attraverso una qualunque rete basata su protocollo IP, comportando dunque un abbattimento dei costi rispetto ad un contratto telefonico tradizionale ed offrendo la possibilità di usufruire di servizi avanzati come conversazioni video e teleconferenze.
L'esigenza di avere un accesso costante alla rete Internet ha portato inoltre allo sviluppo di dispositivi mobili come smartphone e tablet ed a tecnologie per l'implementazione di reti wireless; il WiFi, con lo standard IEEE 802.11, si è imposto in questo senso come soluzione di riferimento, garantendo eleva-

te prestazioni e ottenendo dunque una notevole diffusione. Per tale motivo l'utilizzo di applicazioni VoIP su dispositivi mobili attraverso tecnologie wireless risulta essere sempre più frequente, definendo tuttavia un insieme di problematiche che è necessario affrontare perchè tale servizio possa divenire di uso comune. 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 dunque mostrata un'architettura per il supporto alla mobilità in contesti eterogenei in cui più sistemi e protocolli sono coinvolti; il modello presentato permette di mitigare i problemi sopracitati consentendo l'utilizzo contemporaneo, all'interno di un terminale mobile, di differenti interfacce di rete. Sarà inoltre definito un meccanismo di virtualizzazione per consentire l'integrazione dell'architettura proposta in dispositivi mobili equipaggiati con sistema operativo Android, effettuando il porting di una macchina virtuale attraverso la quale eseguire l'applicazione VoIP; in questo modo sarà possibile ottenere, all'interno del dispositivo stesso, una specifica gestione del flusso di dati prodotto dall'applicazione.

Nel **capitolo 2** verrà introdotto il concetto di virtualizzazione, classificando le differenti metodologie ed implementazioni.

Nel **capitolo 3** si farà una panoramica dello scenario in cui si agisce, ci si soffermerà sulle tecnologie wireless e sui protocolli che a tutti i livelli sono coinvolti nelle coversazioni veicolate sull'Internet Protocol.

Nel **capitolo 4** verrà descritta l'architettura proposta per mitigare i problemi del VoIP su reti wireless, l'uso contemporaneo di più interfacce e i vari moduli di cui è composto il sistema.

Nel **capitolo 5** sarà descritto l'obiettivo del presente lavoro, definendo il problema e introducendo la soluzione scelta.

Nel **capitolo 6** verranno evidenziate le caratteristiche degli strumenti utilizzati, con particolare attenzione alle relazioni che questi hanno con la presente tesi.

Nel **capitolo 7**, attraverso un esempio pratico, verrà descritta una possibile implementazione dello scenario.

Nel **capitolo 8**, si mostrerà nel dettaglio il lavoro di progettazione svolto, evidenziando le modifiche apportate agli strumenti e le funzionalità introdotte.

Nel **capitolo 9**, saranno discussi i test effettuati per valutare la bontà del sistema, dimostrandone validità e correttezza.

Nel **capitolo 10**, verranno infine evidenziati i possibili sviluppi futuri applicabili per estendere e migliorare il lavoro svolto.

Capitolo 2

Prerequisiti

2.1 Metodi di virtualizzazione

2.1.1 Introduzione

In questo capitolo verranno esaminati nel dettaglio gli innumerevoli significati che il concetto di virtualizzazione ha assunto negli anni in base al suo contesto d'uso. A partire da una generica definizione del termine *virtuale* presente nel dizionario[1], è facile notare come in ogni ambito scientifico questo termine assuma una particolare accezione: in filosofia diventa ad esempio sinonimo di potenziale mentre in fisica lo si associa ad una possibile evoluzione di un sistema.

In Informatica il termine virtualizzazione fa riferimento alla possibilità di astrarre una risorsa, creando un'interfaccia esterna che nasconda la parte sottostante e permetta l'accesso concorrente alle risorse da parte di più istanze che funzionano in contemporanea. Assumendo quindi di possedere un determinato strumento caratterizzato da una propria interfaccia (ovvero da un'insieme di operazioni offerte), la sua virtualizzazione consiste in un oggetto che fornisce la stessa interfaccia dell'originale, ma che potrebbe tuttavia avere forma e funzionamento interno differenti.

[1] "Ciò che è in potenza e non in atto"[14]

Il concetto di virtualizzazione è strettamente correlato a quello di emulazione: emulare un oggetto consiste infatti nel crearne uno nuovo, diverso dal precedente, ma che permetta di effettuare le stesse operazioni. Diametralmente opposta è invece l'idea di simulazione, che permette di realizzare un oggetto che si comporta solo a livello macroscopico come l'originale senza però realmente compiere le operazioni richieste. Nella simulazione, inoltre, le operazioni sono scandite da un tempo logico, al contrario dell'emulazione in cui è utilizzato un orologio reale.

2.1.2 Classificazione

Oggi l'emulazione è possibile in varie modalità, e diversi sono i criteri secondo i quali classificare i metodi di virtualizzazione. È infatti possibile analizzare e distinguere le macchine virtuali secondo tre aspetti fondamentali:[9]

- **Comunicazione tra VM e macchina reale**: questo aspetto esamina come la macchina virtuale si interfaccia all'architettura sottostante e comprende due gruppi principali:
  - **Virtualizzazione dell'architettura del processore**: il livello di astrazione è costituito da un intero ambiente hardware, o da un interprete di tutte le istruzioni macchina eseguite dal sistema operativo o dalle applicazioni da esso controllate. Questo tipo di virtualizzazione permette di creare macchine virtuali complete.
  - **System call trapping**: in questo caso vengono intercettate le chiamate di sistema, ovvero l'interfaccia di programmazione implementata dal sistema operativo per eseguire operazioni privilegiate. Tale sistema risulta essere solitamente più performante della virtualizzazione dell'architettura, ma meno portabile.
- **Completezza della virtualizzazione**: Indica quanto dell'hardware reale viene emulato dalla macchina virtuale. Possiamo avere:

- **Virtualizzazione del processore**: il sistema host accede direttamente a tutte le risorse, ad eccezione del processore che è emulato.
- **Macchina Virtuale Parziale**: i programmi sono eseguiti dal processore host e solo alcune parti del sistema reale sono virtualizzate. In questo caso non è necessario avviare un sistema operativo guest.
- **Virtualizzazione completa**: il sistema host è emulato completamente e tutte le risorse sono quindi virtualizzate. È necessario un sistema operativo guest per eseguire i programmi.

- **Invasività nel sistema operativo host**: Indica quanto il sistema operativo reale sia influenzato dall'esecuzione del sistema virtuale.
  - **Livello Utente**: qualunque utente può eseguire la macchina virtuale, che opera senza permessi privilegiati.
  - **Livello Superutente**: solo chi possiede privilegi amministrativi sulla macchina reale può utilizzare il sistema virtualizzato.
  - **Patch al kernel**: la macchina virtuale richiede una modifica al sistema operativo, di solito attraverso l'uso di patch applicate al kernel stesso.

In base alla tassonomia appena introdotta, a seconda del livello di astrazione in cui la virtualizzazione occorre è possibile distinguere *macchine virtuali di sistema* e *macchine virtuali di processo.* [13]

2.1.3 Macchine virtuali di sistema

Per macchina virtuale di sistema si intende un software in grado di emulare completamente l'hardware di una macchina reale; questa tipologia di VM è usata per eseguire un sistema operativo completo (detto sistema guest) e su di esso programmi applicativi. Solo in alcuni casi si ha necessità di

emulare l'intero hardware reale, come ad esempio nel caso si voglia eseguire il sistema operativo su una diversa architettura; se invece l'architettura della macchina reale e di quella emulata coincidono, la necessità di una virtualizzazione completa dipende unicamente dal set di istruzioni da virtualizzare. In concomitanza con la diffusione delle macchine virtuali di sistema si possono individuare tre differenti ambiti di utilizzo principali:

- **Individuale**: utilizzate solitamente per l'esecuzione contemporanea di differenti sistemi operativi su uno stesso computer.

- **Aziendale**: utilizzate per l'esecuzione di vari server con differenti configurazioni di sistema su di una singola macchina in modo da diminuire i costi per l'acquisto di hardware e i consumi elettrici.

- **Servizi di hosting**: per offrire ai clienti una quantità di macchine superiore a quelle realmente possedute, diminuendo i costi e permettendo, in caso di richieste specifiche, la riconfigurazione hardware di un server host senza dover apportare modifiche all'hardware reale.

Esistono numerose implementazioni di macchine virtuali di questo tipo; le principali sono:

- **QEMU**: software open source, permette di emulare completamente un'architettura o di virtualizzare unicamente il processore. Nel caso in cui venga utilizzato come macchina virtuale completa, effettua una traduzione dinamica dell'Istruction Set del processore emulato, generando una nuova istruzione per ogni comando della CPU guest e riutilizzandola all'occorrenza.

- **Bochs**: software open source per l'emulazione di architetture x86 e AMD64. Permette di eseguire numerosi sistemi operativi come Linux, DOS o Microsoft Windows.

- **VirtualBox**: sviluppato da Oracle, è un software di virtualizzazione proprietario per architettura x86 e AMD64/Intel64. Ne esiste inoltre una versione ridotta distribuita secondo i termini della licenza GPL[2].

- **VMWare**: progetto proprietario, permette la virtualizzazione di macchine complete su architetture x86.

2.1.4 Macchine virtuali di processo

Lo spazio logico di memoria, l'insieme di istruzioni e i registri del processore utilizzabili costituiscono la rappresentazione di una macchina dal punto di vista di un processo; i dispositivi di I/O sono visibili ad esso unicamente tramite l'interfaccia fornita dal sistema operativo, costituita dall'insieme delle system call.[13] Per *SystemCall Virtual Machine* si intende un software in grado di porsi tra il sistema operativo ed il processo da eseguire, di intercettare le chiamate di sistema fatte da quest'ultimo e di adattarle alle proprie necessità. Lo scopo di tali emulatori, di solito, non è incentrato sulla creazione di un ambiente virtuale da mostrare all'utente, ma sull'estensione delle funzionalità del sistema operativo.

User-Mode Linux

L'obiettivo di User-Mode Linux[3] è quello di rendere possibile l'esecuzione di un kernel Linux in modalità utente; gli utilizzi più comuni di questo sistema sono principalmente l'hosting di server virtuali, l'isolamento dei processi, l'utilizzo di ambienti differenti all'interno della stessa macchina reale e la creazione di sandbox, ovvero ambienti isolati al cui interno eseguire operazioni potenzialmente pericolose. In UML l'ambiente virtuale è ottenuto modificando opportunamente il kernel per rendere possibile la sua esecuzione come processo in user-mode; l'interfaccia di comunicazione è data dalle system call che vengono redirette al kernel reale piuttosto che direttamen-

[2] GNU General Public License

[3] http://user-mode-linux.sourceforge.net/

te all'hardware della macchina. Intercettazione e modifica delle system call vengono effettuate utilizzando la system call ptrace().

View-OS

Un esempio differente di virtualizzazione di processo è dato da View-OS[4], un progetto ideato e sviluppato dal Dipartimento di Scienze dell'Informazione dell'Università di Bologna. View-OS introduce un concetto nuovo di virtualizzazione, quello di processo avente una visione completamente personalizzata dell'ambiente in cui viene eseguito; ciò permette di aggirare una caratteristica fondamentale dei moderni sistemi operativi, ovvero il concetto di *visione globale* in cui tutti i processi in esecuzione su un dato sistema condividono la stessa "percezione" dei servizi offerti dal sistema stesso (ad esempio i processi utilizzano un unico stack TCP/IP condiviso per le operazioni di rete, e dunque lo stesso insieme di indirizzi IP e politiche di routing)[9]. Come precedentemente anticipato, con View-OS ciò non accade perchè è possibile fornire esplicitamente il particolare ambiente di esecuzione in cui lanciare il processo e dunque, a differenza di quanto accade in User-Mode Linux, non è necessario ricorrere ad una completa virtualizzazione del sistema operativo, ma bensì è possibile virtualizzare unicamente le risorse desiderate. View-OS offre quindi un approccio modulare alla virtualizzazione, mostrando al processo una macchina virtuale costituita dall'unione delle virtualizzazioni in uso. Le possibili applicazioni sono:

- **Emulazione di operazioni privilegiate**: creare, ad esempio, una rete virtuale la cui configurazione sia interamente gestibile dall'utente, senza andare a modificare la rete reale sulla quale si appoggia.
- **Sicurezza**: nascondere al processo porzioni di file system, reti o altri processi.

Attualmente esistono due differenti implementazioni di View-OS: *umView* e *kmView*; entrambe realizzano una macchina virtuale parziale simile a quella

[4] http://sourceforge.net/projects/view-os/

di UML ma differiscono per la system call utilizzata per tracciare le chiamate dei processi.

umView La system call usata per l'intercettazione delle chimate in questo caso è ptrace(); quest'ultima rappresenta l'interfaccia standard messa a disposizione dal kernel Linux ed essendo stata sviluppata originariamente per fornire uno strumento di debug non è ottimizzata per la virtualizzazione; per questo motivo presenta alcune limitazioni che tuttavia sono state superate con l'introduzione di caratteristiche aggiuntive:

- ptrace non supporta più di un tracer per ogni processo e ciò comporta l'impossibilità di richiamare programmi che utilizzano ptrace all'interno di umView. L'ultima versione della Virtual Machine include tuttavia un'implementazione virtuale di ptrace che permette di ovviare a questo problema.

- Nonostante con ptrace sia possibile modificare le system calls intercettate, non è possibile ignorarle. Questa limitazione è stata superata con l'introduzione di un tag, PTRACE_SYSVM, che permette di ignorare la system call corrente.

- ptrace permette di caricare o leggere dati dalla memoria del processo controllato ma ogni chiamata è in grado di trasferire solo una word (4 o 8 bytes a seconda dell'architettura di riferimento). Gli sviluppatori di umview hanno proposto un nuovo tag per ptrace, PTRACE_MULTI, che permette di trattare differenti chiamate di ptrace come una singola system call atomica. Questa modifica ha permesso di aumentare in maniera sensibile le prestazioni di umview.[8]

Umview risulta essere per i motivi sopracitati prestazionalmente non ottimale ma, basandosi su ptrace, è utilizzabile su kernel Linux Standard, e quindi potenzialmente su qualsiasi sistema operativo fondato su tale kernel.

kmview Questa implementazione di View-OS è basata sull'uso della system call utrace(), un' infrastruttura per il monitoraggio di threads che permette di definire un meccanismo di intercettazione kernel-based.

Utrace permette di superare implicitamente quelle che sono le limitazioni di ptrace descritte in precedenza offrendo a kmview prestazioni migliori; tuttavia per utilizzare utrace è necessario un modulo del kernel specifico non presente all'interno del kernel Standard di Linux; questa caratteristica pregiudica quindi l'utilizzo di kmview in ambienti sprovvisti nativamente di tale modulo (come ad esempio Android).

Macchine Virtuali di applicazione

Per completezza di informazione verrà ora introdotto il concetto di macchina virtuale di applicazione, una tipologia di emulazione che mira a rendere il codice di un applicazione portabile. Per application virtual machine si intende, infatti, un software necessario all'esecuzione di un programma scritto in un linguaggio di programmazione non comprensibile dalla macchina sottostante ma compilato per la virtual machine; quest'ultima può fungere da interprete, eseguendo per ogni istruzione del programma un set di istruzioni proprio, o da compilatore: in questo caso avviene, all'avvio o durante l'esecuzione del programma (compilazione junst-in-time), una traduzione del codice sorgente. A questo punto è possibile eseguire il programma compilato su qualsiasi architettura per la quale esista la relativa application virtual machine, garantendo la portabilità dell'applicazione.

Il linguaggio più comune che fa uso di questo tipo di macchina virtuale è Java, il cui codice sorgente viene compilato e tradotto in bytecode, codice nativo della Java Virtual Machine (JVM).

Android stesso è basato su un particolare tipo di application virtual machine, la *Dalvik Virtual Machine* (DVM), ovvero il software che esegue le applicazioni sui dispositivi Android. I programmi, solitamente scritti in Java e compilati in bytecode, vengono poi convertiti dal formato compatibile con la JVM (.class) ad un formato specifico per la DVM (.dex) prima dell'effet-

tiva installazione sul dispositivo; questo poichè il formato compatibile con la DVM risulta essre più compatto e maggiormente adatto all'esecuzione su sistemi limitati in termini prestazionali.

2.1.5 Considerazioni

Alla luce delle informazioni raccolte ed analizzate nei precedenti paragrafi, si è scelto di utilizzare per il presente lavoro la macchina virtuale *umView*, ai cui dettagli implementativi è dedicato il capitolo 6.

Capitolo 3

Scenario

Nel seguente capitolo verranno esaminati nel dettaglio i livelli che costituiscono l'architettura alla base della comunicazione nei sistemi mutlihomed eterogenei, con particolare attenzione ai protocolli direttamente coinvolti nell'architettura oggetto del capitolo 4.

Per *sistemi multihomed eterogenei* si intendono quei dispositivi dotati di più interfacce di rete, come ad esempio Wi-Fi o UMTS, tecnologie che al giorno d'oggi sono sempre più spesso integrate in un unico dispositivo come smartphone e tablet. Queste tecnologie vengono usate nelle comunicazioni Internet per accedere ai più disparati servizi (ad esempio applicazioni multimediali, navigazione web e comunicazione vocale e video). Internet è una rete decentralizzata che si basa su una multitudine di meccanismi e protocolli; questi devono essere standardizzati e regolati in modo che i nodi nella rete possano comunicare senza le limitazioni dovute all'eterogeneità dei dispositivi presenti. Lo standard ISO/OSI [20] definisce un modello di architettura strutturato in livelli ognuno dei quali si occupa di specifici aspetti delle comunicazioni, fornisce funzionalità al livello superiore e utilizza le astrazioni del livello inferiore. In questo modo è possibile ridurre la complessità non banale delle comunicazioni di rete.

3.1 Livello fisico e data-link

3.1.1 Wi-Fi

Una rete LAN wireless (WLAN - wireless local area network) consiste in una rete formata da due o più dispositivi che, grazie a tecnologie di trasmissione radio, possono comunicare tra loro senza l'utilizzo di cavi. La tecnologia permette comunicazioni in un'area limitata, consentendo mobilità all'interno di essa senza perdere l'accesso alla rete. I vantaggi sono ampi e nei più disparati contesti. In ambito privato, le reti senza fili hanno avuto un massivo utilizzo grazie alla facilità di installazione e alla crescente diffusione di portatili, palmari e smartphone. Alcuni tipi di esercizi come coffee-shop o centri commerciali hanno iniziato ad offrire questo tipo di servizio ai clienti. Nelle città sono nati progetti per piccole reti pubbliche, ad esempio quelle di biblioteche e università, oppure grandi reti civiche che coprono le vie del centro. La tecnologia di gran lunga più diffusa per la creazione di reti wireless è senza dubbio quella Wi-Fi.

3.1.2 IEEE 802.11

Il Wi-Fi si basa sulla famiglia di standard IEEE 802.11, che specifica un insieme di regole da seguire per il livello fisico e data link del modello ISO/OSI per permettere l'interoperabilità tra i dispositivi dei diversi produttori esistenti. Come indicato dal numero, IEEE 802.11 si adegua perfettamente agli altri standard 802.x per reti locali wired e le applicazioni che lo utilizzano non dovrebbero notare nessuna differenza logica, mentre è possibile una degradazione delle performace. All'interno della famiglia, i protocolli dedicati alla trasmissione delle informazioni sono a, b, g e n. Gli altri standard riguardano estensioni dei servizi base e miglioramenti di servizi già disponibili. Il primo protocollo (estremamente diffuso) è stato lo b; in seguito si sono diffusi il protocollo a e soprattutto il protocollo g. Recentemente, con il protocollo n, le performace teoriche sono state notevolmente aumentate da 54

Mb/s a 600 Mb/s [5] grazie all'uso della teconologia MIMO (Multiple-Input Multiple-Output).

