Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

Virtualizzazione parziale del livello Transport ai fini della gestione della mobilità

Relatore:
Chiar.mo Prof.
Vittorio Ghini

Presentata da:
Raffaele Lovino

Sessione II
Anno Accademico 2013/2014

No matter how low you are,

there's always someone to look down upon

Entombed - Contempt

Introduzione

La *comunicazione* è uno dei concetti fondamentali alla base della rivoluzione tecnologica degli ultimi anni, rivoluzione che trova il suo punto di massima espressione con la nascita del World Wide Web a cavallo tra gli anni '80 e '90 ad opera di **Tim Berners-Lee**[1]. Da semplice progetto utilizzato solo in ambiento accademico, il Web è diventato in pochissimo tempo veicolo di conoscenza e mezzo di comunicazione che, tra i tanti pregi e difetti, ha letteralmente abbattuto le barriere geografiche che separano i popoli, fornendo inoltre nuovi supporti ai tradizionali mezzi di comunicazione come servizi postali, stampa, radio e televisione, tutto in un'unica grande infrastruttura distribuita e decentrata che tutti conosciamo come rete Internet.

Questa infrastruttura ha permesso anche il rinnovamento di una delle più grandi invenzioni di tutti i tempi, il **telefono**, passando da una comunicazione tradizionale basata su reti a *commutazione di circuito* a una comunicazione basata su reti a *commutazione di pacchetto* tramite la tecnologia VoIP[2], grazie anche alla costante crescita della **telefonia cellulare**.

Nonostante i tentativi effettuati alla fine degli anni '90 di integrare Internet e telefonia cellulare (ricordiamo i dispositivi cellulari con supporto al protocollo WAP), la vera e propria unificazione arriva verso la fine del primo decennio del secolo attuale, con l'invenzione degli smartphone, provocando il progressivo abbandono dello standard GSM (rete cellulare di seconda generazione) in favore degli standard di terza generazione come UMTS. Una delle

[1] http://info.cern.ch/Proposal.html

[2] Voice over IP, cioè l'incapsulamento di una codifica numerica della voce in una serie di datagram IP

caratteristiche fondamentali di questi dispositivi mobili è data dalla possibilità di connettersi alla rete utilizzando due interfacce di rete diffenti: una Wi-Fi e una UMTS.

Parallelamente allo studio e alla diffusione di massa di queste nuove tecnologie, assistiamo ad un rapido aumento di interesse verso le problematiche legate al concetto di *mobilità*. Lo scopo è garantire ad un utente mobile una connessione alla rete che sia quanto più possibile stabile e performante nonostante i limiti fisici imposti dall'ambiente circostante, cercando di sfruttare a pieno le caratteristiche e le potenzialità fornite dai dispositivi mobili.

In questo lavoro di tesi verrà presentato un modo (ancora sperimentale) per utilizzare simultaneamente due generiche interfacce di rete, prendendo spunto dal modello *Always Best Packet Switching* (ABPS)[6] e ricorrendo ai principi della *virtualizzazione*. Il modello ABPS permette ad una applicazione di usare simultaneamente tutte le interfacce di rete, inviando e ricevendo i datagram IP attraverso l'interfaccia idonea, in base alle caratteristiche del datagram stesso e alla disponibilità della rete. L'organizzazione del presente documento è la seguente:

- **capitolo 1**: breve esposizione dei principi e di alcuni tra i più importanti paradigmi di *virtualizzazione*;
- **capitolo 2**: tipico scenario di utilizzo;
- **capitolo 3**: strumenti software impiegati per la realizzazione del progetto;
- **capitolo 4**: spiegazione più dettagliata degli obiettivi di questo lavoro;
- **capitoli 5** e **6**: architettura su cui si basa il progetto;
- **capitoli 7** e **8**: scelte progettuali, implementazione e testing del progetto;
- **capitolo 9**: problemi aperti e sviluppi futuri.

Elenco delle figure

1.1 Schema della virtualizzazione totale 3
1.2 Schema della virtualizzazione parziale 3
1.3 Schema della virtualizzazione di processo 5
1.4 Schema della paravirtualizzazione con hypervisor 6

2.1 Nodo mobile all'interno di area coperta da segnale Wi-Fi . . . 8
2.2 Nodo mobile all'interno di area coperta solo da segnale UMTS 8
2.3 umView: incapsulamento applicazione 9

5.1 Architettura preliminare con singolo proxy 16
5.2 Architettura preliminare con proxy dedicato 18
5.3 Architettura definitiva 20

6.1 Architettura software del sottomodulo umNETDIF 25

7.1 Creazione di un socket virtuale usando `umnetcurrent` 31
7.2 Creazione di un socket virtuale usando `umnetdif` 32
7.3 Sequenza di redirezione delle system call 38
7.4 Costruzione dell'array bidimensionale per `umnetdif_poll()` . 61

8.1 Rete locale per il testing di `umnetdif` 65

Elenco delle tabelle

7.1 Tabella riassuntiva delle operazioni sui flag 36

Indice

Introduzione **i**

Elenco delle figure **iii**

Elenco delle tabelle **v**

1 Virtualizzazione **1**

1.1 Virtualizzazione totale 2

1.2 Virtualizzazione parziale 2

1.3 System Call Virtual Machine 4

1.4 Virtualizzazione di processo 5

1.5 Paravirtualizzazione 6

2 Scenario **7**

3 Strumenti **11**

4 Obiettivi **13**

5 Architettura generale **15**

5.1 Singolo proxy 16

5.2 Proxy dedicato 17

5.3 Soluzione definitiva 19

6 Architettura di umNETDIF **23**

7 Implementazione **27**

7.1 `msocket()` e il comando `mstack` . . . 28
7.2 SockHT e gestione dei socket descriptor . . . 31
7.2.1 Socket virtuale . . . 33
7.3 Module Main Thread . . . 35
7.3.1 Inizializzazione e terminazione . . . 35
7.3.2 Flag dei socket . . . 36
7.4 Manipolazione delle system call . . . 37
7.4.1 `socket()` . . . 39
7.4.2 `bind()` . . . 40
7.4.3 `getsockopt()` . . . 42
7.4.4 `setsockopt()` . . . 44
7.4.5 `listen()` . . . 45
7.4.6 `accept()` . . . 46
7.4.7 `connect()` . . . 49
7.4.8 `getsockname()` e `getpeername()` . . . 51
7.4.9 `write()` e `send*()` . . . 52
7.4.10 `read()` e `recv*()` . . . 54
7.4.11 `close()` e `shutdown()` . . . 56
7.5 Umnetdif Utils . . . 57
7.6 Poll Thread . . . 58
7.6.1 Event subscription e system call bloccanti . . . 58
7.6.2 Wrapper per la chiamata a `poll()` . . . 60
7.6.3 Gestione delle richieste . . . 62
7.6.4 Monitoraggio dei socket . . . 62
7.6.5 Poll Utils . . . 63

8 Test **65**

8.1 Ambiente di testing . . . 65
8.2 Configurazione . . . 66
8.3 Test di connessione multipla . . . 68
8.3.1 Processo server interno a umview . . . 68

8.3.2 Processo client interno a umview 69
8.4 Supporto flag `SOCK_NONBLOCK` e `MSG_DONTWAIT` 71
8.5 Test di concorrenza 72
8.5.1 `connect()` test 72
8.5.2 `read()` test 73
8.6 Numero massimo di socket aperti 75