3.1.3 Architettura

Le reti wireless possono usare due diverse architetture: ad-hoc, nella quale i nodi mobili (Mobile Node, MN) comunicano direttamente tra loro, o basata su infrastruttura, dove ogni nodo comunica con una stazione radio base (Base Station, BS), detta comunemente access point (AP). Questa stazione si occupa di instradare i dati verso il destinatario finale, direttamente se è associato alla stessa Base Station, oppure ad altri Access Point nel caso di una rete cablata o mesh. Quando un nodo mobile cambia posizione allontanandosi dalla copertura dalla propria Base Station e avvicinandosi ad un'altra, avviene il cosiddetto roaming attraverso il quale il nodo mobile viene associato alla nuova stazione. Poiché questo avviene a livello 2 del modello ISO/OSI, l'indirizzo IP rimane lo stesso e quindi le connessioni presenti nel nodo restano intatte.

3.1.4 Livello MAC

I compiti principali del livello MAC (Media Access Control) [3] sono: regolamentare l'accesso al medium, frammentare i dati, gestire le modalità di risparmio di energia ed applicare la crittografia.

Accesso al canale Solo un nodo alla volta può trasmettere all'interno del proprio raggio d'azione. Una collisione avviene se un dispositivo riceve contemporanemanete trasmissioni (detti frame) da due nodi diversi (in questo caso i dati inviati sono confusi tra loro e il ricevente non riesce ad utilizzarli). L'accesso al canale è regolato dal protocollo CSMA/CA (Carrier Sense Multiple Access with Collision Avoidance), una modifica del CSMA/CD (CSMA/Collision Detection), che non può essere usato nelle reti senza fili a causa della difficoltà di realizzazione di un apparato di ricetrasmissione

che possa contemporaneamente trasmettere ed ascoltare sullo stesso canale radio. Prima di trasmettere un nodo si mette in ascolto: se il canale è libero per un determinato periodo di tempo (detto DIFS) la trasmissione può avvenire, mentre se il canale è occupato, il nodo sceglie un intervallo di tempo casuale (detto random backoff) per il quale attendere e poi riprovare. Il nodo ricevente legge il dato e controlla che sia stato ricevuto senza errori; in questo caso attende un intervallo prefissato di tempo (SIFS, più piccolo del DIFS, quindi a maggiore priorità) per spedire al mittente un ACK di consegna correttamente avvenuta. Se nessun ACK è stato ricevuto il mittente ritrasmette il dato secondo una sequenza prestabilita, dovendo comunque competere nuovamente per l'accesso al canale. Il meccanismo, così descritto, non consente dunque ad ogni nodo di accedere al canale in modo equo; IEEE 802.11 supera questo problema facendo scegliere al mittente un random backoff soltanto al primo tentativo di accesso, per poi decrementare il tempo di attesa finché non gli è consentito trasmettere.

Problema dei terminali nascosti Una modalità di accesso al canale così descritta è soggetta al problema dei terminali nascosti: due nodi, disposti in modo che abbiano raggio di ricezione non sovrapposto e raggio di trasmissione intersecante, cercano di trasmettere contemporaneamente ad un terzo nodo che si trova nell'area di trasmissione comune ai precedenti due. Con questa disposizione, un nodo mittente non può determinare se il canale è occupato o meno e quindi nel nodo ricevente si verificano un gran numero di collisioni. Per superare ciò, lo standard definisce dei meccanismi opzionali che usano due pacchetti di controllo, RTS e CTS. Invece di inviare direttamente un frame dati, un nodo trasmette un RTS (Request To Send) che contiene il mittente, il ricevente della trasmissione e la durata. Se il destinatario riceve l'RTS, risponde con un CTS (Clear To Send) che contiene gli stessi dati; il canale viene quindi "prenotato" e i dati di prenotazione sono propagati anche alla stazione radio nascosta al mittente, che si pone in attesa per la durata della trasmissione come specificato nei frame di controllo. Le collisioni

possono ancora avvenire, ad esempio quando i due mittenti inviano un RTS contemporaneamente, ma essendo questo pacchetto molto piccolo (da 0 a 2347 ottetti), la probabilità è più bassa. Se il frame dati da inviare è più piccolo dell'RTS, un nodo può decidere di inviare direttamente il pacchetto senza utilizzare questo meccanismo di prenotazione.

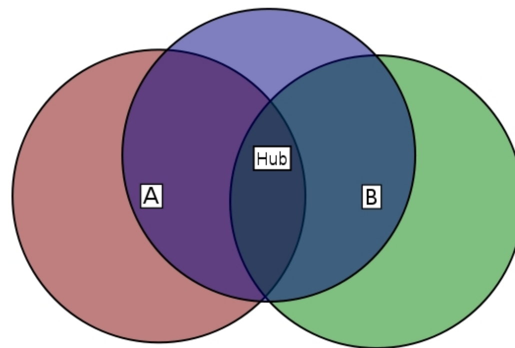

Figura 3.1: I due nodi A e B hanno raggio di ricezione separato, quando inviano dati all'hub contemporaneamente, avviene una collisione.

Risparmio energetico I nodi mobili non hanno una fonte stabile di energia ma utilizzano batterie; un aspetto importante è quindi risparmiare il più possibile energia per allungare i tempi di operatività tra una ricarica e l'altra. IEEE 802.11 include una modalità dedicata a questo aspetto indicando per ogni nodo due possibili stati: *sleep*, quando la ricetrasmittente è spenta e *awake* se completamente funzionante (gli access point si trovano generalmente sempre nello stato awake). Quando un access point deve inoltrare dati ad un nodo mobile in stato sleep, li bufferizza finché il nodo mobile non risulta essere di nuovo attivo. Ad intervalli determinati, i nodi sleep si "svegliano" e gli access point possono annunciare le destinazioni dei frame bufferizzati; se un nodo mobile è tra essi, rimane awake fino alla corretta ricezione.

Sicurezza Il mezzo di trasmissione del Wi-Fi è percepibile da tutti senza restrizioni fisiche e la sicurezza è quindi un aspetto da non trascurare. Diversi standard basati su crittografia sono stati sviluppati per superare il problema.

Le versioni originali dei protocolli 802.11 erano fondati sulla crittografia WEP (Wired Equivalent Privacy) basata su un cifrario a flusso con 40 bit per le chiavi e 24 bit per il vettore di inizializzazione che non deve mai ripetersi per la corretta implementazione dell'algoritmo; 24 bit non sono però sufficienti e in una rete con molto traffico potrebbero essere usati più volte rendendo WEP molto insicuro, tale da poter essere superato in pochi minuti [7]. WPA (Wi-Fi protected access) [1] cerca di superare i problemi di WEP allungando il numero di bit usati per la chiave ed includendo un meccanismo per cambiare dinamicamente il vettore di inizializzazione.

3.2 Livello Rete

Il livello Rete è il terzo livello nella pila ISO/OSI, esso si occupa dello scambio logico di pacchetti tra due nodi arbitrari, che in generale non sono direttamente connessi, ovvero non hanno un collegamento diretto tra di loro e si possono trovare in reti diverse, spesso anche basate su tecnologie differenti. Il livello rete si occupa dunque di indirizzamento e instradamento (routing) verso la giusta destinazione attraverso il percorso di rete più appropriato. È quindi importante per questo livello conoscere la topologia di reti e sottoreti per potere scegliere il giusto percorso attraverso di esse, soprattutto nel caso in cui sorgente e destinazione si trovino in reti differenti. Sono stati creati diversi protocolli per questo livello, come IPX e AppleTalk, ma quello di gran lunga più usato è il protocollo IP, declinato nella versione 4 e 6.

3.2.1 IPv4

IPv4, come descritto nell'RFC 791 [16] dell'IETF del 1981, è un protocollo senza connessione per l'uso su reti a commutazione di pacchetto, come ad esempio Ethernet. Opera con un modello di tipo best effort, non garantisce cioè la consegna né assicura che i pacchetti inviati, detti datagrammi, siano recapitati in ordine o duplicati; questi aspetti infatti sono controllati dal livello superiore nella pila di protocolli.

Indirizzamento IPv4 usa indirizzi a 32 bit per identificare un host e lo spazio di indirizzamento è così fissato a 4.294.967.296 possibili indirizzi. In realtà, molti di questi sono riservati per scopi speciali come reti private e indirizzi multicast, riducendo quindi il numero di indirizzi che possono essere potenzialmente allocati per il routing nella rete pubblica.

Il numero di indirizzi IP pubblici si sta esaurendo ad una velocità che non era stata anticipata quando il protocollo è stato ideato[1]. L'esaurimento è dovuto a vari fattori tra cui il rapido incremento del numero di utenti di internet e degli apparati constantemente connessi, come i modem ADSL e i nuovi dispositivi mobili come smartphone e PDA. Alcune soluzioni temporanee sono state adottate per mitigare il problema come l'introduzione delle Network Address Translation (NAT), l'uso di reti private, l'uso del DHCP, un controllo più stretto da parte dei RiR nell'assegnare blocchi di indirizzi e talvolta il reclamo di blocchi assegnati ma non utilizzati; tuttavia l'unica soluzione definitiva resta il passaggio ad IPv6, che ha allargato lo spazio di indirizzamento a 128 bit e che verrà ampiamente analizzato nel paragrafo successivo.

Un indirizzo IPv4 è di solito rappresentato con gruppi di quattro numeri separati da punti (es. 127.0.0.1) ed è diviso in due parti: la prima, definita nei bit più significativi, indica la rete di appartenenza mentre la seconda identifica l'host. Originariamente le reti erano rappresentate da un numero di bit prefissato che permetteva di realizzare una classificazione (ad esempio una rete di classe A è identificata da 8 bit e può indirizzare 24 bit per gli host); oggi, per avere maggiore flessibilità, una rete definisce il numero di bit ad essa dedicati, ad esempio la rete 192.168.0.0/16 ha 16 bit usati per l'indirizzamento degli host. Un indirizzo IP non risulta valido ai livelli inferiori e la sua associazione ad una interfaccia di rete, identificata a livello data link da un indirizzo MAC, è effettuata attraverso il protocollo ARP.

[1] il 3 febbraio 2011, lo IANA (Internet Assigned Numbers Authority, l'organismo che ha responsabilità nell'assegnazione degli indirizzi IP) ha consegnato gli ultimi cinque blocchi di indirizzi /8 ai RiR (Regional Internet Registry) dei cinque continenti [19]

Reti private e NAT Dei circa quattro miliardi di indirizzi disponibili in IPv4, tre gruppi di indrizzi sono riservati per reti private: 172.16.0.0/12 192.168.0.0/16 e 10.0.0.0/8; queste reti non sono instradabili da indirizzi al di fuori di esse né i nodi all'interno possono comunicare con l'esterno se non con il meccanismo NAT. Il NAT è un metodo semplice per permettere a tutti i computer di una sottorete di dialogare con la rete pur non possedendo un indirizzo visibile dall'esterno. Il metodo consiste nell'avere un solo gateway con ip pubblico accessibile dall'esterno che presta il suo indirizzo per le comunicazioni di tutte le macchine della sottorete traducendo e tenendo traccia dei pacchetti in transito per poter recapitare le risposte ai legittimi destinatari. In questo modo si migliora la sicurezza rendendo le macchine interne alla rete non direttamente accessibili dall'esterno e si migliora l'utilizzo dello spazio di indirizzamento possibile poichè un'intera rete privata utilizza un solo indirizzo IP pubblico. Questa tecnica è utilizzata dalla maggior parte delle reti casalinghe.

DHCP Il Dynamic Host Configuration Protocol è un protocollo con il quale è possibile autoconfigurare i nodi di una rete eliminando il bisogno dell'intervento di un amministratore di rete. DHCP permette anche di avere un database centrale per tener traccia degli host che sono connessi alla rete, prevenendo la possibilità che due nodi abbiano lo stesso indirizzo. Oltre all'IP, DHCP permette l'assegnazione del server NTP (Network Time Protocol), quello DNS (Domain Name System) ed altri particolari.

ARP L'Address Resolution Protocol è usato per determinare, dato un indirizzo IP, il corrispettivo indirizzo hardware del livello data-link. Quando un host vuole spedire un datagramma ad un altro nodo conoscendo il suo indirizzo IP ma non quello fisico, effettua una richiesta ARP tramite un broadcast sulla rete di appartenenza: in questo modo l'host che possiede l'indirizzo IP richiesto comunica il proprio indirizzo MAC.

Formato del pacchetto Un paccehtto IP consiste di un header e di una sezione dati. Nella Figura 3.2 è riportata la sua struttura.

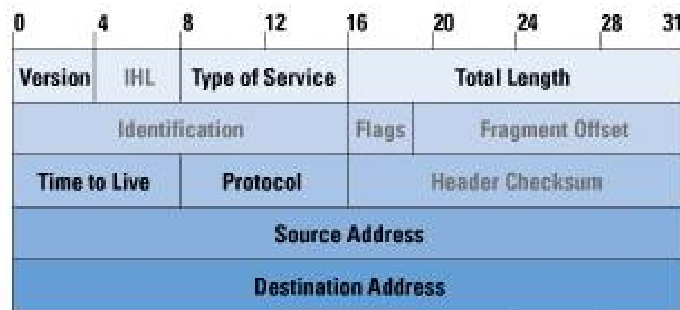

Figura 3.2: Formato del pacchetto IPv4.

Nell'header ci sono quattordici campi:

- **Version** indica la versione del datagramma IP.
- **Internet Header Length** indica la lunghezza (in word da quattro byte) dell'header del paccehtto IP.
- **Type of Service** specifica la precedenza con cui l'host deve trattare il datagram. Questo tipo di servizio è caduto in disuso e recentemente questi 8 bit sono stati ridefiniti ed hanno la funzione di Differentiated services (DiffServ nell'IETF e Explicit Congestion Notification (ECN) codepoints (vedi RFC 3168 [18])), necessari per le nuove tecnologie basate sullo streaming dei dati in tempo reale, come ad esempio per il Voice over IP (VoIP) usato per lo scambio interattivo dei dati vocali.
- **Total Length** indica la dimensione (in byte) dell'intero datagramma, comprendendo header e dati; tale lunghezza può variare da un minimo di 20 byte (header minimo e campo dati vuoto) ad un massimo di 65535 byte. In ogni momento, ad ogni host è richiesto di essere in grado di gestire datagrammi aventi una dimensione minima di 576 byte mentre sono autorizzati, se necessario, a frammentare datagrammi di dimensione maggiore.

- **Identification** è usato per identificare in modo univoco i vari frammenti di un datagramma IP originale.
- **Flags** sono bit utilizzati per il controllo del protocollo e della frammentazione dei datagrammi.
- **Fragment Offset** indica l'offset (misurato in blocchi di 8 byte) di un particolare frammento relativamente all'inizio del datagramma IP originale: il primo frammento ha offset 0. L'offset massimo risulta pertanto pari a 65528 byte che, includendo l'header, potrebbe eccedere la dimensione massima di 65535 byte di un datagramma IP.
- **Time To Live** indica il numero massimo di hop che il datagramma può fare ed è necessario per evitare la persistenza indefinita del pacchetto sulla rete nel caso in cui non si riesca a recapitarlo al destinatario. Quando questo parametro assume valore zero il datagramma non viene più inoltrato ma scartato. Tipicamente, quando un datagramma viene scartato per esaurimento del TTL, viene automaticamente inviato un messaggio ICMP al mittente del datagramma, specificando il codice di richiesta scaduta; la ricezione di questo messaggio ICMP è alla base del meccanismo del traceroute.
- **Protocol** indica il tipo di protocollo di livello trasporto nella porzione dati del pacchetto IP.
- **Header Checksum** è un campo usato per il controllo degli errori dell'header. Ad ogni hop, il checksum viene ricalcolato e confrontato con il valore di questo campo: se non corrisponde il pacchetto viene scartato.
- **Source address** indica l'indirizzo IP associato al mittente del datagramma.
- **Destination address** indica l'indirizzo IP associato al destinatario del datagramma.

- **Options + padding** sono opzioni (facoltative e non molto usate) per usi più specifici del protocollo. Devono essere necessariamente di dimensione multipla di 32 bit; in caso contrario sono aggiunti dei bit senza significato detti padding.

Frammentazione Per rendere il protocollo tollerante alle differenze delle varie sottoreti è stato inserito il meccanismo della frammentazione grazie al quale ogni device ha la possibilità di spezzare i dati in più pacchetti. Questo risulta essere necessario nel caso in cui la MTU (maximum transmission unit) della rete sia minore della dimensione del pacchetto. Ad esempio la dimensione massima di un pacchetto IP è di 65.535 byte ma la dimensione tipica del MTU di una rete Ethernet è di 1500. Questo comporta che per inviare tutto il payload del pacchetto IP sono necessati 45 frame ethernet. Quando un device riceve un pacchetto IP esamina la destinazione e sceglie su quale interfaccia instradarlo: ogni interfaccia ha infatti una MTU specifica e si controlla quindi se sia neccessaria una frammentazione. In questo caso i dati vengono suddivisi in diversi pacchetti, ognuno dei quali ha un proprio header con impostate la nuova dimensione, l'offset dei dati dall'originale e il bit MF (more fragment) in caso seguano altri frammenti. Una volta che tutti i frammenti sono stati ricevuti dalla destinazione, l'host può riassemblarli e passarli ai livelli superiori.

Routing Si definisce routing il meccanismo attraverso il quale un nodo determina quale interfaccia utilizzare per inviare un pacchetto che deve essere racapitato alla destinazione desiderata. Generalmente questa decisione viene presa consultando una tabella che, per ogni destinazione, indica quale interfaccia utilizzare. Questa tabella può essere costruita manualmente per piccole reti, mentre reti più complicate presentano topologie complesse e collegamenti ridondanti che possono cambiare rapidamente (ad esempio a causa di guasti); in questo caso la configurazione manuale non può essere quindi un'opzione attuabile. Gli algoritmi di routing, come RIP o il più usato Open Shortest Path First (OSPF), utilizzano le informazioni scambiate tra i nodi

per generare automaticamente queste tabelle e si possono classificare in due principali famiglie: Distance-vector e Link-state. Nell'algoritmo Distance-vector, ogni router misura la distanza che lo separa dai nodi adiacenti e riceve informazioni da questi ultimi con le quali aggiorna la propria tabella; ad ogni nuovo aggiornamento, la tabella viene ricalcolata ed inviata ad i nodi adiacenti. Nel link-state, invece, ogni nodo propaga le informazioni locali di sua conoscenza facendo in modo che alla fine dello scambio ogni router abbia raccolto dati sull'intera topologia della rete; viene successivamente applicato l'algoritmo di Dijkstra per determinare il cammino minimo verso ogni nodo e quindi il prossimo hop per ogni destinazione; questo metodo è usato dall'algoritmo OSPF. Per l'interconnessione di reti non è possibile usare queste famiglie di algoritmi poiché genererebbero troppo traffico e tabelle di routing enormi; in tali casi si ricorre invece ad altri metodi come il BGP (Boarder Gateway Protocol).

3.2.2 IPv6

Internet Protocol versione 6 (IPv6) è la versione del protocollo IP designata per succedere alla versione 4. IPv6 è stato sviluppato dalla Internet Engineering Task Force (IETF) ed è descritto dall'RFC 2460 [10] pubblicato nel dicembre 1998. IPv6 risolve il problema della saturazione di IPv4 aumentando lo spazio di indirizzamento a 128 bit, pari a circa $3,4 \times 10^{38}$ possibili indirizzi, giudicato abbastanza grande per qualsiasi espansione futura della rete globale. Oltre al nuovo spazio di indirizzamento, al protocollo sono state aggiunte nuove caratteristiche come il supporto nativo alla sicurezza (IPsec) e meccanismi di autoconfigurazione. IPv6 non è direttamente compatibile con IPv4 e ciò crea effettivamente una rete indipendente da quella esistente; ciò nonostante sono stati sviluppati diversi meccanismi per agevolare la transizione tra i due protocolli.

Indirizzamento La caratteristica più importante di IPv6 è il più ampio spazio di indirizzamento, portato da 32 a 128 bit. Ciò è stato fatto per

semplificare l'allocazione di indirizzi, permettere una più efficente aggregazione delle rotte ed implementare speciali metodi di indirizzamento come il multicast e l'anycast.

Gli indirizzi sono scritti in forma di otto gruppi di quattro cifre esadecimali separati da due punti e sono divisi in due parti: i primi 64 bit rappresentano il prefisso di rete mentre i restanti 64 bit l'identificativo di rete. Indirizzi unicast identificano l'interfaccia di un singolo host, indirizzi anycast un gruppo di interfacce mentre gli indirizzi multicast sono usati per inviare dati ad un gruppo di nodi differenti. Il broadcast così come è conosciuto in IPv4 è implementato attraverso il multicast all'indirizzo ff02::1. Gli indirizzi inoltre hanno uno scope che specifica in quale parte della rete sono validi (ad esempio solo nel collegamento diretto con un'altra interfaccia (link-local), nella propria sottorete (site-local, deprecato), o globale).

Formato del pacchetto IPv6 specifica un nuovo formato per i pacchetti ideato per minimizzare il lavoro che i ruoter devono compiere. Tale formato è simile a quello della versione 4 ma è stato semplificato eliminando molti campi poco usati e spostandoli in estensioni degli header separate. L'header ha dimensione fissa di quaranta byte, solo circa il doppio più grande rispetto alla precedente versione, a fronte di un aumento di quattro volte dello spazio di indirizzamento.

Le opzioni sono implementate come estensioni addizionali dell'header e sono inviate subito dopo l'intestazione fissa: queste hanno un campo, chiamato next header, che indica il tipo di header successivo, se presente, o il protocollo di quarto livello trasportato nel payload (quindi più di un estensione può essere presente per ogni datagram). Questo meccanismo permette flessibilità, supporto per futuri servizi per la sicurezza e la mobilità senza la necessità di riprogettare il protocollo e una maggiore efficienza dei router nel processare i pacchetti classici: nella maggior parte dei casi, infatti, i router intermedi non devono effettuare il parsing di queste estensioni.

Di seguito è riportata una lista delle estensioni più comuni:

- **Hop-by-hop options** Definisce un insieme arbitrario di opzioni per ogni hop attraversato.
- **Routing packet** Definisce un metodo per permettere al mittente di specificare la rotta da segure per un datagramma.
- **Fragment packet** È usato nel caso di frammentazione di un datagramma.
- **No next header** Indica che i dati nel payload del datagramma non sono incapsulati in nessun altro protocollo.
- **Destination options** Definisce un insieme arbitrario di opzioni per il destinatario.
- **Mobility options** Usato per Mobile IPv6.
- **Altri protocolli (TCP, UDP, ...)** Indica il protocollo di livello 4 del pacchetto.

L'efficienza è ulteriormente aumentata grazie alla rimozione del checksum (si assume che l'integrità dei datagrammi sia garantita dai livelli superiori come il TCP e l'UDP e da quelli inferiori come il data-link) e all'eliminazione della frammentazione nei router intermedi imponendo al mittente di inviare pacchetti piccoli quanto la minore MTU della rotta (per reperire questa informazione viene usato il Path MTU Discovery); quando la MTU di un link è più piccola del datagram, un messaggio ICMPv6 è inviato dal nodo che riscontra il problema verso il nodo mittente, che quindi è avvisato di ridurre la quantità di dati inviata per pacchetto.

Figura 3.3: Formato del pacchetto IPv6.

Autoconfigurazione statica Gli host possono autoconfigurarsi automaticamente, quando connessi a una rete IPv6, usando messaggi ICMPv6 di router discovery. Dopo aver effettuato la connessione, un host spedisce una richiesta multicast per avere i suoi parametri di configurazione; i router rispondono alla richiesta con un messaggio di router advertisement che contiene i parametri necessari. Questo meccanismo può non essere indicato per tutti i tipi di applicazioni e se necessario una rete può comunque utilizzare il DHCPv6 o essere configurata manuamente.

Supporto obbligato alla sicurezza IPsec (Internet Protocol Security) è stato originariamente sviluppato per IPv6 ma ha trovato una larga diffusione anche per IPv4, nel quale è stato integrato successivamente. IPsec è parte integrante dello standard IPv6 in cui il suo supporto è obbligatorio.

Mobilità Mobile IPv6 evita il problema del routing triangolare ed i router possono supportare la mobilità di rete permettendo ad intere sottoreti di essere spostate senza cambiare prefisso.

Meccanismi di transizione da IPv4 a IPv6 IPv4 e IPv6 sono effettivamente due diversi protocolli indipendenti e non compatibili. Nonostante il passaggio a IPv6 sia necessario, non è possibile richiedere a tutti i produttori ed utilizzatori di fissare una data precisa per passare da un sistema all'altro. Diversi protocolli sono stati sviluppati per agevolare il graduale passaggio e qui di seguito sono riportati i principali meccanismi:

Dual Stack L'implementazione del doppio stack IP in un sistema operativo permette di interpretare entrambe le versioni del protocollo, necessità fondamentale nelle tecnologie di transizione da IPv4 a IPv6. Questo meccanismo può essere implementato sviluppando due stack completamente indipendenti oppure utilizzando un unico modulo software che sia in grado di trattare le due forme. La seconda soluzione è più comunemente scelta nei moderni sistemi operativi ed è descritta nel RFC 4213 [15].

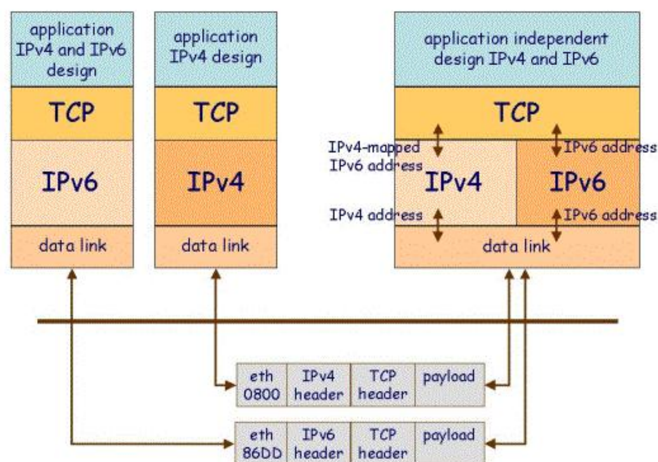

Figura 3.4: Applicazione indipendente dal protocollo su un host con doppio stack.

Il supporto dual stack permette agli sviluppatori di applicazioni di lavorare trasparentemente con IPv4 o IPv6 sullo stesso socket: quando si vogliono usare indirizzi IPv4, essi vengono rappresentati in un indirizzo IPv6 con i primi ottanta bit a zero, i successivi sedici ad uno e i restanti trentadue con l'effettivo indirizzo IPv4. Tale mappatura, inoltre, può essere usata per permettere ad host IPv6 che non hanno un indirizzo IPv4 assegnato di comunicare con altri nodi IPv4.

Tunneling Per raggiungere la rete Internet IPv6, un host che ha accesso unicamente ad una rete IPv4 può usare la tecnica del tunnelling che consiste nell'incapsulare pacchetti IPv6 in pacchetti IPv4. Questa procedura è comunemente indicata come protocollo 41. IPv6 può essere inoltre incapsulato in pacchetti UDP per superare una rotta con una rete nattata o con un firewall che blocca il traffico del protocollo 41. Allo stesso modo se da una rete IPv6 si vuole raggiungere un nodo che supporta soltanto IPv4 è possibile utilizzare altri tipi di tunnel basati sullo stesso principio.

Proxying e traduzione per host IPv6-only Il NAT che traduce da IPv4

Figura 3.5: Tunnelling: il pacchetto IPv6 viene incapsulato in un pacchetto IPv4. In questo modo è possibile attraversare la rete IPv4.

a IPv6 (e viceversa) è possibile con i protocolli NAT-PT e NAPT-PT definiti nel RFC 2766 [21]. Questa tecnica è stata molto discussa e considerata da taluni controversa fino al RFC 4966 [6] che la depreca definitivamente.

3.3 Livello Trasporto

Il livello trasporto permette il dialogo strutturato tra mittente e destinatario. I protocolli definiti permettono la comunicazione end-to-end, senza considerare i passaggi intermedi, tra processi residenti in host diversi.

Il TCP (Transmission Control Protocol) è connection-oriented e permette al flusso di dati di essere recapitato senza errori. I frammenti dello stream vengono impacchettati e passati ai layer inferiori delegando al ricevente il compito di riassemblarli e in caso di errore a chiederne il rinvio. Attraverso un meccanismo di flow control questo protocollo non permette ad una sorgente veloce di congestionare un ricevente lento, mentre con il congestion control si evita la congestione dei router posti tra i due end-point.

Al contrario l'UDP (User Datagram Protocol) è senza connessione e non assicura che i pacchetti arrivino correttamente a destinazione. Tali caratteristiche lo rendono maggiormente indicato per quelle applicazioni che necessitano di interattività e velocità (ad esempio audio e video vengono quasi sempre inviati attraverso UDP poichè in questi casi è molto più importante la regolarità e la continuità del flusso di dati).

Quest'ultimo protocollo ricopre un ruolo importante nell'architettura descritta all'interno del capitolo 4.

3.3.1 Protocollo UDP

L'User Datagram Protocol è uno dei protocolli fondamentali di Internet. Le applicazioni possono creare e inviare messaggi brevi (detti datagram) attraverso degli appositi socket. Il protocollo è stato definito nel 1980 da David P. Reed (RFC 768 [17]) e non garantisce l'affdabilità della connessione o che i pacchetti vengano ricevuti nell'ordine di invio; questi ultimi possono infatti essere duplicati o non arrivare affatto senza che questo venga segnalato in alcun modo. L'eliminazione di questo controllo rende il protocollo veloce ed efficiente per le applicazioni che non necessitano di queste garanzie. Applicazioni che utilizzano UDP sono ad esempio il Domain Name System (DNS), streaming media applications come IPTV e Voice over IP (VoIP).

UDP header Il protocollo UDP definisce l header dei pacchetti secondo la struttura riportata nella Figura 3.6.

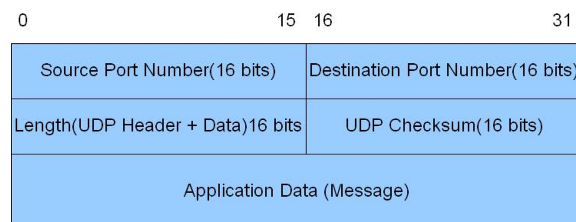

Figura 3.6: Formato del pacchetto UDP.

Caratteristiche del protocollo Come precedentemente accennato UDP è un semplice protocollo basato sul trasferimento di messaggi e privo di connessione (non si crea una connessione end-to-end dedicata). Lo stream di pacchetti viene inviato alla destinazione senza nessun controllo e tutti i pacchetti sono completamente indipendenti.

Caratteristiche principali sono dunque:

- **Inaffidabilità:** Dopo l'invio di un messagggio non è possibile in alcun modo avere informazioni sul suo corretto recapito.

- **Mancanza di ordinamento:** Il flusso di messaggi inviati può essere ricevuto in ordine differente dall'ordine di invio.
- **Leggerezza:** Data la mancanza di controlli è leggero in termini di dimensione dell'header, efficiente e veloce.
- **Datagram:** I pacchetti vengono inviati individualmente e nel caso arrivino a destinazione sono sicuramente integri. Hanno dimensione predefinita e non vengono frammentati o riassemblati a livello trasporto.

3.4 Livello Applicazioni

Il livello applicazioni si riferisce alla parte più alta della pila ISO/OSI e contiene tutti i protocolli che si occupano di fornire servizi per i processi delle applicazioni usate dagli utenti finali. Un programma applicativo interagisce con uno dei protocolli del livello di trasporto per ricevere dati o inviarli passandoli nella forma richiesta. Il modello OSI suddivide il Livello Applicazioni in tre ulteriori livelli:

- **Livello Sessione** offre servizi che consentono ad utenti che operano su macchine differenti di colloquiare tra loro attraverso la rete di comunicazione;
- **Livello Presentazione** ha il compito di trasformare i dati forniti dal Livello applicazioni in un formato standard e offrire servizi di comunicazione comuni, quali la crittografia, la compressione del testo e la riformattazione;
- **Livello Applicazioni** il livello ultimo dedicato al vero e proprio scambio di dati per le applicazioni.

Nell'uso comune, spesso questi livelli sono unificati.

A questo livello appartengono numerosi protocolli tra i quali: FTP (File Transfer Protocol) per il trasferimento di dati, HTTP (HyperText Transfer

Protocol) su cui è basato il World Wide Web, IMAP per i servizi di email e SSH per il controllo di terminali remoti; in questa sezione verranno descritte le tecnologie VoIP, apparteneti anch'esse al Livello Applicazioni, ed analizzati i protocolli che intervengono all'interno di una comunicazione di questo tipo.

3.4.1 VoIP

Voice over Internet Protocol (Voice over IP, VoIP) è una famiglia di tecnologie internet e protocolli di comunicazione progettata per distribuire comunicazioni vocali e sessioni multimediali attraverso il protocollo IP in sostituzione della rete tradizionale PSTN (public switched telephone network). Il vantaggio principale di questa tecnologia sta nell'eliminazione dell'obbligo di riservare una quota di banda fissa per ogni telefonata (come avviene nelle reti a commutazione di circuito), sfruttando l'allocazione dinamica delle risorse, caratteristica del protocollo IP.

Protocolli Per effettuare una chiamata VoIP, generalmente, si digitalizza il segnale fisico della voce codificando il segnale analogico in digitale (eventualmente comprimendolo) e si pacchettizzano i dati che vengono poi trasmessi attraverso il protocollo IP; similmente, in fase di ricezione vengono effettuate le corrispettive operazioni inverse.

Il VoIP richiede l'utilizzo di due tipologie di protocolli di comunicazione in parallelo: una per il trasporto dei dati (pacchetti voce su IP), ed una per la segnalazione della conversazione, come la ricostruzione del frame audio, la sincronizzazione e l'identificazione del chiamante. Per il trasporto dei dati, nella grande maggioranza delle implementazioni VoIP viene adottato il protocollo *RTP* (Real-time Transport Protocol, a sua volta basato su UDP) mentre sono stati sviluppati diversi protocolli di segnalazione: H. 323, della ITU (International Telecommunications Union), è stato uno tra i primi protocolli VoIP; nasce in ambito telefonico e delinea un'architettura completa per lo svolgimento di conferenze multimediali, comprendente la definizione dei formati di codifica a livello applicativo e la gestione degli aspetti di si-

curezza. *SIP*, della IETF (Internet Engineering Task Force), è più recente e sta riscontrando un maggiore successo di H. 323; ha funzionalità di instaurazione e terminazione della sessione, operazioni di segnalazione, tono di chiamata, chiamata in attesa, trasferimento, identificazione del chiamante ed altro ancora. Altri protocolli utilizzati per la codifica della segnalazione della conversazione sono: Skinny Client Control Protocol, protocollo proprietario della Cisco, Megaco (conosciuto anche come H.248), MGCP, MiNET, protocollo proprietario della Mitel, Inter Asterisk Xchange, (soppiantato da IAX2) usato dai server Asterisk open source PBX e dai relativi software client, e XMPP, usato da Google Talk, inizialmente pensato per l'IM ed ora esteso a funzioni Voip grazie al modulo Jingle. È da notare che uno dei protocolli più usati per le utenze domestiche è quello di Skype, proprietario e non rilasciato pubblicamente, che si basa sui principi del Peer-to-Peer.

Benefici

I benefici dell'utilizzo di una tecnologia innovativa quale il VoIP sono molteplici:

Diminuzione dei costi Il VoIP consente un notevole abbattimento dei costi pe il singolo utente. Sia per chiamate nazionali che internazionali o intercontinentali, il costo della chiamata resta invariato; utilizzando una linea ADSL si può chiamare qualsiasi altro PC gratuitamente; per contattare telefoni PSTN, invece, le compagnie fornitrici di VoIP devono pagare unicamente l'accesso all'ultimo tratto della comunicazione che appartiene all'azienda telefonica del ricevente.

Agevolazioni in campo lavorativo Il VoIP può essere usato sia all'interno di una stessa azienda che per telefonate più complesse, come teleconferenze in multi-point. Inoltre si può utilizzare una sola rete integrata per voce e dati, che garantisce una riduzione dei bisogni totali di infrastruttura e, in ambito aziendale, uno scambio agevolato delle informazioni.

Flessibilità Il VoIP può fornire servizi che possono essere molto difficili da implementare usando PSTN, come ad esempio la possibilità di trasmettere più di una telefonata contemporaneamente con una sola linea, l'indipendenza del numero dalla locazione fisica, l'integrazione con altri servizi Internet (ad esempio conversazioni video), lo scambio di messaggi e dati, le conferenze audio e la gestione di rubriche telefoniche.

Sicurezza Rispetto al PSTN, il VoIP consente con relativa facilità di eseguire conversazioni non intercettabili grazie all'uso della crittografia e protocolli standardizzati come il Secure Real-time Transport Protocol..

Numeri di emergenza Dal 2006 la maggior parte dei provider di servizi VoIP ha abilitato le chiamate ai numeri di emergenza, localizzando in tempo reale l'utente.

Problemi

Anche il VoIP, come tutte le tecnologie in via di sviluppo, presenta alcuni aspetti da perfezionare:

Qualità del servizio Le comunicazioni basate su IP sono meno affidabili rispetto alla commutazione di circuito del PSTN perché per definizione IP non fornisce un meccanismo di prenotazione del canale, e non assicurando la corretta ricezione dei pacchetti, non offre garanzie QoS (Quality of Service). Le conversazioni VoIP possono presentare due problemi fondamentali:

- **Latenza:** normalmente la politica di gestione dei router IP è FIFO, cioè il primo pacchetto ad entrare è anche il primo ad essere processato; router posti su una linea ad alto volume di traffico potrebbero quindi introdurre una latenza che eccede quella massima per avere una conversazione di qualità.

- **Jitter:** con questo termine si intende la variazione della velocità con la quale i pacchetti voce arrivano; se questo valore varia eccessivamente la qualità della conversazione degrada. Questo problema è risolto introducendo un buffer per i pacchetti in entrata in modo da mitigare l'arrivo fuori ordine e regolare la velocità con la quale i dati in ingresso sono trasformati in audio al costo di un incremento della latenza.

Necessità della rete elettrica Un dispositivo telefonico classico è direttamente alimentato dalla rete telefonica ed in caso di guasti alla rete elettrica il telefono è comunque funzionante. A causa dei differenti tipi di dispositivi, un telefono VoIP non può contare su questa peculiarità; alcuni produttori hanno iniziato tuttavia ad aggiungere delle batterie ai propri device in modo da garantire il servizio per un certo periodo anche in assenza di alimentazione diretta.