9 Sviluppi futuri **77**

Conclusioni **79**

Bibliografia **81**

Capitolo 1

Virtualizzazione

In ambito informatico, il termine **virtualizzazione** ha assunto nel tempo svariati significati tutti accomunati dal concetto di **livello di astrazione**. Un sistema è composto da vari livelli di astrazione separati tra di loro da **interfacce ben definite**. I livelli di astrazione permettono di ignorare o semplificare i dettagli implementativi dei livelli sottostanti, semplificando quindi la progettazione di componenti ai livelli superiori[12]. I dettagli di un hard disk, per esempio, che è suddiviso in **tracce** e **settori**, vengono semplificati dal sistema operativo in modo che il disco appaia ad una applicazione semplicemente come un insieme di **file**. Perciò un'applicazione può creare, scrivere e leggere file senza preoccuparsi dell'organizzazione interna dell'hard disk. I livelli d'astrazione sono organizzati in una gerarchia. In linea generale si può affermare che il sistema operativo è una astrazione dell'hardware sottostante e, attraverso la sua interfaccia, rende possibile l'utilizzo della macchina senza tener conto dei dettagli hardware. Il concetto di virtualizzazione può essere applicato non solo ai sottosistemi (come per esempio l'hard disk) ma, in maniera più ampia, all'intera macchina. Una macchina virtuale viene implementata aggiungendo uno strato software alla macchina reale per poter fornire una diversa architettura o un diverso formato binario. Ad esempio, un software di virtualizzazione installato su una piattaforma Linux con architettura AMD64 può fornire una macchina virtuale capace di avviare

applicazioni scritte per piattaforme Windows con architettura i386.

1.1 Virtualizzazione totale

Per virtualizzazione totale si intende emulazione completa di una architettura, quindi dell'intero set di istruzioni (ISA) di un processore (figura 1.1). Questo metodo permette di avviare sistemi operativi **guest** all'interno del sistema operativo **host** della macchina reale senza apportare modifiche. Viene quindi creata una **sandbox**, un'area sicura in cui poter sperimentare, compiere azioni anche potenzialmente pericolose per una macchina reale o simulare scenari reali. Uno degli aspetti negativi è rappresentato dalla perdita di prestazioni, in quanto l'emulazione di una operazione all'interno della macchina virtuale può richiedere piú operazioni da parte della macchina reale.

Tra i software di virtualizzazione totale più noti troviamo Oracle VirtualBox[1], QEMU[2]

1.2 Virtualizzazione parziale

Al contrario della virtualizzazione totale, la **virtualizzazione parziale** prevede la simulazione solo di alcune componenti hardware. È più semplice da implementare e non permette l'esecuzione di un intero sistema operativo guest ma soltanto l'esecuzione di una o più applicazioni (figura 1.2). Tra i vari scopi della virtualizzazione parziale troviamo l'estensione di alcune funzionalità del sistema sottostante, la coesistenza di applicazioni destinate a diverse piattaforme o architetture, backward compatibility e portabilità.