Numerazioni speciali Un ulteriore svantaggio è dato dalla impossibilità di alcuni operatori di chiamare i numeri di emergenza ed alcuni numeri speciali.

Capitolo 4

Architettura

In questo capitolo verranno descritti i principali problemi relativi all'utilizzo delle tecnologie VoIP su reti Wi-Fi. Dopo l'introduzione degli standard fino ad ora sviluppati per il supporto all'interattività nelle reti mobili, si passerà alla defnzione di un'architettura che intende superare alcune di queste limitazioni.

4.1 VoWIFI

Il VoIP è una tecnologia con requisiti molto specifici che, se non rispettati, impediscono una corretta fruizione del servizio. Le caratteristiche tecniche di un canale Wi-Fi dovrebbero permettere di soddisfare tali requisiti ma in realtà questo non sempre è possibile.

L'attuale standard 802.11b/g è stato realizzato senza porre la dovuta attenzione alle necessità delle applicazioni real-time: il protocollo MAC definito dallo standard permette di coordinare l'accesso al canale da parte dei vari trasmettitori ma non discrimina le tipologie di traffico generate e di conseguenza non garantisce che un'applicazione real-time possa spedire i propri dati entro determinati limiti di tempo.

L'accesso al canale avviene mediante contesa dello stesso da parte dei vari trasmettitori e non sono previsti limiti temporali entro i quali liberare

il canale stesso; un'applicazione real-time, in genere più sensibile ai ritardi di trasmissione, ha quindi le stesse probabilità di accesso delle altre. Un ulteriore fattore limitante è dato dalla procedura di *roaming* eseguita dal terminale quando questo si sposta eccessivamente dall' area di copertura/ricezione dell'access point a cui è connesso. Quando il collegamento con l'AP è perso, il terminale avvia tale procedura per ripristinare il collegamento con un AP diverso. Il roaming viene eseguito solo per connettersi ad AP facenti parte dello stesso ESS (Extended Service Set)[1] e richiede svariati secondi per il suo completamento nel caso in cui sia necessario riconfigurare anche il livello network. Inoltre un singolo AP può gestire non più di dieci sessioni VoIP simultanee, pena un degrado netto della qualità di conversazione. Infine i pacchetti VoIP, in cui le dimensioni del payload sono molto ridotte, subiscono un forte overhead dovuto ai vari header RTP/UDP/IP/MAC.

Le limitazioni finora descritte sono state oggetto di studio ed hanno portato a due estensioni dello standard: 802.11e per fornire il QoS e 802.11r per velocizzare i tempi di roaming.

4.1.1 802.11e

IEEE 802.11e [2] definisce un insieme di miglioramenti riguardanti il Quality of Service per le reti wireless lan. Questo standard è considerato di cruciale importanza per le applicazioni sensibili alla latenza come quelle multimediali e il VoIP.

Nello standard originale è definito il concetto di coordinazione (PCF, point coordination function) così implementato: gli access point spediscono frame di tipo "beacon" ad intervalli regolari (di solito 0,1 secondi) in base ai quali sono definiti due periodi: un periodo con possibilità di contesa (CP, Contention Period) ed uno senza (CFP, Contention Free Period), dove è possibile assegnare turni fissi ai nodi che vogliono inviare traffico sensibile al ritardo (Distributed Coordination Function (DCF)). Lo standard 802.11e

[1] Insieme costituito da due o più BSS collegati tra loro al fine di generare un'area di copertura maggiore

estende tale meccanismo introducendo il concetto di classi/tipologie di traffico. Questa nuova funzione di coordinazione è chiamata *Hybrid Coordination Function (HCF)* e presenta due metodi di accesso al canale:

- **Enhanced Distributed Channel Access (EDCA):** il traffico ad alta priorità ha maggiori possibilità di essere spedito; viene usata una finestra di contesa ridotta e viene fornito al nodo che la utilizza un periodo limitato di acesso al canale senza contesa (TXOP, Transmit Opportunity), durante il quale un nodo mobile può inviare tanti frame quanti disponibili (se un frame è troppo grande per essere spedito in questo periodo deve essere frammentato in modo da rientrare nell'intervallo).

- **HCF Controlled Channel Access (HCCA):** mantiene l'impostazione di base della PCF aggiungendo ad essa i concetti di classi di traffico e TXOP. È considerato il più avanzato e complesso meccanismo di coordianzione ma è definito come opzionale dallo standard e scarsamente diffuso e supportato dai dispositivi hardware attualmente in commercio.

Con lo standard 802.11e, inoltre, sono introdotte funzionalità di:

- **Admission Control** Permette all'AP di impedire l'accesso al canale ad un terminale con una determinata tipologia di traffico.

- **QosNoAck** Questo parametro, specificabile per ogni frame che un terminale desidera spedire all'AP, indica di non generare un ACK di risposta in seguito alla corretta ricezione del frame per evitare tentativi potenzialmente inutili di ritrasmissione di dati real-time.

- **DLS (Direct Link Setup)** Permette la comunicazione diretta tra terminali all'interno dello stesso BSS[2] .

[2]singolo AP con tutte le stazioni ad esso associate

- **Block acknowledgments** Permette di ricevere un solo ACK per tutti i frame in un TXOP.
- **Automatic power save delivery** Permette un risparmio di energia consentendo all'AP di inviare dati al MN solo subito dopo aver ricevuto un frame da quest'ultimo; il MN passa poi allo stato di sleep in cui resta fino all'invio del prossimo frame.

4.1.2 802.11r

Lo standard 802.11r [4] ha come obiettivo quello di ridurre il tempo necessario a completare la procedura di roaming del nodo mobile tra 2 access point appartenenti allo stesso Extended Service Set.

Il roaming di un nodo all'interno di un ESS è supportato sin dalla prima versione dello standard 802.11; quando un terminale si allontana dal range di un access point ed entra nel range di un altro, è possibile disconnettersi dall'AP originale per associarsi a quello nuovo tramite una procedura chiamata *handoff*.

In un primo momento l'handoff nelle reti Wi-Fi era molto semplice e consisteva nello scambio di soli quattro messaggi; con l'introduzione degli standard per la sicurezza ed altre estensioni i tempi di associazione si sono dilatati fino ad arrivare ad impiegare da due a dieci secondi.

Lo standard 802.11r specifica di eseguire la procedura di autenticazione solo la prima volta che un terminale si connette ad un AP dell' ESS; l'autenticazione verrà mantenuta nella rete wireless mediante l'uso di una chiave che sarà richiesta in fase di roaming, senza necessità di ulteriori negoziazioni per l'accesso.

4.1.3 Considerazioni

Le soluzioni sopra descritte rappresentano un supporto solo parziale alla costruzione di una solida infrastruttura per l'utilizzo di applicazioni multimediali e VoIP su reti wireless.

Alla luce delle considerazioni esposte le funzionalità di QoS offerte da 802.11e possono sembrare sufficienti ad una gestione opportuna del traffico real-time, ma la gestione della mobilità dovrebbe essere ulteriormente potenziata.

Sebbene lo standard 802.11r abbia ridotto significativamente la durata della procedura di roaming va evidenziato che tale soluzione non considera il ritardo introdotto da un eventuale riconfigurazione del livello rete con conseguente modifica dell'indirizzo IP; se l'AP verso cui avviene il roaming è attestato su di una sub-net diversa da quella di provenienza allora sarà necessario eseguire una riconfigurazione di livello rete ed applicazione. L'uso di MobileIP contribuisce a mitigare senza però risolvere completamente il problema dei ritardi. Inoltre la soluzione proposta dallo standard 802.11r è valida solo per AP appartenenti allo stesso ESS.

Risulta quindi necessario un monitoraggio costante del collegamento Wi-Fi, utile a rilevare tempestivamente questo tipo di problemi in modo da poter eseguire azioni correttive o di recupero prima che causino un degrado nel QoS percepito dall'applicazione VoIP.

4.2 Always Best Packet Switching

Always Best Packet Switching (ABPS) [11], sviluppata dal Dipartimento di Informatica dell'Università di Bologna, permette alle applicazioni di usare simultaneamente tutte le interfacce di rete disponibili e di inviare pacchetti UDP attraverso l'interfaccia reputata al momento più adeguata. ABPS garantisce il rispetto dei requisiti di QoS dettati da ITU per un servizio VoIP assumendo che:

- il terminale sia equipaggiato con due o più interfacce Wi-Fi
- la probabilità di perdita di un pacchetto sul collegamento Wi-Fi non sia superiore al 10% dei pacchetti trasmessi

- il numero di pacchetti non persi sul tratto wireless ma ricevuti dal destinatario dopo 150 millisecondi non sia superiore all'1%

in particolare, è possibile ottenere un one-way delay medio inferiore a 150 millisecondi ed un packet loss rate inferiore al 3%, a fonte del 10% sperimentato senza l'uso di tale sistema.

Questa architettura si basa su un approccio cross-layer utile a far risalire, dal livello data-link al livello applicazione, opportune informazioni sullo stato del collegamento Wi-Fi; queste ultime saranno gestite da un'attività di monitoraggio eseguita a livello applicazione, il cui scopo è quello di rilevare eventuali violazioni dei requisiti di QoS del servizio VoIP. In tal caso il sistema, sfruttando la presenza multipla di interfacce Wi-Fi, si occupa di ritrasmettere eventuali pacchetti persi. Il sistema inoltre gestisce e configura opportunamente le interfacce Wi-Fi in modo che siano sempre pronte all'uso in base alle informazioni derivanti dalla fase di monitoraggio.

4.3 Architettura

Si immagini l'esistenza di una comunicazione VoIP tra due terminali, denominati A e B nella Figura 4.1. Il terminale A è di tipo mobile, equipaggiato con due o più interfacce Wi-Fi in standard 802.11b/g/n ed è localizzato in una tipica area metropolitana, supportata da connettività Wi-Fi, in cui sono presenti anche altri terminali che utilizzano la medesima rete composta da molteplici AP connessi ad internet mediante infrastruttura wired. Si suppone inoltre, per semplicità, che il terminale B sia collegato direttamente ad internet mediante collegamento wired.

Tra i due end point è frapposta una terza componente, un server esterno alle reti wireless che fa da proxy per il nodo mobile; il suo compito è quello di mantenere la comunicazione con tutte le interfacce del MN comunicando con il corrispettivo proxy lato client. Il proxy server è un punto fisso della rete e in caso di riconfigurazione tramite roaming delle interfacce del MN, mantiene la connessione VoIP in corso e nasconde i cambiamenti di indirizzo

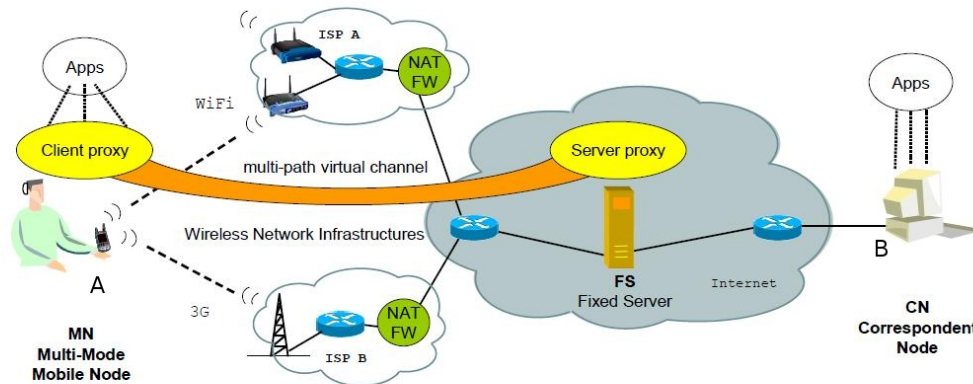

Figura 4.1: L'architettura descritta per la gestione della mobilità ed il mantenimento del QoS.

al nodo B (Corresponded Node, CN). Il nodo mobile, inoltre, potrebbe essere coperto da firewall e NAT ma grazie al proxy server tale limitazione viene superata.

È da notare che l'infrastruttura di rete esistente non deve essere alterata in alcun modo e che solo il nodo mobile avrà bisogno di software aggiuntivo per sfruttare il meccanismo ABPS; questa architettura è sufficentemente generale da poter essere applicata in un contesto completamente wireless dove i due end system della comunicazione VoIP sono entrambi nodi mobili connessi a differenti reti wireless, equipaggiati con più di un'interfaccia.

Un aspetto fondamentale del sistema è il monitoraggio del QoS e la gestione delle molteplici interfacce Wi-Fi presenti nel terminale. La fase di monitoraggio serve a rilevare tempestivamente, e possibilmente a prevenire, eventuali violazioni del QoS. Lo stato del collegamento Wi-Fi in uso dal servizio VoIP, rappresentato dal numero di pacchetti persi, viene dedotto mediante informazioni fatte risalire dal livello data-link con un approccio cross-layer. Poiché la grande maggioranza di perdite di pacchetti avviene nel link wireless, tale approccio risulta essere notevolmente più vantaggioso, ad esempio, di un ACK a livello applicazione che viene recapitato dopo un intero round trip tra i due nodi in comunicazione. Se il meccanismo di monitoraggio sospetta una perdita di dati, viene intrapresa un'azione correttiva:

il pacchetto può essere ritrasmesso attraverso un'altra interfaccia in un lasso di tempo sufficente a rispettare i requisiti di interattività precedentemente menzionati. Le interfacce Wi-Fi non utilizzate per la trasmissione vengono mantenute possibilmente attive e configurate, pronte quindi ad essere usate nel caso l'interfaccia in uso introduca violazioni di QoS.

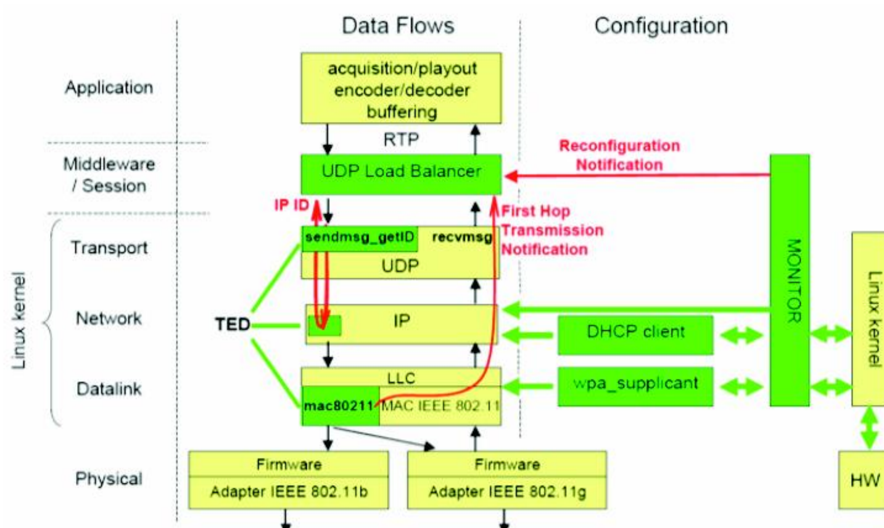

Figura 4.2: Componenti del meccanismo cross-layer.

L'architettura del sistema proposto, come mostrata nella Figura 4.2, prevede le seguenti componenti:

Transmission Error Detector (TED) Considerato il componente più importante del sistema, il TED ha il compito di monitorare la trasmissione dei dati provenienti dall'applicazione VoIP. In particolare si assicura che tali dati siano stati ricevuti correttamente dall'AP verificando per ogni pacchetto spedito la corretta ricezione del relativo ACK proveniente dall'AP stesso. Ha inoltre il compito di notificare all'UDP Load Balancer (descritto in seguito), tramite la "First-hop Notification", eventuali errori nella spedizione di uno specifico pacchetto.

Monitor Eseguito come applicazione separata, configura dinamicamente le interfacce Wi-Fi e le relative regole di routing; comunica con il kernel Linux e con l'ULB, informando quest'ultimo quando un' interfaccia è stata correttamente configurata oppure disabilitata in seguito ad un errore di trasmissione (Reconfiguration Notification, in Figura 4.3).

UDP Load Balancer (ULB) In base alle notifiche provenienti dal TED, l'ULB gestisce un'eventuale ritrasmissione dei pacchetti non ricevuti dall'AP. Stabilisce inoltre, basandosi sulle suddette notifiche, quale interfaccia Wi-Fi tra quelle disponibili debba essere usata per le future trasmissioni e/o ritrasmissioni di pacchetti. Utilizza un socket UDP per ogni interfaccia Wi-Fi attiva e configurata dal Monitor, effettuando una bind tra socket e interfaccia in modo da assicurare che l'effettiva trasmissione del pacchetto avvenga attraverso l'interfaccia scelta.

L'ULB può ricevere tre tipologie di notifica (indicate con linee non tratteggiate in Figura 4.3):

- **Reconfiguration Notification** Proveniente dal Monitor, notifica all'ULB la configurazione, l'attivazione o la disattivazione di una interfaccia Wi-Fi; in quest'ultimo caso viene chiuso il relativo socket ed ogni pacchetto spedito tramite l'interfaccia, per il quale non è stata ricevuta alcuna First-hop Transmission Notification dal TED, viene considerato perso.

- **ICMP Notification** Proviene dal protocollo ICMP e notifica l'eventuale perdita di pacchetti avvenuta nel tratto di comunicazione tra A e B.

- **First-hop Transmission Notification** Notifica proveniente dal TED che indica per ogni pacchetto spedito se è stato correttamente ricevuto dall'AP oppure definitivamente scartato dal livello MAC.

Queste tre tipologie di notifica permettono ad ULB di stabilire se un determinato pacchetto debba essere ritrasmesso (mediante un'altra interfaccia

Wi-Fi) o definitivamente scartato. Permettono inoltre di realizzare un opportuno algoritmo di selezione dinamica dell'interfaccia Wi-Fi che offre maggiori garanzie in trasmissione.

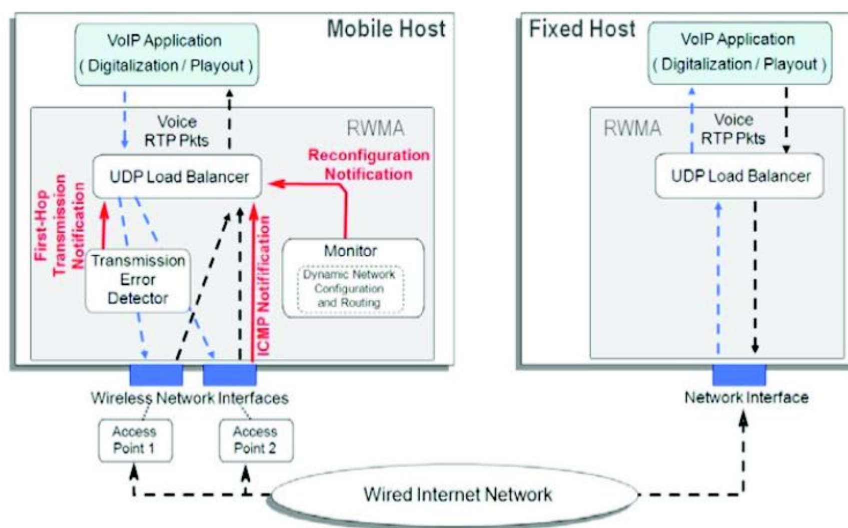

Figura 4.3: L'architettura descritta per la gestione della mobilità ed il mantenimento del QoS.

L'architettura appena descritta rappresenta il middleware presente sul terminale mobile; il server proxy presenta invece un'architettura molto più semplice, composta unicamente da una versione semplificata dell'ULB che controlla una sola interfaccia di rete, assumendo che i pacchetti trasmessi non vengano scartati dal first-hop wired. In sintesi il suo unico e principale compito è quello di tenere traccia degli indirizzi IP da cui provengono i pacchetti trasmessi dal nodo mobile, informazione essenziale per poter trasmettere correttamente i propri dati verso il MN.

4.3.1 Algoritmo di selezione delle interfacce Wi-Fi

Mediante le notifiche ricevute dalle altre componenti del sistema l'ULB realizza un algoritmo di gestione e selezione delle interfacce Wi-Fi del terminale.

Tale algoritmo assume che un pacchetto trasmesso sia stato perso se si riceve un'opportuna notifica dal TED oppure se non si riceve alcuna notifica entro 30 millisecondi. Il timeout è necessario perché il firmware di alcune interfacce Wi-Fi attualmente in commercio non informa il protocollo MAC se un determinato pacchetto è stato scartato, notificando tuttavia la corretta ricezione di un pacchetto da parte dall'AP.

I pacchetti vengono spediti da ULB, nello stesso ordine in cui sono stati ricevuti dall'applicazione VoIP, tramite l'interfaccia che può fornire le maggiori garanzie di trasmissione. Un'interfaccia configurata ed attiva si trova inizialmente nello stato *funzionante*; alla ricezione di un errore ICMP o di un Reconfiguration Notification l'interfaccia passa allo stato *disabilitata* mentre se entro il timeout non viene ricevuta alcuna segnalazione per un pacchetto, o viene ricevuta una First-hop Transmission Notification che ne segnala la perdita, l'interfaccia passa allo stato *sospetta*. Nel caso in cui si trovi in quest'ultimo stato e venga ricevuto un pacchetto dal destinatario B, o venga ricevuta una First-hop Transmission Notification riguardante la corretta trasmissione di un pacchetto, l'interfaccia torna ad assumere lo stato funzionante. Quando ULB deve trasmettere un pacchetto seleziona quindi un'interfaccia che sia nello stato funzionante o, se non ve ne sono, un'interfaccia nello stato sospetta; altrimenti il pacchetto viene scartato. Se la scelta dell'interfaccia ha avuto successo viene avviato il timeout; al suo scadere, in mancanza di notifiche di avvenuta trasmissione, il pacchetto viene rispedito selezionando un'interfaccia diversa (anche in questo caso se non sono disponibili interfacce nello stato funzionante o sospetta il pacchetto viene scartato).

Capitolo 5

Obiettivo

Il presente lavoro è incentrato sullo studio e l'implementazione di una struttura atta a migliorare il supporto alla mobilità per applicazioni multimediali e VoIP in dispositivi mobili, con riferimento allo scenario e all'architettura descritti nel precedente capitolo; in particolare la tesi è focalizzata sulla possibilità di esecuzione, in un dispositivo mobile equipaggiato con sistema operativo Android, di un'applicazione all'interno di una Macchina Virtuale, rendendo così possibile all'interno del dispositivo stesso una specifica gestione del flusso di dati prodotto dall'applicazione.

5.1 Scenario Nodo Mobile

Si supponga di disporre di un dispositivo mobile in cui sia presente un proxy client, implementato con architettura ABPS descritta in precedenza, per il miglioramento del QoS e la gestione dinamica delle interfacce disponibili; per permettere ad un'applicazione VoIP di usufruire delle funzionalità messe a disposizione dal proxy client risulta necessario che i datagrammi UDP prodotti dall'applicazione siano presi in consegna dal proxy stesso e non vengano al contrario inoltrati direttamente sulle interfacce di rete attraverso lo stack TCP/IP di sistema. Tale aspetto non è banale: per compiere questa operazione è necessario intervenire all'interno del codice sorgente del-

l'applicazione modificandone alcuni aspetti. Questo approccio non è però sempre possibile, ad esempio per quanto riguarda applicazioni proprietarie non opoen source per le quali non si ha accesso al codice, ed inoltre non rappresenta una soluzione abbastanza generica.

Una possibile alternativa consiste quindi nell'eseguire l'applicazione VoIP all'interno di una Macchina Virtuale, in modo tale da poter disporre di un'interfaccia di rete virtuale; quest'ultima verrà vista dall'applicazione come l'unica interfaccia di rete disponibile ed inoltrerà dunque i propri datagrammi UDP attraverso di essa; senza apportare alcuna modifica al codice sorgente dell'applicazione sarà quindi possibile redirigere il flusso dati in uscita dall'interfaccia virtuale verso il proxy client, che a quel punto gestirà i pacchetti secondo le proprie politiche interne.

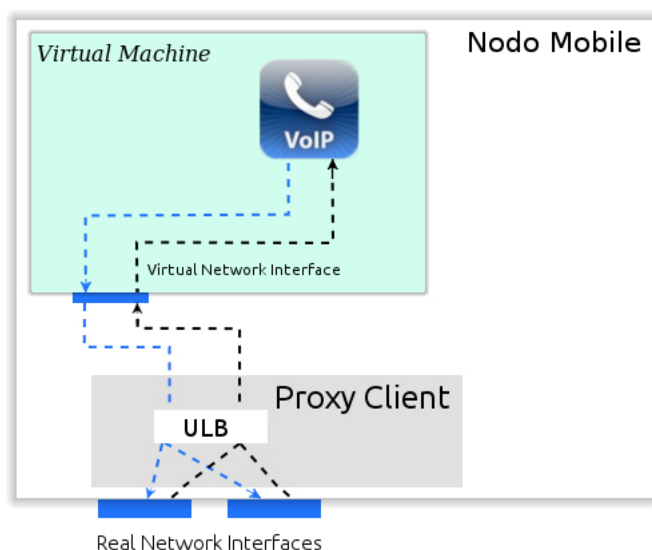

Figura 5.1: Comunicazione tra Applicazione VoIP e interfacce di rete.

Il sistema operativo mobile a cui si è fatto riferimento all'interno di questa tesi è Android, ai cui dettagli è dedicato il paragrafo seguente.

5.1.1 Android

Android è un sistema operativo per dispositivi mobili creato da Google e presentato nel Novembre 2007; è basato sul kernel Linux (nella versione 3.x da Android 4.0 in poi), con middleware, librerie e API realizzate in C o C++. Utilizza la Dalvik Virtual Machine con compilatore just-in-time per l'esecuzione di Dalvik Dex-code, solitamente tradotto da codice bytecode Java. Il kernel Linux di Android contiene delle modifiche all'architettura classica proposte da Google esternamente al ciclo di sviluppo ufficiale del kernel. Un tipico sistema Android non possiede infatti X Window System nativo, non supporta il set completo standard di librerie GNU e nel caso del C++ è presente solo una parziale implementazione delle Standard Template Library; queste caratteristiche rendono quindi complesso il porting di applicazioni Linux o librerie su Android. Per semplificare lo sviluppo di programmi scritti in linguaggio nativo C, è stato rilasciato l'Android Native Development Kit, un tool utilizzato nel presente lavoro e che sarà ampiamente descritto nel capitolo successivo; le applicazioni Android sono tuttavia Java-based e dunque programmi scritti in codice nativo C/C++ devono essere richiamati da codice Java.

Gli aspetti principali del kernel Linux ereditati dal sistema operativo Android riguardano la gestione della memoria, la gestione dei processi ed il network stack, il quale ricopre un ruolo di particolare importanza all'interno di questa tesi e sarà quindi analizzato nel paragrafo seguente.

5.1.2 Linux Network Stack

Lo stack di rete del kernel Linux, a cui ci si riferisce comunemente con il termine *Internet Protocol Suite* o stack TCP/IP, deriva dallo stack BSD ed è strutturato in differenti livelli di interfacce che costituiscono l'implementazione del modello ISO/OSI descritto nel capitolo 3, di cui tuttavia vengono specificati unicamente i livelli di rete e di trasporto [12]; come rappresentato in Figura 5.2, in cima allo stack è presente un livello Applicazione, che

definisce i protocolli coinvolti nelle comunicaizioni al livello dei processi e rappresenta gli utenti nello stack di rete; l'ultimo livello è invece costituito dai dispositivi fisici di trasmissione, come ad esempio schede di rete Ethernet o Wi-Fi. Fra i due livelli sopracitati è posto il network stack del kernel che è a sua volta composto dalle seguenti componenti:

- **System Call Interface:** livello che fornisce un meccanismo attraverso il quale un'applicazione eseguita in user-space ha accesso al sottosistema network del kernel; una possibile system call di rete effettuata dall'utente attraverso una delle funzioni delle socket API, viene presa in consegna da quest'interfaccia che provvede ad inoltrarla al livello socket sottostante. Ognuna delle funzioni delle API socket corrisponde infatti ad una chiamata del kernel (a cui viene anteposto il prefisso sys_) definita nel file `linux/net/socket.c`.

- **Socket Interface:** interfaccia che fornisce un insieme di funzioni per il supporto di una grande varietà di protocolli tra i quali, oltre ai tipici protocolli TCP e UDP, vi sono il protocollo IP, raw Ethernet e altri protocolli di trasporto come lo Stream Control Transmission Protocol (SCTP). La comunicazione attraverso lo stack di rete avviene mediante l'utilizzo dei socket, la cui struttura, chiamata struct sock e definita nel file `linux/include/net/sock.h`, specifica tutte le caratteristiche relative ad un particolare socket, incluso il protocollo utilizzato e le operazioni consentite. Compito principale del socket layer è dunque quello di fornire un'interfaccia di comunicazione tra il livello Applicazione e il livello di Trasporto, permettendo la corretta transizione dei dati; questi ultimi, all'interno del network stack, sono rappresentati da una struttura chiamata sk_buff in cui sono mantenute tutte le informazioni riguardanti il pacchetto, come payload e header, e che ne permette una migliore gestione delle modifiche applicate da ogni livello attraversato (come ad esempio l'aggiunta degli header).

- **Protocolli di rete:** sezione che definisce i protocolli di rete disponibili, come ad esempio TCP, UDP etc. Corrisponde ai livelli di Trasporto e di Rete del modello OSI e fornisce dunque le funzionalità descritte nel capitolo 3 da questi ultimi due, procedendo all'effettiva costruzione del pacchetto da spedire.

- **Interfaccia dei device:** fornisce un insieme di funzioni che possono essere utilizzate dai driver dei dispositivi di rete, che costituiscono il livello inferiore, per comunicare con lo stack dei protocolli superiore; per l'invio di un sk_buff dal protocol layer ad un dispositivo è utilizzata la funzione dev_queue_xmit, che accoda il sk_buff per permetterne la trasmissione tramite il device driver inferiore; ogni sk_buff contiene al suo interno, identificato dalla struttura net_device, il dispositivo di rete attraverso il quale il pacchetto sarà spedito (o attraverso il quale è stato ricevuto).

- **Device Drivers:** posto al termine dello stack, gestisce la comunicazione con le interfacce di rete fisiche.

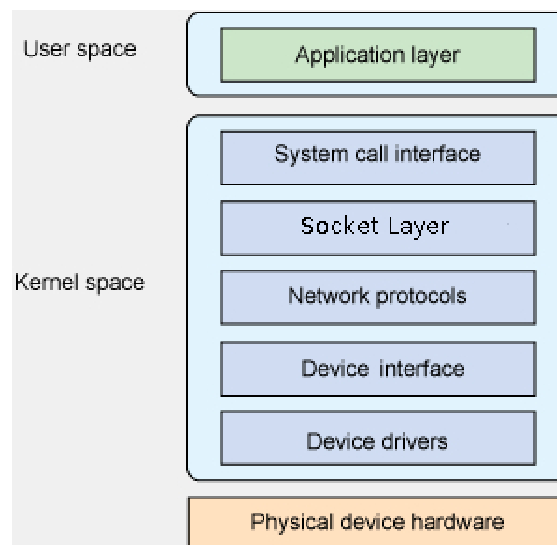

Figura 5.2: Struttura dell'Internet Protocol Suite

Trasmissione di pacchetti attraverso lo stack TCP/IP

Un'applicazione utilizza l'interfaccia fornita dalle socket API[1] per inviare dati attraverso la rete; numerosi sono gli strumenti messi a disposizione da quest'ultima come ad esempio la funzione socket() che permette la creazione di un nuovo socket del tipo specificato o le funzioni send() e recv() per la spedizione e la ricezione di dati[2].

Nel momento in cui l'applicazione invia i dati attraverso il socket utilizzando una delle funzioni sopracitate, il controllo raggiunge la system call sock_sendmsg appartenente al Socket Interface Layer che, ponendosi tra l'applicazione e il livello di trasporto, redirige la chiamata verso il corretto protocollo del livello inferiore; a seconda del protocollo desiderato, se TCP o UDP, viene chiamata rispettivamente la funzione tcp_sendmsg() o udp_sendmsg() che definisce dunque il passaggio al livello di trasporto.

Nel caso di applicazioni VoIP, come precedentemente anticipato, il protocollo di trasporto utilizzato è quello UDP, il quale dopo aver effettuato una serie di controlli sul dato ricevuto richiama la funzione ip_append_data; quest'ultima controlla che la coda del socket sia vuota e definisce l'header IP per ogni frammento in cui il dato sarà suddiviso (la frammentazione è effettuata in base al valore della MTU), costruendo una sequenza di sk_buf e inserendoli nella coda di output del socket. Attraverso la funzione udp_push_pending_frames viene costruito l'header UDP e passato il controllo al livello rete che, dopo aver inserito il relativo header, effetua il routing dei pacchetti in base alla loro destinazione: se quest'ultima corrisponde ad un indirizzo esterno il pacchetto viene inoltrato al livello data-link, mentre nel caso in cui debba essere recapitato all'interno dell'host stesso viene rediretto ai livelli superiori.

[1]Lo standard più comune per le socket API è conosciuto come BSD socket; la maggior parte delle implementazioni di socket sono conformi a questo standard, compresi i socket Linux.

[2]L'API implementa inoltre le funzioni sendto() sendmsg(), recvfrom() e recvmsg() per l'invio e ricezione di dati attraverso socket; differiscono dalla send() e dalla recv() per dei parametri aggiuntivi

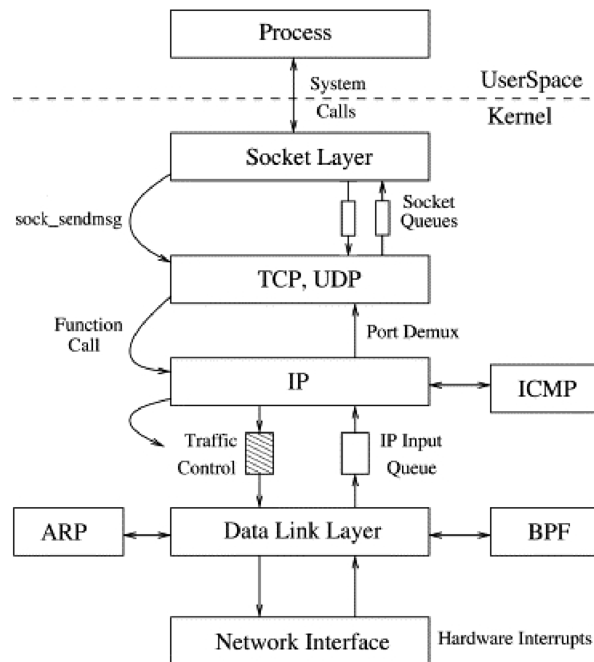

Figura 5.3: Schema del percorso seguito da un pacchetto all'interno del network stack.

5.2 Considerazioni

In base alla classificazione riportata nel capitolo 2, è evidente come l'utilizzo di una Macchina Virtuale di sistema per la realizzazione dello scenario descritto per il nodo mobile non sia una scelta appropriata; è necessario infatti considerare le caratteristiche del tipo di dispositivo su cui si vuole effettuare la virtualizzazione, ovvero smartphone o tablet, i quali non dispongono di elevate prestazioni; l'inserimento di uno strato di virtualizzazione completa condizionerebbe dunque eccessivamente le performance del device. Inoltre l'utilizzo di una Virtual Machine di sistema comporterebbe la necessità di installazione di un sistema operativo guest, ovvero una versione di Android che possa essere eseguita attraverso la VM su un dispositivo su cui è già presente il medesimo sistema operativo mobile, creando quindi un'inutile duplicazione dello stesso.

Quello che risulta essere quindi fondamentale è una Macchina Virtuale

di processo, in grado di ingabbiare l'applicazione VoIP evitando tuttavia di virtualizzare totalmente l'hardware sottostante; l'unica componente ad essere virtualizzata sarà infatti la rete, della quale l'applicazione avrà una visione personalizzata, attraverso un proprio stack TCP/IP. Come anticipato si è scelto di utilizzare umView come System Call Virtual Machine che avrà dunque il compito di intercettare le system call di rete dell'applicazione e di redirigerle attraverso lo stack TCP/IP virtuale.

È stato effettuato un porting parziale della suddetta Macchina Virtuale per il sistema operativo Android utilizzando l'Android Native Development Kit; alla descrizione degli strumenti utilizzati è dedicato il capitolo successivo.

Capitolo 6

Strumenti

Tra gli applicativi utilizzati all'interno del presente lavoro particolare attenzione va posta principalmente su due strumenti: la Virtal Machine umView e l'Android Native Development Kit(NDK), che verranno dettagliatamente descritti nel seguente capitolo.

6.1 umView

umView rappresenta una delle possibili implementazioni del progetto ViewOS e realizza una System Call Virtual Machine in grado di fornire al processo virtualizzato una visione completamente personalizzata dell'ambiente in cui viene eseguito. Come già anticipato nel capitolo 2, è utilizzata la system call `ptrace` per intercettare le chiamate di sistema effettuate dal processo e redirigerle ai moduli virtuali caricati.

6.1.1 Struttura

La struttura interna di umView è basata sull'utilizzo di *moduli*, ognuno dei quali permette di fornire differenti tipologie di virtualizzazione; un modulo al suo interno utilizza a sua volta una serie di sottomoduli per supportare particolari funzionalità ed estensioni e può essere caricato all'interno

di umView all'avvio della Virtual Machine oppure durante l'esecuzione della stessa.

Per tenere traccia di quelle che sono le virtualizzazioni attive in un dato momento, viene utilizzata una Hash Table globale in cui un particolare oggetto viene inserito al caricamento di ogni modulo; la Hash Table è in grado di gestire differenti tipi di oggetti tra i quali i principali sono:

- **Pathname:** Se la system call intercettata contiene al suo interno un percorso, come avviene ad esempio per la open o la stat, viene effettuata una ricerca all'interno dell'Hash Table per determinare se è presente una virtualizzazione per il path; un oggetto di questo tipo è aggiunto alla tabella hash attraverso un'operazione di mount.

- **Filesystem:** Questi oggetti sono utilizzati dalla system call mount; quando un modulo è caricato un oggetto Filesystem è aggiunto nella Hash Table.

- **Protocol Family:** Rappresentano il supporto alla virtualizzazione delle system calls socket.

Una componente fondamentale di umView è il *system call capture layer* che permette di determinare quelle che sono le system call effettuate dal processo controllato; l'intercettazione di una system call è quindi notificata al *dispatching layer* a cui è delegato il compito di cercare all'interno della Hash Table una possibile virtualizzazione per la chiamata di sistema intercettata ed eventualmente reindirizzarla verso il modulo appropriato.

6.1.2 Moduli

In umView un modulo è implementato come una libreria dinamica e può essere caricato a runtime tramite il processo virtualizzato (utilizzando il comando um_add_service *modulo*), oppure all'avvio della Macchina Virtuale attraverso il comando umview -p *modulo*.