[1]virtualbox.org
[2]qemu.org

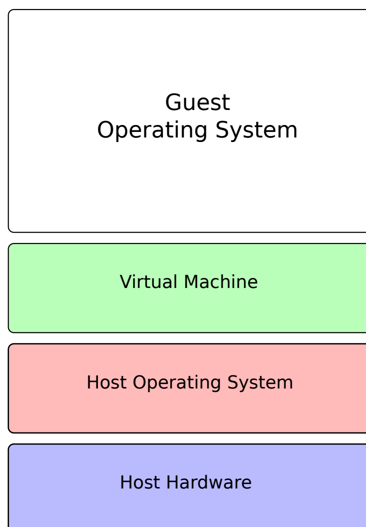

Figura 1.1: Schema della virtualizzazione totale

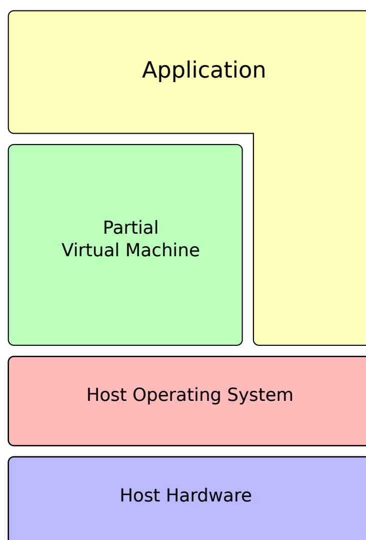

Figura 1.2: Schema della virtualizzazione parziale

1.3 System Call Virtual Machine

Il presente progetto di tesi si basa sull'utilizzo di **umView**, implementazione di una macchina virtuale del progetto ViewOS[3] del Dipartimento di Informatica dell'Università di Bologna. L'obiettivo di ViewOS è quello di forzare la *global view assumption*, principio secondo cui più processi condividono la stessa percezione del sistema operativo sottostante, fornendo la possibilità ad ogni processo di modificare la propria visione delle risorse e dell'ambiente d'esecuzione[5].

UmView realizza proprio questo concetto implementando a **livello utente** una *partial virtual machine* modulare in cui vengono virtualizzate solo le *system call*. Affinché sia possibile virtualizzarle, le system call vengono *intercettate* tramite `ptrace()`[4].

Ogni modulo per questa macchina virtuale fornisce la virtualizzazione per una famiglia di system call, perciò si parla di System Call Virtual Machine (SCVM). Ad esempio, le *socket call* vengono virtualizzate grazie al modulo *umnet*, mentre le system call relative alla gestione dei file system vengono virtualizzate dal modulo *umfuse*. Esiste anche una implementazione a **livello kernel**, detta *kmView*[5], che tende a migliorare le performance e che sfrutta la chiamate a `utrace()` per intercettare le system call.

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

[4] `man 2 ptrace`

[5] infatti spesso si parla di `xmview` per indicare genericamente le due versioni user e kernel mode

1.4 Virtualizzazione di processo

La **virtualizzazione di processo** consiste nel fornire un ambiente di esecuzione indipendente dalla piattaforma sottostante, astraendo i dettagli dell'architettura o del sistema operativo host.

Le **Process Virtual Machine** (PVM) sono dei normali processi all'interno del sistema host e supportano un solo processo target (figura 1.3), infatti ogni istanza di una PVM ha la stessa durata temporale del processo target. Uno degli scopi fondamentali di questo tipo di virtualizzazione è garantire la portabilità del software: l'esempio più famoso è sicuramente Java Virtual Machine.

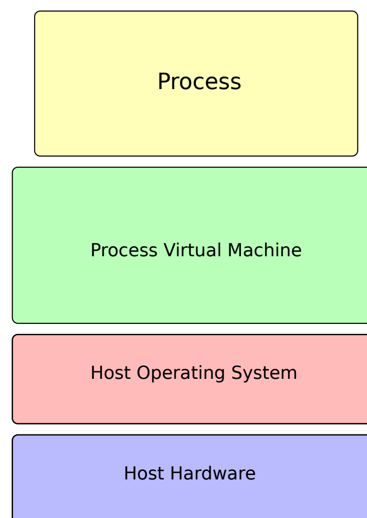

Figura 1.3: Schema della virtualizzazione di processo

1.5 Paravirtualizzazione

Con **paravirtualizzazione** si intende quel processo di condivisione delle risorse di una macchina reale tra varie istanze di macchine virtuali, gestito da un particolare software chiamato **hypervisor** o Virtual Machine Monitor (figura 1.4). Il software più famoso di questa famiglia è **XEN**[1].

I diversi sistemi operativi, detti **domini**, concorrono all'utilizzo delle risorse attraverso l'interfaccia **hypercall**, seguendo lo stesso principio di funzionamento delle system call di un comune sistema operativo.

La paravirtualizzazione è molto usata nei servizi di remote hosting, in cui vengono offerti servizi di Virtual Private Server.