I moduli attualmente implementati per umView sono i seguenti:

- **umfuse:** Fornisce ad umView il supporto ai filesystem virtuali ed è basato su FUSE[1], un progetto open source ideato per realizzare un'implementazione a livello utente di filesystem. Un filesystem implementato attraverso FUSE è costituito da un programma in cui è utilizzata una specifica interfaccia fornita dalla libreria FUSE; nel caso in cui un filesystem di questo tipo sia montato, qualsiasi azione che coinvolga l'utilizzo del filesystem è intercettata da uno specifico modulo del Kernel e inoltrata automaticamente al filesystem implementato a livello utente. FUSE è divenuto ufficialmente parte del codice del kernel Linux a partire dalla release 2.6.14.

  Umfuse fornisce all'interno di umView la medesima interfaccia e le stesse funzionalità di FUSE; a differenza di quest'ultimo però, in umfuse i filesystem sono realizzati come librerie dinamiche e non come programmi applicativi.

  I sottomoduli di umfuse attualmente sviluppati sono:

  - **umfuseext2:** implementazione di ext2.
  - **umfuseiso9660:** implementazione di iso9660.
  - **umfusefat:** implementazione di fat o vfat.
  - **umfusessh:** implementazione di file system remoti via ssh.
  - **umfuseencfs:** implementazione di file system criptati.

- **umdev:** Fornisce ad umView il supporto per l'utilizzo di device virtuali, implementati attraverso specifici sottomoduli, e caricati da umdev tramite la system call `mount`.

  - **umdevmbr:** sottomodulo per la creazione di device virtuali che permettano la gestione del Master Boot Rercord.
  - **umdevramdisk:** sottomodulo per la creazione di ramdisk virtuali.

[1] http://en.wikipedia.org/wiki/Filesystem_in_Userspace

- **umdevtap:** sottomodulo che fornisce un'interfaccia tun/tap virtuale.

Sono inoltre stati sviluppati dei sottomoduli per il testing come umdevnull, che implementa un device nullo simile a /dev/null in Linux, e umdevtrivhd che costituisce un esempio di ramdisk virtuale.

- **viewfs:** Modulo per la riorganizzazione del filesystem; offre la possibilità di nascondere o spostare file e directory senza modificare il filesystem reale, di ridefinire permessi e fondere directory reali e virtuali.

- **umbinfmt:** Offre la possibilità di definire nuovi formati eseguibili a livello utente; si basa sul modulo binfmt_misc fornito dal kernel Linux, che permette di compiere questa operazione senza la necessità di ricompilare l'intero kernel.

- **umnet:** Modulo per la virtualizzazione della rete e l'utilizzo di differenti stack TCP/IP da parte del processo virtualizzato; permette quindi di indicare per ogni network stack specifiche interfacce ed indirizzi.

  All'interno di questo modulo è stato sviluppato un meccanismo per superare due importanti limitazioni presenti nella gestione network in ambiente Linux; come anticipato nel capitolo precedente la programmazione di rete è infatti basata sulla Berkley Socket API, uno standard che si è imposto come riferimento per la programmazione di rete ma che presenta tuttavia un'importante limitazione: permette infatti l'utilizzo di un solo stack per ogni famiglia di protocolli, non consentendo quindi di definire esplicitamente per un particolare processo lo stack TCP/IP da utilizzare. Inoltre, nonostante all'interno dei sistemi Unix il filesystem sia utilizzato per la rappresentazione e l'accesso a svariati tipi di risorse (come files, dispositivi, sockets etc.), non è definita una rappresentazione del network stack. Questi due problemi sono stati superati in umnet introducendo un'estensione alla Berkley Socket API, denominata msosckets, che risulta essere retrocompatibile con le applicazioni

conformi allo standard originale e che introduce la possibilità di definire stack TCP/IP da poter utilizzare simultaneamente, permettendone la rappresentazione mediante filesystem.

Msockets API

È definito un particolare special file[2] per l'accesso allo stack TCP/IP, attraverso il quale l'utente può definire interfacce ed indirizzi (se ha permessi in esecuzione) oppure aprire un socket ed avviare una comunicazione dati (se ha i permessi in lettura).

La funzione *msocket* rappresenta la chiamata principale dell'API e permette di specificare lo stack TCP/IP da utilizzare; la sintassi è la seguente:

```
int msocket(char *path, int domain, int type,
            int protocol);
```

in cui i parametri domain, type e protocol corrispondono ai rispettivi parametri della funzione socket mentre il parametro path permette di specificare quale Stack special file utilizzare (se path ha valore nullo è utilizzato lo stack di default). È inoltre possibile ridefinire lo stack di default del processo chiamante utilizzando la funzione msocket con parametro type=SOCK_DEFAULT: effettuando ad esempio la chiamata

```
msocket("/dev/net/personal\_stack", PF\_INET,
         SOCK\_DEFAULT,0);
```

si imposta /dev/net/personal_stack come stack IPv4 di default per il processo chiamante.

[2]Nei sistemi Unix uno special file o device file è un'interfaccia per i device definita nel filesystem per permette l'interazione tra software e dispositivi utilizzando system call di I/O standard.

Il comando *mstack* permette di eseguire un determinato processo specificando esplicitamente lo stack da utilizzare; richiama infatti al suo interno la funzione msocket per la definizione dello stack a cui farà riferimento il processo da eseguire. Lanciando ad esempio il comando

```
mstack /dev/net/personal\_stack ip link
```

si esegue ip link attraverso lo stack /dev/net/personal_stack, utilizzando quindi le interfacce e gli indirizzi definiti da quest'ultimo.

I sottomoduli attualmente implementati per umnet sono:

- **umnetnull:** implementa uno stack di rete nullo, in cui le chiamate socket falliscono ritornando l'errore EAFNOSUPPORT per tutte le famiglie di protocolli. Può dunque essere utilizzato per impedire l'accesso alla rete ad un processo.
- **umnetnative:** fornisce uno stack special file che fa riferimento allo stack TCP/IP di default del sistema.
- **umnetlink:** permette di realizzare un link simbolico per un sottomodulo caricato precedentemente, con la possibilità di impostare esplicitamente per quali protocolli effettuare il linking.
- **umnetlwipv6:** realizza uno stack ibrido IPv4/IPv6; durante la fase di mount del dispositivo è possibile indicare quali interfacce definire nello stack: lwipv6 supporta infatti interfacce di tipo vde[3], tun e tap; se non è indicata alcuna opzione viene impostata un'unica interfaccia vde.

6.2 Android Native Development Kit

Il Native Development Kit è un set di strumenti che permette di integrare all'interno di applicazioni Android componenti scritte in codice nativo come

[3] Virtual Distributed Ethernet: http://vde.sourceforge.net/.

C e C++, consentendo quindi di bypassare la Dalvik Virtual Machine, all'interno della quale vengono eseguite generalmente tutte le applicazioni, ed accedere direttamente al codice macchina sottostante.

È possibile richiamare metodi implementati in codice nativo dall'interno del codice Java di un'applicazione Android attraverso l'utilizzo della JNI (Java Native Interface), uno specifico framework del linguaggio Java: nei sorgenti dell'applicazione è necessario dichiarare esplicitamente, attraverso la parola-chiave `native`,i metodi nativi che saranno utilizzati; l'implementazione di questi ultimi sarà contenuta all'interno di una libreria dinamica che dev'essere conforme allo standard UNIX, con prefisso lib e suffisso .so, e che dev'essere esplicitamente caricata all'interno dell'applicazione (ad esempio mediante l'istruzione: `static { System.loadLibrary("NomeLibreria");}`).

L'Android NDK va quindi ad ampliare quelle che sono le funzionalità offerte dall'Android SDK mettendo a disposizione del programmatore un set completo di strumenti per la cross-compilazione come compilatori, linker e specifici header di sistema; all'interno dell'NDK sono definiti infatti numerosi livelli di API (contenuti nella directory `$NDK/platform`)[4], ognuno dei quali contiene gli header e le librerie dinamiche necessarie per compilare correttamente un programma scritto in codice nativo per una determinata architettura (ARM, mips o x86) e release del sistema operativo. Ogni livello corrisponde infatti ad una specifica versione di Android secondo l'elenco riportato di seguito:

- **android-3:** Corrisponde alla versione 1.5 di Android.
- **android-4:** Corrisponde alla versione 1.6 di Android.
- **android-5:** Corrisponde alla versione 2.0 di Android.
- **android-6:** Corrisponde alla versione 2.0.1 di Android.
- **android-7:** Corrisponde alla versione 2.1 di Android.

[4]Con $NDK è indicata la directory di installazione dell'Android Native Development Kit.

- **android-8:** Corrisponde alla versione 2.2 di Android.
- **android-9:** Corrisponde alla versione 2.3 di Android.
- **android-14:** Corrisponde alla versione 4.0 di Android.

e fornisce dunque differenti librerie e header. È possibile specificare esplicitamente per quale particolare piattaforma e architettura compilare il proprio programma tramite il file Application.mk, utilizzando le variabili APP_ABI e APP_PLATFORM per definire rispettivamente l'Application Binary Interface (l'architettura per la quale produrre il codice macchina) e la versione di Android a cui fare riferimento. L'ABI di default, nel caso in cui non ne venga specificata una differente, è l' *armeabi*, che corrisponde al codice macchina per processore ARMv5TE.

6.2.1 Utilizzo dell'NDK

All'interno del presente lavoro il Native Development Kit è stato utilizzato per effettuare una cross-compilazione del codice sorgente della Macchina Virtuale umView, per rendere quest'ultima eseguibile su dispositivi mobili equipaggiati con sistema operativo Android.

Per compilare correttamente codice nativo tramite l'NDK è necessario creare all'interno della directory principale del proprio progetto, da qui in poi indicata con $PROJECT, tre sottodirectory da rinominare rispettivamente in: jni, libs e obj; nella prima devono essere inseriti i sorgenti C da compilare mentre nella seconda saranno posti automaticamente dal build system le librerie e gli eseguibili correttamente compilati; in obj si troveranno infine i file oggetto creati dall'assembler in fase di compilazione.

Android.mk

Per procedere alla compilazione dei sorgenti è necessario realizzare un particolare tipo di Makefile che prende il nome di Android.mk, al cui interno

è possibile definire differenti moduli per l'implementazione di librerie statiche, dinamiche ed eseguibili.

Di seguito è riportato un esempio di file Anrdoid.mk ed una descrizione delle principali variabili utilizzate al suo interno:

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

include $(CLEAR_VARS)

LOCAL_MODULE := nome_modulo
LOCAL_SRC_FILES := nome_sorgente.c
LOCAL_C_INCLUDE := dir/path_da_includere
LOCAL_CFLAGS := -I/path_da_includere -Wall
LOCAL_SHARED_LIBRARIES := libreria_personale_da_linkare
LOCAL_LDLIBS := -ldl -llibreria_di_sistema

include $(BUILD_EXECUTABLE)
```

All'inzio del file deve essere necessariamente definita la variabile LOCAL_PATH, utilzzata per indicare dove sono collocati i file sorgenti. Nell'esempio riportato la macro my-dir, fornita dal build system, ritorna il path della directory corrente (ovvero la directory contenente il file Android.mk).

La macro include $(CLEAR_VARS), fornita anch'essa dal build system, richiama un particolare script che provvede a inizializzare le variabili utilizzate (come ad esempio LOCAL_MODULE, LOCAL_SRC_FILES etc..); è necessario effettuare questa operazione ogni qual volta si definisce un nuovo modulo poichè ogni modulo specifica differenti valori per le variabili sopracitate.

Con la variabile LOCAL_MODULE si indica l'identificativo che verrà associato al modulo all'interno dell'Android.mk; questo dev'essere unico e privo di spazi e corrisponde al nome che il compilatore assegnerà all'eseguibile.

La variabile LOCAL_SRC_FILES permette di specificare i sorgenti C e C++ che saranno compilati ed assemblati all'interno dell'eseguibile. Non è necessario indicare esplicitamente gli header, in quanto il compilatore risolve

automaticamente le dipendenze necessarie.

Con LOCAL_C_INCLUDE e LOCAL_CFLAGS è possibile definire rispettivamente i path da includere in fase di compilazione e l'elenco di opzioni da passare al compilatore. Come mostrato nell'esempio, la variabile LOCAL_CFLAGS permette inoltre di specificare path di include aggiuntivi tramite l'opzione -I.

La variabile LOCAL_SHARED_LIBRARIES indica la lista delle librerie dinamiche, aggiuntive a quelle standard, da cui il modulo dipende a runtime; tramite questa variabile devono quindi essere specificate tutte quelle librerie create dall'utente di cui il modulo necessita.

Le librerie standard di sistema da passare al linker sono invece definite tramite la variabile LOCAL_LDLIBS utilizzando il prefisso -l. Le librerie di sistema attualmente implementate sono contenute all'intero nella directory $NDK/platforms/android-<APIlevel>/<targetArch>/usr/lib e, con riferimento all'API android-14, comprendono:

- **libC:** costituisce l'implentazione per Android della libreria C standard; prende il nome di *bionic* e presenta numerose differenze dalla corrispettiva GNU C Library (glibc). Data l'importanza che riveste all'interno del presente lavoro, nel paragrafo successivo saranno evidenziate nel dettaglio le caratteristiche di questa libreria.

- **libstdc++:** libreria estremamente ridotta per il supporto di codice scritto in C++.

- **libm:** implementazione per Android della libreria Math.

- **liblog:** libreria per l'invio da codice nativo di messaggi di log al kernel.

- **libZ:** libreria per la compressione di dati.

- **libdl:** libreria per il linking dinamico.

- **OpenGL ES 1.1 e 2.0:** librerie grafiche basate su OpenGL.

- **libjnigraphics:** libreria che fornisce un'interfaccia per consentire l'accesso da codice nativo ai pixel buffer di oggetti bitmap definiti in Java.

- **OpenSL ES:** libreria che permette gestire input e output audio in Android tramite codice nativo.

- **OpenMax AL :** libreria per la gestione di output multimediale all'interno di sorgenti scritti in codice nativo.

Alcune delle librerie sopraelencate, come libc, libstdc++ e libm, sono linkate automaticamente dal build system in fase di compilazione e non è quindi necessario indicarle esplicitamente tramite la variabile LOCAL_LDLIBS.

Infine la macro include $(BUILD_EXECUTABLE) presente nell'Android.mk è utilizzata per richiamare un particolare script a cui è delegato il compito di creare l'eseguibile in base alle informazioni esperesse dalle variabili LOCAL_XXX contenute nel modulo. È possibile creare inoltre librerie statiche e dinamiche sostituendo a $(BUILD_EXECUTABLE) rispettivamente le variabili $(BUILD_STATIC_LIBRARY) e $(BUILD_SHARED_LIBRARY).

Per ottenere un elenco maggiormente completo delle variabili definibili all'interno dei file Android.mk e Application.mk si rimanda alla documentazione ufficiale del progetto Android NDK.

ndk-build

Dalla versione r4 dell'NDK è stato introdotto lo script *ndk-build*, un semplice wrapper per il comando GNU Make che richiama automaticamente il corretto NDK build script a seconda delle informazioni contenute nei file Android.mk e Application.mk, permettendo quindi di semplificare la cross-compilazione di codice nativo. Lo script va lanciato dalla directory $PROJECT tramite linea di comando, e necessita di una versione di Gnu Make pari o superiore a 3.81.

6.2.2 Bionic

Come anticipato nel paragrafo precedente bionic rappresenta l'implementazione per Android della libreria Gnu Lib C , presentando però notevoli differenze rispetto a quest'ultima; l'idea alla base di bionic è infatti la semplicità, e questo comporta la mancanza al suo interno di numerose funzionalità messe a disposizione dalla libreria glibc. Bionic è stato infatti appositamente sviluppato per utilizzare quelle che sono le peculiarità del sistema operativo Android, che lo differenziano da GNU/Linux.

La libreria contiene al suo interno una specifica implementazione per la gestione dei threads; a differenza della glibc quindi per la compilazione di codice sorgente in cui sono utilizzati threads, non è necessario linkare esplicitamente la libreria pthread.

Gli eseguibili ottenuti attraverso l'utilizzo della libreria Bionic non sono compatibili con la libreria GNU C: questo comporta il rischio di incorrere in errori nel caso in cui venga linkato Bionic per un programma compilato utilizzando gli header di glibc; per eseguire quindi la compilazione di un programma basato su Bionic è necessario utilizzare il cross-compiler messo a disposizione dall'NDK.

Le numerose differenze tra Bionic e glibc hanno condizionato notevolmente il presente lavoro di tesi, portando alla realizzazione di una libreria aggiuntiva in cui sono state implementate alcune funzionalità mancanti in Bionic ma necessarie per la corretta esecuzione di umView in ambiente Android.

Ad una dettagliata descrizione delle funzioni implementate e delle modifiche apportate al codice della Virtual Machine è dedicato il capitolo 8.

Capitolo 7

Deployment

All'interno del presente capitolo verrà mostrato, attraverso un breve ma significativo esempio, come l'utilizzo della macchina virtuale *umView* e del modulo di virtualizzazione *umnet* permettano di realizzare in maniera completa ed efficace lo scenario introdotto nel capitolo 5.

7.1 Esempio

L' esempio qui riportato è stato realizzato in ambiente GNU Linux[1] poichè attualmente non è ancora possibile disporre del modulo umnet su piattaforma Android; ciò nonostante l'esempio dimostra chiaramente come la struttura modulare di umView ben si presti a fornire la virtualizzazione di rete necessaria per l'implementazione dello scenario.

Per semplicità si supponga che il programma da eseguire all'interno della Virtual Machine sia una shell bash; è quindi necessario lanciare da riga di comando la macchina virtuale umview indicando come argomento l'applicazione da virtualizzare:

```
umview bash
```

[1]Sistema operativo Linux Mint 14 Nadia con kernel Linux 3.5.0-17-generic i686.

A questo punto la shell bash è in esecuzione attraverso la virtual machine; tuttavia nessun modulo di virtualizzazione è stato ancora caricato e dunque il processo ha accesso diretto allo stack TCP/IP di sistema e alle interfacce di rete della macchina reale su cui è in esecuzione. Affinchè l'applicazione utilizzi uno stack TCP/IP virtuale è necessario caricare all'interno di umView il modulo umnet tramite il comando[2]:

```
um_add_service umnet
```

In questo modo tutte le system call di rete effettuate dal processo virtualizzato saranno gestite da umnet, che le inoltrerà sullo stack TCP/IP virtuale specificato dall'utente; all'interno del codice di umnet è riportata la definizione del tipo Stack e dello stack TCP/IP di default:

```
#define S_IFSTACK 0160000
#define DEFAULT_NET_PATH "/dev/net/default"
```

quest'ultimo sarà utilizzato dal modulo come stack virtuale sul quale inoltrare inizialmente il traffico di rete prodotto dall'applicazione; è possibile tuttavia definire differenti stack TCP/IP caricando all'interno di umnet appositi sottomoduli che li implementino. Attraverso il comando:

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

lo stack TCP/IP implementato nel sottomodulo `umnetnull` è caricato in umnet ed il path `/dev/net/null` definisce la sua rappresentazione all'interno del file system; è possibile a questo punto utilizzare il comando

```
mstack /dev/net/null ip link
```

per far sì che le operazioni di rete effettuate dal programma virtualizzato vengano inoltrate attaraverso lo stack virtuale nullo, come mostrato in Figura 7.1.

[2]Come anticipato nel capitolo 6 è possibile indicare all'avvio della virtual machine un modulo da precaricare tramite l'opzione -p.

```
Terminal
File  Edit  View  Search  Terminal  Help
slash@Hari ~ $ umview bash
slash@Hari ~ $ um_add_service umnet
umnet init
slash@Hari ~ $ ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 16436 qdisc noqueue state UNKNOWN mode DEFAULT
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc pfifo_fast state DOWN
 mode DEFAULT qlen 1000
    link/ether 90:fb:a6:a2:4a:8f brd ff:ff:ff:ff:ff:ff
3: wlan0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP mode DORMA
NT qlen 1000
    link/ether 78:e4:00:a9:40:19 brd ff:ff:ff:ff:ff:ff
slash@Hari ~ $ mount -t umnetnull none /dev/net/null
slash@Hari ~ $ mstack /dev/net/null ip link
Cannot open netlink socket: Address family not supported by protocol
slash@Hari ~ $
```

Figura 7.1: Utilizzo del modulo umnet con stack di rete nullo

La figura mostra come la prima esecuzione del comando `ip link` venga effettuata tramite il modulo umnet senza però far riferimento ad alcuno stack di rete virtuale; nel secondo caso invece, tramite il comando `mstack`, `ip link` è eseguito sullo stack di rete nullo, ritornando dunque l'errore EAFNOSUPPORT.

Uno stack virtuale implementato tramite sottomodulo può inoltre sostituire lo stack di default definito in umnet, indicando in fase di mount il path corrispondente:

```
mount -t umnetnull none /dev/net/default
```

In questo caso non è quindi necessario indicare attraverso il comando mstack il network stack da utilizzare poichè il traffico di rete prodotto dall'applicazione sarà automaticamente inoltrato sullo stack virtuale caricato.

In figura 7.2 è mostrato come la shell bash venga eseguita in umview precaricando il modulo umnet; l'aggiunta del sottomodulo `umnetnull`, come nell'esempio precedente, è effettuata tramite il `mount` su un particolare path, che in questo caso corrisponde allo stack di rete di default. Le successive operazioni di `ip link` è `telnet` falliscono quindi entrambe, venendo automaticamente indirizzate sullo stack di rete virtuale. È utile notare come in se-

guito all'`umount` dello stack nullo, il processo virtualizzato abbia nuovamente accesso allo stack di rete di sistema.

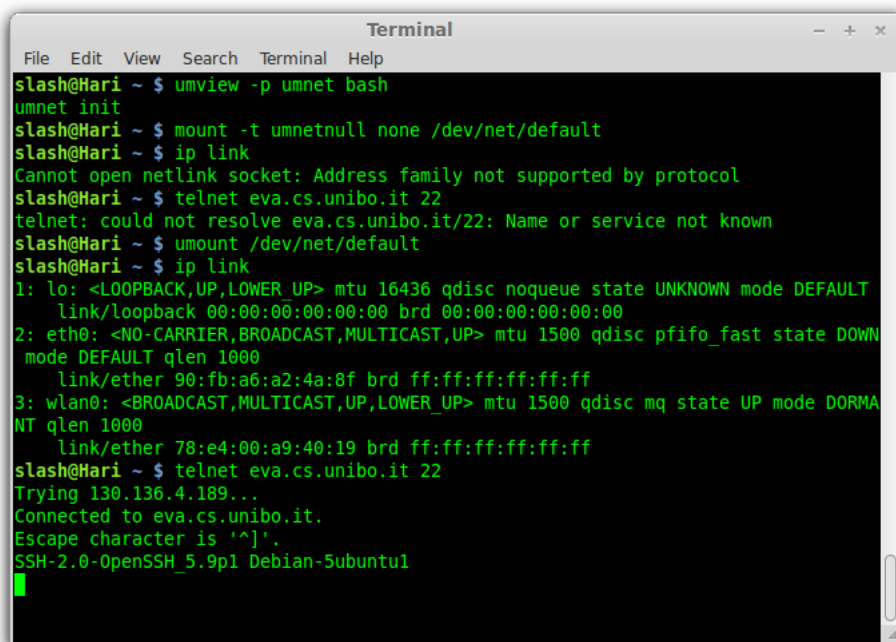

Figura 7.2: Esecuzione di operazioni di rete impostando un network stack nullo come stack di rete di default.

7.2 Considerazioni

Il precedente esempio dimostra come l'utizzo di umview permetta di far sì che un'applicazione acceda in maniera completamente automatica ad uno stack di rete virtuale; eseguendo infatti un'applicazione VoIP all'interno di un'instanza di umView in cui è stato precaricato il modulo umnet e lo stack da utilizzare, è possibile redirigere il traffico dati prodotto dall'applicazione su una specifica interfaccia, dalla quale il proxy client andrà a leggere i pacchetti da inoltrare verso le interfacce di rete del dispositivo.

Capitolo 8

Progettazione

Per rendere disponibile su piattaforma Android la Virtual Machine umView è stata effettuata una ricompilazione del codice sorgente utilizzando l'Android Native Development Kit; la versione di umView scelta è la 0.8.2 mentre la versione dell'NDK utilizzata è la r8d rilasciata nel Dicembre 2012. Le prove sperimentali sono state effettuate attraverso un Android Virtual Device (AVD), uno strumento messo a disposizione dall'Android SDK per la realizzazione di dispositivi virtuali che emulino completamente un device reale; nel dettaglio, la configurazione hardware e software dell'AVD utilizzato nel presente lavoro è la seguente:

```
Sistema Operativo: Android versione 4.1.2-API 16
Architettura Processore (ABI): Intel Atom x86
```

8.1 Porting

Nonostante ARM risulti essere l'architettura di riferimento nel panorama dei dispositivi mobili grazie all'ottimo rapporto fra il consumo richiesto e le prestazioni offerte, si è deciso di considerare unicamente device equipaggiati con processore x86 a causa della mancanza, all'interno del codice sorgente di umView, di un header che contenesse le specifiche relative ai registri per CPU ARM. Il riferimento a tali registri risulta essere necessario nel file

`defs.h` contenuto nella directory `xmview`, in cui sono definite costanti, macro e prototipi di funzioni utilizzate dal capture layer per intercettare le system call e apportare le opportune modifiche ai valori dei registri del processore tramite ptrace; tale operazione è strettamente dipendente dall'architettura per la quale si effettua la compilazione e per questo motivo all'interno del file è definita l'inclusione di uno specifico header sulla base di quest'ultima informazione:

```
// part of defs that's strictly architecture dependent
#if defined(__i386__)
//getregs/setregs and so on, for ia32
#       include "defs_i386_um.h"
#elif defined(__powerpc__)
//setregs/getresg and so on, for ppc
#       include "defs_ppc_um.h"
#elif defined(__x86_64__)
//setregs/getresg and so on, for ppc
#       include "defs_x86_64_um.h"
#else
#       error Unsupported HW Architecure
#endif /* architecture */
```

Com'è possibile notare non è presente un header contenente le specifiche relative all'architettura ARM.

8.1.1 Modifiche al codice sorgente

A causa delle numerose differenze esistenti tra la libreria standard C (GNU C Library) sulla quale umView si basa, e la sua implementazione per Android (Bionic), è stato necessario apportare alcune modifiche ai sorgenti della virtual machine; in Bionic non sono presenti infatti alcuni header e implementazioni di funzioni indispensabili per la corretta esecuzione di umView; nel seguente paragrafo saranno dunque descritte le modifiche apportate ai sorgenti e le funzionalità introdotte per ovviare a questo problema.

Header implementati

Un header non presente in Android ma necessario all'interno di umView è `wordsize.h` utilizzato nel sorgente `treepoch.c`, in cui è indicata la dimensione di una word in base all'architettura di riferimento.

```
/*Determine the wordsize from the preprocessor defines*/
#if defined __x86_64__
# define __WORDSIZE     64
# define __WORDSIZE_COMPAT32    1
#else
# define __WORDSIZE     32
#endif
```

Nel file `um_plusio.c` è presente inoltre un riferimento alla struttura `statfs64` definita, all'interno della GNU C Library, nell'header `bits/statfs.h`; Bionic non ne fornisce un'implementazione indicando tuttavia, nel file `$NDK/platforms /android-14/arch-x86/usr/include/sys/vfs.h`, la struct `statfs` come struttura corrispondente all'interno del proprio kernel:

```
/*note: this corresponds to the kernel's statfs64 type*/
struct statfs {
    uint32_t        f_type;
    uint32_t        f_bsize;
    uint64_t        f_blocks;
    uint64_t        f_bfree;
    uint64_t        f_bavail;
    uint64_t        f_files;
    uint64_t        f_ffree;
    __kernel_fsid_t f_fsid;
    uint32_t        f_namelen;
    uint32_t        f_frsize;
    uint32_t        f_spare[5];
};
```

Gli header mancanti sono stati dunque implementati e inseriti nella directory `include_adds`.

Definizione _UTSNAME_LENGTH

All'interno dell'header `um_lib.h` è contenuta la struttura `viewinfo` così definita:

```
struct viewinfo {
        struct utsname uname;
        pid_t serverid;
        viewid_t viewid;
        char viewname[_UTSNAME_LENGTH];
}
```

in cui è la dimensione della stringa `viewname` è quindi pari a `_UTSNAME_LENGTH`; nella GNU C Library la definizione di quest'ultima costante è contenuta all'interno dell'header `<bits/utsname.h>`, corrisponde al valore 65 ed indica la dimensione dei campi della struttura `utsname` definita nell'header `<sys/utsname.h>`

```


FILE <bits/utsname.h>
/* Length of the entries in ‘struct utsname’ is 65.  */
#define _UTSNAME_LENGTH 65



FILE <sys/utsname.h>
#ifndef _UTSNAME_SYSNAME_LENGTH
# define _UTSNAME_SYSNAME_LENGTH _UTSNAME_LENGTH
#endif
/* Structure describing the system and machine.  */
struct utsname
{
  char sysname[_UTSNAME_SYSNAME_LENGTH];
```

```
  char nodename[_UTSNAME_NODENAME_LENGTH];
  char release[_UTSNAME_RELEASE_LENGTH];
  char version[_UTSNAME_VERSION_LENGTH];
  char machine[_UTSNAME_MACHINE_LENGTH];
#if _UTSNAME_DOMAIN_LENGTH - 0
/* Name of the domain of this node on the network.*/
# ifdef __USE_GNU
    char domainname[_UTSNAME_DOMAIN_LENGTH];
# else
    char __domainname[_UTSNAME_DOMAIN_LENGTH];
# endif
#endif
};
```

All'interno di Bionic tuttavia la costante `_UTSNAME_LENGTH` risulta essere assente e la struttura `utsname`, presente nel file `$NDK/platforms/android-14/arch-x86/usr/include/sys/utsname.h`, è dunque così definita:

```
#define SYS_NMLN 65

struct utsname {
    char  sysname   [SYS_NMLN];
    char  nodename  [SYS_NMLN];
    char  release   [SYS_NMLN];
    char  version   [SYS_NMLN];
    char  machine   [SYS_NMLN];
    char  domainname[SYS_NMLN];
};
```

La definizione della costante `_UTSNAME_LENGTH` è stata quindi aggiunta all'interno del file `um_lib.h`

```
#ifndef _UTSNAME_LENGTH
#define _UTSNAME_LENGTH SYS_NMLN
#endif
```

rimanendo coerenti con la definizione presente nella libreria standard C.

Definizione __fd_mask

Un ulteriore lacuna evidenziata in Bionic nel corso della ricompilazione del del codice di umView riguarda l'assenza della costante `__fd_mask` utilizzata all'interno del sorgente `um_select.c`:

```
static void getfdset(long addr,struct pcb* pc, int max,
                     fd_set *lfds)
{
   FD_ZERO(lfds);
         if (addr != umNULL)
           umoven(pc,addr,(__FDELT(max)+1)*
                  sizeof(__fd_mask),lfds);
}

static void putfdset(long addr, struct pcb* pc, int max,
                     fd_set *lfds)
{
        if (addr != umNULL)
                ustoren(pc,addr,(__FDELT(max)+1)*
                        sizeof(__fd_mask),lfds);
}
```

Nella GNU C Library la definizione di `__fd_mask` è contenuta all'interno dell'header di sistema `<sys/select.h>` e, com'è possibile notare, è utilizzata all'interno dell'header stesso per la definizione della costante `__NFDBITS` e della macro `__FD_MASK(d)`:

```
/* The fd_set member is required to be an array of longs.
*/
typedef long int __fd_mask;
#define __NFDBITS  (8 * (int) sizeof (__fd_mask))
#define __FD_MASK(d)  ((__fd_mask) 1 << ((d)%__NFDBITS))
```

Queste ultime due definizioni sono tuttavia presenti anche in Bionic all'interno dell'header `$NDK/platforms/android-14/arch-x86/usr/include/linux/posix_types.h`:

```
#define __NFDBITS (8 * sizeof(unsigned long))
#define __FDMASK(d) (1UL << ((d) % __NFDBITS))
```

È evidente come le dichiarazioni di `__NFDBITS` e `__FD_MASK(d)` contenute nelle due librerie siano sostanzialmente coincidenti ad eccezione dell'utilizzo della costante `__fd_mask` che nel secondo caso è sostituita dalla dicitura `unsigned long`; per tale motivo all'interno del file `um_select.c` è stata aggiunta la definizione

```
typedef unsigned long   __fd_mask;
```

che risulta essere concorde con le definizioni presenti nel relativo header.

Supporto per ptrace

In Bionic è fornita una differente implementazione della system call `ptrace`, utilizzata in umView per l'intercettazione delle chiamate di sistema del processo virtualizzato; quest'ultima permette infatti ad un processo, chiamato *tracer*, di controllare l'esecuzione di un secondo processo, che prende il nome di *tracee*, esaminandone e modificandone la rispettiva area di memoria; attraverso il parametro `request` della funzione `ptrace`, il tracer può specificare una serie operazioni da compiere sul processo controllato.

Nella libreria standard C la definizione della funzione `ptrace` è contenuta nell'header di sistema `sys/ptrace.h`:

```
extern long int ptrace (enum __ptrace_request __request,
                        ...) __THROW;
```

Nello stesso file è inoltre definita l'enumerazione `enum __ptrace_request` in cui sono elencate le operazioni che è possibile indicare come parametro `request` della funzione `ptrace`.

All'interno dei sorgenti di umView l'header `defs_i386_um.h` definisce la funzione `setregs` per la modifica e l'impostazione dei registri del processo controllato e, in accordo con la definizione di `ptrace` appena descritta, indica per il parametro `call` il tipo `enum __ptrace_request`:

```
static inline long setregs(struct pcb *pc,
                           enum __ptrace_request call,
                                   long op, long sig){
...}
```

Tuttavia, come precedentemente anticipato, Bionic fornisce un'implementazione differente per la funzione `ptrace`, contenuta all'interno dell'header `$NDK/platforms/android-14/arch-x86/usr/include/sys/ptrace.h`:

```
extern long ptrace(int request, pid_t pid,
                   void *addr, void *data);
```

Com'è possibile notare a differenza della definizione presente nella GNU C Library, il parametro formale `request` ha tipo intero e nel file `$NDK/platforms/android-14/arch-x86/usr/include/linux/ptrace.h` sono definite, nella forma di singole costanti intere, le operazioni che è possibile indicare. Per permettere dunque la corretta compilazione dell'header `defs_i386_um.h` attraverso Bionic, la definizione della funzione `setregs` è stata così modificata:

```
static inline long setregs(struct pcb *pc,
                           int call,
                           long op, long sig){
...}
```

Directory Temporanea

Nella funzione `um_proc_open`, contenuta nel file sorgente `um_proc.c`, è stato modificato il path in cui la virtual machine crea la directory per la memorizzazione dei file temporanei relativi al processo virtualizzato. Ciò si è rivelato necessario poichè il path originale, `/tmp/.umview`, non è disponibile

su sistema operativo Android.
Nello stesso sorgente è stata inoltre commentata la funzione `rec_rm_all` a causa di un errore evidenziato in fase di compilazione[1]; tale scelta è non pregiudica il corretto funzionamento della virtual machine ma comporta la necessità di eliminare manualmente la directory temporanea creata a runtime.

Backtrace

Il file `gdebug.c`, presente all'interno dei sorgenti di umView, contiene le definizioni di alcune funzioni utilizzate per effettuare operazioni di debug sul codice; in particolare la funzione `fgbacktrace` richiama le funzioni `backtrace` e `backtrace_symbols` definite all'interno dell'header `<execinfo.h>`; queste ultime permettono di determinare il backtrace del thread corrente, definito come la lista delle chiamate attive all'interno del processo. La versione corrente di Bionic non offre tuttavia funzionalità di backtrace e per tale motivo si è deciso di eliminare dal file `gdebug.c` l'implementazione di `fgbacktrace`, dal momento che quest'ultima non risulta essere utilizzata all'interno degli altri sorgenti. Sempre nel file `gdebug.c` è stato modificato il riferimento alla libreria `libc.so` contenuto nel costruttore del sorgente[2], poichè a differenza di quanto accade nei sistemi Linux in cui è utilizzato il soname `libc.so.6`, in Android quest'ultimo assume semplicemente il nome di `libc.so`.

Sorgenti esclusi dalla compilazione

Durante la cross-compilazione del file `loginshell.c` è stata riscontrata l'assenza, all'interno dell'header `pwd.h` implementato per Android, della funzione `getpwuid_r`; com'è possibile infatti notare, all'interno dell'header è presente la seguente dichiarazione:

[1] In function '`rec_rm_all`': 330:15: error: dereferencing pointer to incomplete type.

[2] Per costruttori si intende un particolare tipo di funzioni, definite dal compilatore GCC, attraverso le quali è possibile indicare all'interno di un programma delle azioni da eseguire prima dell'esecuzione della funzione main.