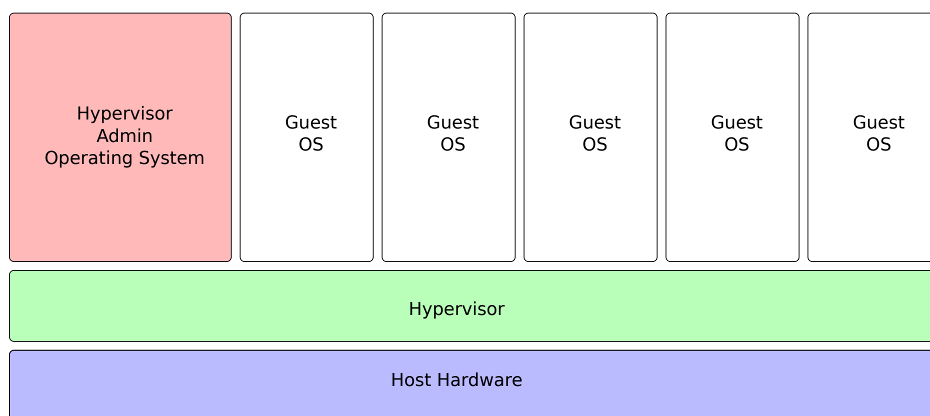

Figura 1.4: Schema della paravirtualizzazione con hypervisor

Capitolo 2

Scenario

Supponiamo di avere un nodo mobile (smartphone, tablet...) munito di due interfacce di rete:

- una interfaccia di rete wireless collegata ad esempio ad una WLAN
- una interfaccia di rete UMTS collegata ad una rete cellulare

Generalmente, in presenza di un Access Point wireless, si tende a sfruttare l'interfaccia wireless per ovvi motivi di carattere tecnico (una maggiore banda in upload/download, migliore qualità del segnale...) o di carattere economico (costo del traffico dati). D'altra parte, le caratteristiche tecniche degli standard IEEE 802.11 non permettono di avere un range del segnale superiore a qualche decina o centinaia di metri, range molto limitato se si considera la copertura territoriale del segnale cellulare.

Il seguente esempio mostra un tipico scenario: supponiamo di voler usare il nostro nodo mobile per aprire uno stream di dati duraturo nel tempo (ad esempio il download di un file molto grande) e di trovarci inizialmente in un'area effettivamente coperta da segnale wifi (figura 2.1).

Una connessione TCP resta aperta finché ci si muove all'interno dell'area, ma viene chiusa non appena si supera il range di copertura, provocando anche la chiusura del canale di comunicazione a livello più alto (figura 2.2).