```
#if 0 /* MISSING FROM BIONIC */
int getpwnam_r(const char*, struct passwd*, char*,
                size_t, struct passwd**);
int getpwuid_r(uid_t, struct passwd*, char*,
                size_t, struct passwd**);
struct passwd* getpwent(void);
int setpwent(void);
#endif /* MISSING */
```

Per tale motivo si è deciso di escludere dalla compilazione il file sorgente `loginshell.c`, rimuovendo dunque da umView la possibilità di lanciare la virtual machine come shell di login; tale funzionalità non risulta essere comunque essenziale per il presente lavoro e dunque l'esclusione del file non pregiudica la corretta implementazione dello scenario.

Parametri di compilazione

All'interno del file Android.mk, tra i parametri di compilazione del modulo *umview* è stata rimossa la definizione di `_UM_EPOLL`, presente invece nel Makefile.am originale per l'utilizzo di `epoll`[3]. La necessità di apportare tale modifica deriva dall'assenza, all'interno dell'header `sys/epoll.h` implementato per Android, della definizione delle costanti `EPOLL_CLOEXEC` ed `EPOLL_NONBLOCK` utilizzate nel file `um_select.c`; tuttavia, come documentato all'interno del codice, il supporto per epoll in umView risulta essere ancora in fase di sviluppo e non costituisce dunque una funzionalità indispensabile per il corretto funzionamento della macchina virtuale.

Sono state inoltre definite le costanti `MODULES_DIR`, `USER_MODULES_DIR` e `LIBEXECDIR` attraverso le quali sono indicati i path in cui saranno contenuti i moduli di virtualizzazione all'interno del dispositivo Android.

[3]Meccanismo di I/O notification introdotto nel kernel Linux dalla versione 2.5.44 e si propone di sostituire le system call POSIX `select` e `poll`

8.1.2 Libreria implementata

Per rendere possibile il porting di umView su Android è stata sviluppata una libreria aggiuntiva in cui sono implementate alcune funzioni assenti in Bionic ma fondamentali per la corretta esecuzione della virtual machine.

mempcpy

La funzione `mempcy`, il cui prototipo è contenuto nell'header `string.h`, è definita come segue:

```
void *mempcpy(void *dest, const void *src, size_t n);
```

Permette di copiare *n* bytes dall'area di memoria indicata con *src* all'area di memoria a cui punta *dst*, svolgendo dunque la stessa operazione della funzione memcpy; tuttavia, al contrario di quest'ultima, non ritorna un puntatore a *dst*, bensì un riferimento al byte successivo all'ultimo byte scritto.
È utilizzata nel file `canonicalize.c`, all'interno della funzione `rec_realpath`, ed è stata implementata come segue:

```
void *mempcpy(void *dest, const void *src, size_t n) {
  return (char *) memcpy (dest, src, n) + n;
}
```

fmemopen

```
FILE *fmemopen(void *buf, size_t size, const char *mode);
```

La funzione permette di aprire uno stream per effettuare operazioni di I/O sulla stringa o il buffer di memoria *buf*; il parametro *mode* permette di indicare la modalità di accesso al buffer mentre *size* ne indica la dimensione massima. I possibili valori dell'argomento *mode* coincidono con quelli indicabili per la funzione `fopen`. `fmemopen` è richiamata all'interno del file sorgente `services.c`, nella funzione `list_services`:

```
int list_services(char *buf,int len)
{
        FILE *f=fmemopen(buf,len,"w");
        forall_ht_tab_do(CHECKMODULE,list_item,f);
        fclose(f);
        return(strlen(buf));
}
```

clone

Citando il Manuale di Programmazione Linux "`clone()` creates a new process, in a manner similar to `fork`. It is actually a library function layered on top of the underlying `clone()` system call..."[4]; la funzione può essere dunque utilizzata per creare un nuovo processo tramite un processo padre, ed all'interno di umView è richiamata nel sorgente `ptrace_multi_test.c`.
Ricercando la definizione di `clone` nei sorgenti di Bionic, reperibili all'indirizzo `https://code.google.com/p/android-source-browsing/source/browse?repo=platform--bionic`, è possibile notare come questa sia apparsa per la prima volta nella revision `97cf7f339478` datata 22/01/2010 e sia contenuta nel file `bionic_clone.c`; tuttavia, come indicato dallo sviluppatore nel Change Log, questa versione rendeva disponibile le funzionalità offerte dalla `clone` unicamente per architettura ARM a causa della mancata implementazione della system call `__bionic_clone` in linguaggio assembly per architettura x86.
Con la revision `22d366cc0938` datata 08/08/2012, tuttavia, è stato introdotto il supporto alla `clone` anche per quest'ultima architettura, definendo dunque all'interno del file `clone.S` la relativa implementazione in assembly della system call.
Nonostante l'ultima release dell'Android NDK risalga al Dicembre 2012, e sia dunque successiva all'introduzione in Bionic della `clone` per architettura

[4] Trad: clone() crea un nuovo processo, in maniera simile alla fork. È una funzione di libreria che si interfaccia con la system call clone() sottostante...

x86, nella libreria `libc.so` in essa contenuta risulta essere presente l'implementazione della funzione `__bionic_clone` unicamente per ARM; ciò è stato riscontrato ricercando all'interno delle versioni di `libc.so` relative alle due architetture i riferimenti alla funzione `clone` mediante l'utilizzo del comando `readelf`[5]:

```
$ readelf -Ws arch-arm/usr/lib/libc.so | grep clone
  31: 0000881c 20 FUNC GLOBAL ... 4 __bionic_clone
  32: 00008830 20 FUNC GLOBAL ... 4 __bionic_clone_entry
  131: 00008fec 20 FUNC GLOBAL ... 4 __pthread_clone
  206: 000095c8 20 FUNC GLOBAL ... 4 __sys_clone
  286: 00009c08 20 FUNC GLOBAL ... 4 clone
  35: 0000881c 20 FUNC GLOBAL ... 4 __bionic_clone
  36: 00008830 20 FUNC GLOBAL ... 4 __bionic_clone_entry
  135: 00008fec 20 FUNC GLOBAL ... 4 __pthread_clone
  210: 000095c8 20 FUNC GLOBAL ... 4 __sys_clone
  290: 00009c08 20 FUNC GLOBAL ... 4 clone

$ readelf -Ws arch-x86/usr/lib/libc.so | grep clone
  108: 00008467 5 FUNC GLOBAL ... 4 __pthread_clone
  184: 000085e3 5 FUNC GLOBAL ... 4 __sys_clone
  110: 00008467 5 FUNC GLOBAL ... 4 __pthread_clone
  186: 000085e3 5 FUNC GLOBAL ... 4 __sys_clone
```

Le funzioni presentate all'interno di questo paragrafo sono state implementate attraverso la libreria `bionic_addon`, che dev'essere dunque inclusa fra le librerie da linkare dinamicamente in fase di compilazione del modulo *umview*. Alla valutazione delle funzionalità introdotte in Bionic e della bontà del porting effettuato è dedicato il capitolo successivo.

[5]`readelf` permette di ottenere informazioni riguardo un oggetto ELF (Exutable and Linkable Format), ovvero il formato di file adottato dai sistemi Unix-like per eseguibili, librerie e file oggetto.

Capitolo 9

Valutazione

Sono state effettuate delle prove sperimentali al fine di testare e valutare il corretto funzionamento dei moduli implementati nella libreria `bionic_addon` e della virtual machine *umView* in ambiente Android. I test presentati sono stati effettuati mediante un Android AVD con specifiche descritte nel capitolo precedente, e riguardano nel dettaglio la possibilità di utilizzo delle funzioni `fmemopen` e `clone` e la virtualizzazione di eseguibili tramite umView su sistema operativo Android 4.1.2.

Il trasferimento e l'esecuzione sull'emulatore dei file sviluppati in fase di progettazione sono stati effettuati attraverso l'Android Debug Bridge (ADB)[1], uno strumento a linea di comando per la comunicazione con un AVD o un device Android reale connesso al computer tramite interfaccia usb. Nel dettaglio, l'Android Debug Bridge è costituito da tre componenti:

- Un client, in esecuzione sul computer utilizzato per lo sviluppo, richiamabile all'interno di una shell bash tramite il comando `adb`.
- Un demone, eseguito in background sull'AVD o il device Android.
- Un server, implementato come processo in background in esecuzione sul computer, gestisce la comunicazione tra il client e il demone presente sul dispositivo.

[1] L'Android Debug Bridge è incluso all'interno dell'Android SDK.

Attraverso il comando `adb device` è possibile determinare i device connessi al computer o gli emulatori attualmente in esecuzione:

```
$ adb devices
  List of devices attached
  emulator-5554 device
```

mentre il comando `adb shell` permette di ottenere una shell remota interattiva tramite la quale comunicare direttamente col sistema operativo del dispositivo; quest'ultimo comando è stato dunque utilizzato per l'effettuazione dei test di valutazione sull'Android Virtual Device.

Per un elenco più dettagliato delle funzionalità offerte da `adb` si rimanda alla documentazione ufficiale dello strumento, ottenibile attraverso il comando `adb help`.

9.1 Bionic_addon

Sono stati sviluppati due semplici esempi per determinare la correttezza dell'implementazione delle funzioni `fmemopen` e `clone` all'interno della libreria; poichè quest'ultima non è contenuta nel path standard in cui il linker ricerca le librerie dinamiche da caricare a runtime, per una corretta esecuzione sull'emulatore di entrambi i programmi è necessario esportare la variabile d'ambiente `LD_LIBRARY_PATH`, indicando il path in cui la libreria `bionic_addon` è situata.

```
export LD_LIBRARY_PATH=/data/local/bin/bionic_addon
```

9.1.1 Fmemopen test

Il programma sviluppato per verificare l'implementazione della funzione è molto semplice e consiste nel leggere e stampare a video i caratteri di una stringa contenuta in un buffer, attraverso l'utilizzo della `fmemopen`:

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

static char buffer[] = "So long, and thanks for all\
                        the fish";

int main (void)
{
       int ch;
       FILE *stream;

       stream = fmemopen (buffer, strlen (buffer), "r");
       while ((ch = fgetc (stream)) != EOF)
         printf ("%c", ch);
       printf("\n");
       fclose (stream);
       return 0;
}
```

Il sorgente è stato compilato attraverso l'Android NDK ed eseguito sull'Android Virtual Device, senza riscontrare errori ed ottenendo dunque la funzionalità desiderata.

9.1.2 Clone test

Nel programma d'esempio sviluppato per testare la correttezza della funzione `clone`, il processo principale effettua una `open` sul file `/dev/null`[2] ottenendo un file descriptor attraverso il quale compiere operazioni sul file aperto; viene quindi richiamata la system call `clone` per la creazione di un nuovo processo, attendendo poi la terminazione di quest'ultimo tramite l'utilizzo della `waitpid`. Il processo creato richiama la funzione `close` sul file descriptor ri-

[2] Nei sistemi operativi Unix-like, `/dev/null` è un file virtuale con la caratteristica di non memorizzare i dati che vengono scritti su di esso.

cevuto come parametro, eliminando così qualsiasi riferimento al file; questa rappresenta l'unica operazione svolta dal processo, che a questo punto termina la propria esecuzione. Il controllo torna quindi al processo iniziale il quale tenta di scrivere sul file `/dev/null` attraverso il file descriptor aperto in precedenza, ricevendo come previsto l'errore `Bad file descriptor`.

```
#include <malloc.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>
#include <linux/sched.h>
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include <stdlib.h>

#define STACK 1024*64

int threadFunction( void* argument ) {
     close((int*)argument);
     return 0;
}
int main() {
     void* stack;
     pid_t pid;
     int fd;
     fd = open("/dev/null", O_RDWR);

     stack = malloc(STACK);

     pid = clone(&threadFunction,
                 (char*) stack + STACK,
                 SIGCHLD | CLONE_FS | CLONE_FILES |\
                  CLONE_SIGHAND | CLONE_VM,
```

```
                 (void*)fd);

    pid = waitpid(pid, 0, 0);

    if (write(fd, "c", 1) < 0) {
        printf("fd closed: %s\n", strerror(errno));
    }
    free(stack);
    return 0;
}
```

Come in precedenza il programma di test è stato eseguito attraverso l'emulatore Android, dimostrando in questo modo la correttezza dell'implementazione fornita.

9.2 umView

Per procedere alla virtualizzazione di un processo attraverso la virtual machine *umView* sull'Android Virtual Device, è necessario inizializzare le variabili d'ambiente `LD_LIBRARY_PATH`, indicando il path in cui sono contenute le librerie necessarie per l'esecuzione di umview[3], e `HOME` poichè quest'ultima risulta essere assente all'interno del sistema operativo.
La correttezza del porting effettuato è stata verificata eseguendo attraverso la virtual machine una shell bash, accessibile in Android tramite il path `/system/bin/sh`; com'è possibile notare nella Figura 9.1, l'esecuzione del processo all'interno della macchina virtuale umView non ne pregiudica il corretto funzionamento, ma offre bensì la possibilità di modificare la visione che il processo ha dell'infrastruttura di rete del dispositivo, mediante l'eventuale caricamento di moduli di virtualizzazione aggiuntivi.

[3] Le librerie necessarie da caricare a runtime sono `bionic_addon` e `um_lib`.

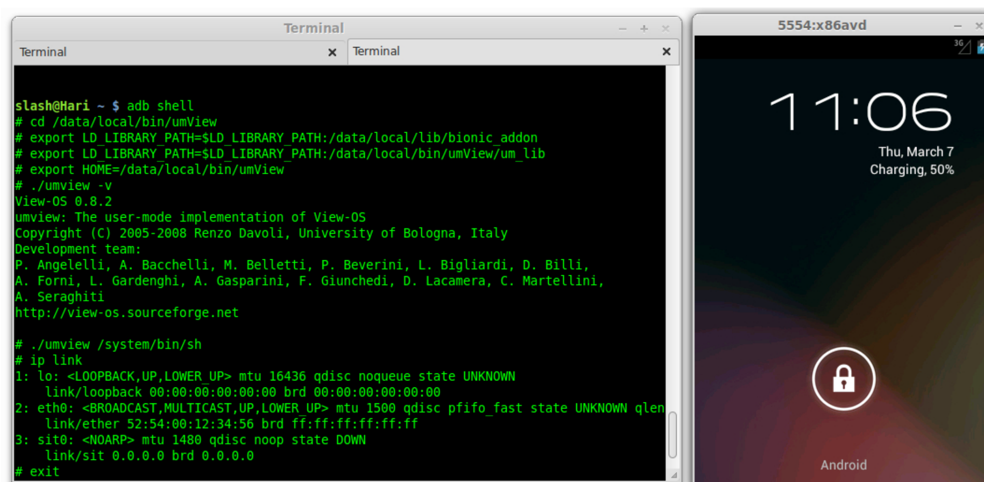

Figura 9.1: Esecuzione della shell bash tramite umView su emulatore Android.

9.2.1 Stato attuale del porting

Al momento sono stati correttamente portati sul sistema operativo Android i seguenti componenti del progetto umView:

- umlib, ovvero la libreria necessaria per la gestione dei servizi a livello utente.

- I comandi:
  - um_add_service
  - um_del_service
  - um_ls_service
  - um_add_service
  - um_aliasimplementata
  - um_fsalias
  - um_shutdown
  - viewmount
  - viewname

  - viewsu
  - viewsudo
  - viewumount
  - vuname

  utilizzabili all'interno della virtual machine.

- umview, che costituisce il nucleo della macchina virtuale.

Eventuali sviluppi futuri sono introdotti all'interno del capitolo successivo.

Capitolo 10

Conclusioni e sviluppi futuri

A partire dai dati raccolti e dalle soluzioni proposte nel presente lavoro di tesi, è evidente come sia possibile ampliare il supporto alla mobilità per sistemi multihomed eterogenei; il modello ABPS descritto all'interno del capitolo 4 permette infatti di migliorare notevolmente la qualità delle comunicazioni VoIP tra dispositivi mobili, consentendo l'utilizzo contemporaneo di più interfacce wireless e l'inoltro dei datagrammi prodotti dall'applicazione sull'interfaccia che fornisce maggiori garanzie. A partire da tale architettura, nella presente tesi si è proceduto alla ricerca e allo sviluppo di un meccanismo che permettesse di aumentare l'integrazione del proxy client all'interno di dispositivi mobili equipaggiati con sistema operativo Android, consentendo l'esecuzione di un'applicazione attraverso una macchina virtuale. In questo modo è possibile far sì che l'applicazione invii automaticamente i datagrammi attraverso l'interfaccia di rete virtuale, dalla quale ABPS andrà quindi a leggere pacchetti da inoltrare sulle interfacce di rete reali.
A tale scopo è stato realizzato il porting su sistema operativo Android di umView, una system call virtual machine in grado di fornire all'applicazione eseguita al suo interno una visione personalizzata della configurazione di rete del dispositivo; la ricompilazione del codice sorgente è stata effettuata utilizzando l'Android NDK, ed i test effettuati tramite un emulatore hanno confermato la bontà del lavoro svolto.

Tuttavia, il porting realizzato risulta essere ancora parziale, e per poter implementare completamente lo scenario descritto è necessario ampliare le funzionalità fino ad ora offerte.

In prima istanza è essenziale completare il porting degli strumenti implementati nel progetto umView rendendo disponibile su Android il modulo umnet, in modo da ottenere la virtualizzazione delle system call di rete effettuate dall'applicazione VoIP; successivamente è necessario implementare un sottomodulo di umnet che realizzi uno stack TCP/IP ad hoc, per consentire al proxy client di reperire attraverso le interfacce di rete virtuali definite nello stack i datagram prodotti dall'applicazione.

Un ulteriore sviluppo futuro potrebbe coinvolgere il porting della virtual machine su dispositivi Android dotati di architettura ARM, attraverso l'implementazione degli header necessari per la compilazione di umview per tale categoria di processori; un'evoluzione dello scenario in tal senso permetterebbe di ampliare notevolmente la diffusione del modello proposto, considerata l'elevata percentuale di device in commercio basati su questo tipo di architettura.

Bibliografia

[1] *IEEE 802.11i: Amendment 6: Medium Access Control (MAC) Security Enhancements*. 2004.

[2] *IEEE 802.11e: Wireless LAN Medium Access Control and Physical Layer Specification: Amendment for Quality of Service Enhancements*. 2005.

[3] *IEEE 802.11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*. 2007.

[4] *IEEE 802.11r: Wireless LAN Medium Access Control and Physical Layer Specification: Amendment for Fast BSS Transition*. 2007.

[5] *IEEE 802.11n Amendment 5: Enhancements for Higher Throughput*. 29 Ottobre 2009.

[6] C. Aoun and E. Davies. *RFC 4966 - Reasons to Move the Network Address Translator - Protocol Translator (NAT-PT) to Historic Status*. Luglio, 2007.

[7] Humphrey Cheung. The feds can own your wlan too. www.smallnetbuilder.com, Marzo 2005.

[8] Renzo Davoli and Michael Goldweber. *View-OS: Change your View of Virtualization*. 2009.

[9] Renzo Davoli, Michael Goldweber, and Ludovico Gardenghi. *UMview: View-OS implemented as a System Call Virtual Machine*. 2006.

[10] S. Deering and R. Hinden. *RFC 2460 - Internet Protocol, Version 6 (IPv6) Specification*. Dicembre, 1998.

[11] Vittorio Ghini, Giorgia Lodi, and Fabio Panzieri. *The Always Best Packet Switching architecture for SIP-based mobile multimedia servi-ces*, volume 84. Journal of Systems and Software, Elsevier Science Inc, Novembre 2011.

[12] Thomas F. Herbert. *The Linux TCP/IP Stack: Networking for Embedded Systems*. 2004.

[13] Ravi Nair J.E.Smith. *The architecture of virtual machines*, pages 32–38. Maggio, 2005.

[14] Felice Le Monnier, editor. *Vocabolario della lingua italiana*, page 1291. Luglio, 1987.

[15] E. Nordmark and R. Gilligan. *RFC 4213 - Basic Transition Mechanisms for IPv6 Hosts and Routers*. Ottobre, 2005.

[16] Information Sciences Institute University of Southern California. *RFC 791 - Internet Protocol*. 1981.

[17] J. Postel. *RFC 768 - User Datagram Protocol*. 28 Agosto 1980.

[18] K. Ramakrishnan, S. Floyd, and D. Black. *RFC 3168 - The Addition of Explicit Congestion Notification (ECN) to IP*. 1981.

[19] Fahmida Y. Rashid. Ipv4 address depletion adds momentum to ipv6 transition. www.eweek.com, Febbraio, 2011.

[20] Andrew S. Tanenbaum. *Computer Networks*. 1996.

[21] G. Tsirtsis and P. Srisuresh. *RFC 2766 - Network Address Translation - Protocol Translation (NAT-PT)*. Febbraio, 2000.

Ringraziamenti

Grazie ai miei genitori, per aver creduto in me fino alla fine. A mia madre, che attraverso le sue continue esortazioni mi ha aiutato lungo tutto questo percorso, e a mio padre, che con i suoi silenzi e i suoi consigli è sempre stato in grado di capirmi e darmi forza. Mi avete sempre sostenuto in ogni scelta e non potrò mai ringraziarvi abbastanza per questo.

Grazie a Eleonora, per essermi stata accanto in questi ultimi mesi frenetici. Senza di te non ce l'avrei mai fatta.

Un ringraziamento particolare inoltre al professor Ghini, per l'attenzione e l'infinita disponibilità dimostratami. È stato un piacere lavorare con Lei.