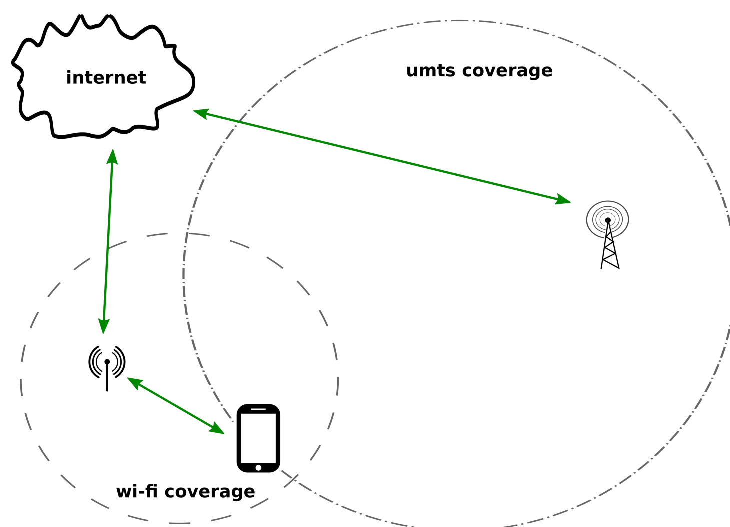

Figura 2.1: Nodo mobile all'interno di area coperta da segnale Wi-Fi

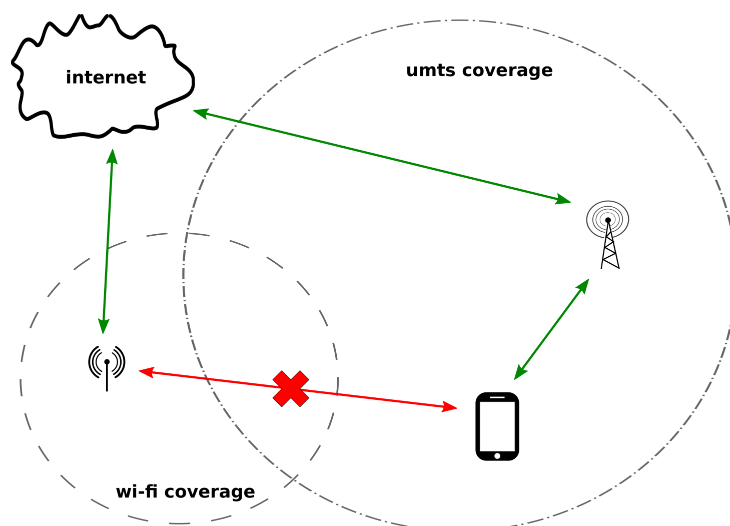

Figura 2.2: Nodo mobile all'interno di area coperta solo da segnale UMTS

A questo punto non resta che attivare l'interfaccia UMTS e ristabilire una nuova connessione a partire da un diverso indirizzo IP.

Il problema resta invariato anche nel caso di protocolli di livello applicativo che si basano su invio e ricezione di datagram UDP (protocolli VoIP), in quanto ogni interfaccia di rete possiede un proprio indirizzo IP, quindi lo switch da interfaccia wifi a umts rende irraggiungibile l'indirizzo IP della prima.

Una soluzione può essere quella di incapsulare l'applicazione all'interno di una macchina virtuale parziale (umView), con lo scopo di fornire un'unica interfaccia di rete virtuale. Un particolare sottomodulo di questa macchina virtuale si farà carico di nascondere le due interfacce reali, di dialogare con esse e di mantenere aperte le eventuali connessioni.

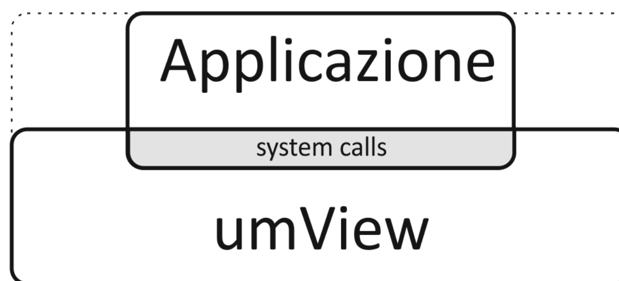

Figura 2.3: umView: incapsulamento applicazione

La figura 2.3 sintetizza due caratteristiche salienti di umView:

1. **incapsulamento**: la linea tratteggiata esterna indica che le applicazioni (possono essere una o più di una) sono processi figli di umView; questo significa che sono soggette alle regole impostate da unView stesso. Inoltre, trattandosi di processi figli, quando umView termina tutta la generarchia termina di conseguenza.

2. **virtualizzazione parziale**: umView non crea un ambiente completamente isolato da quello sottostante: limita la virtualizzazione alle sole system call.

Capitolo 3

Strumenti

Per lo sviluppo di questo progetto di tesi, scritto interamente in linguaggio C std ANSI C89, è stato utilizzato `umview` versione 0.8.2.1 oltre ai seguenti strumenti per il testing:

- **host locale** con sistema operativo Debian GNU/Linux Wheezy kernel 3.2.63-2 i686, CPU Intel Core i5 M460 2.53GHz;

- **host remoto** con sistema operativo Debian GNU/Linux Wheezy kernel 3.2.63-2 i686, CPU AMD Turion 64 ML-34;

- `uml-utilities` appartenente al software di virtualizzazione **UserModeLinux**[1] per la creazione di due interfacce ethernet virtuali `tap0` e `tap1`;

- `vde2`[7] per la creazione di una VLAN tra le due macchine;

- `netcat`[2] per instaurare connessioni e generare traffico tra le due macchine;

- `netstat` per il monitoraggio delle porte.

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

[2] `netcat-openbsd`: versione riscritta per OpenBSD

Capitolo 4

Obiettivi

L'obiettivo del progetto è quello di controllare il traffico di rete di una o più applicazioni al fine di utilizzare l'interfaccia di rete *più conveniente* in quel momento, senza generare interruzioni o frammentazioni della comunicazione. Bisogna sottolineare che, nel contesto del sistema, il comportamento appena descritto deve riguardare la singola applicazione, devono cioè essere definite delle politiche a livello di applicazione e non di dispositivo. Va inoltre puntualizzato che il terminine "*più conveniente*" non si riferisce semplicemente alla disponibilità o alle prestazioni[1] del dispositivo, ma coinvolge anche il tipo di applicazione in essere. Ad esempio, se dobbiamo eseguire il download di un file di grandi dimensioni è meglio scegliere l'interfaccia più veloce, mentre se stiamo utilizzando un'applicazione che realizza VoIP è meglio scegliere l'interfaccia più stabile, che garantisce cioè un flusso di dati costante. Possono inoltre essere definite delle politiche di gruppo che vengono applicate a prescindere dal tipo di applicazione. Per esempio ad ogni interfaccia può essere attribuita una priorità che dipende dal costo del collegamento ad essa associato.

Lo scenario descritto nel capitolo 2 rappresenta il punto dal quale siamo partiti per sviluppare il progetto. Attualmente il punto debole dei disposi-

[1] per esempio ampiezza di banda o potenza del segnale

tivi portatili[2] continua ad essere l'autonomia della batteria, spesso troppo limitata. Sappiamo inoltre che le stime fornite dai costruttori non prevedono l'utilizzo costante di connessioni wi-fi ed i valori sono centrati su applicazioni di uso comune quali browser oppure visualizzatori di testi e immagini.
A fronte di questo, l'obiettivo principale è quello di utilizzare degli strumenti privi di funzionalità inutili; ad esempio sarebbe semplice realizzare quello che stiamo proponendo mediante una macchina virtuale completa, ma l'impegno di risorse sarebbe tale da rendere i consumi non compatibili con l'autonomia dei dispositivi mobili.

Nel capitolo 5 vengono presentate diverse soluzioni che tengono conto di questo aspetto cercando di realizzare al meglio gli obiettivi che ci siamo posti.

[2] laptop, tablet, smartphone

Capitolo 5

Architettura generale

L'idea di base per realizzare quanto descritto negli obiettivi è quella di catturare alcune system call[1] chiamate dall'applicazione che si vuole controllare, al fine di modificarne il comportamento.

Se dal punto di vista concettuale questa descrizione risulta semplice e lineare, dal punto di vista pratico la definizione dell'architettura ha richiesto la valutazione di tre soluzioni. Nelle sezioni che seguono verranno spiegate le ragioni che ci hanno spinto a scartare le prime due soluzioni e ad accettare l'ultima come migliore compromesso funzionale.
Ricordiamo gli attori principali del sistema:

- **applicazione da controllare** è l'oggetto del lavoro; potrebbe essere una applicazione qualsiasi, ma l'architettura viene impiegata al meglio solo se l'applicazione scelta utilizza intensamente le comunicazioni di rete. Sarà un processo figlio di umView.

- **umView** gestisce la virtualizzazione delle system call (socket call nel nostro caso).

- **proxy** o sistema di controllo del traffico locale; nasconde le interfacce di rete reali all'applicazione che si sta controllando (e solo ad essa) in

[1] In particolare le system call del gruppo sys_socketcall

modo da veicolare il traffico sull'unica interfaccia virtuale messa a disposizione dal proxy stesso. La scelta dell'interfaccia fisica da utilizzare può essere fatta dal proxy oppure da un modulo specifico di umView.

- **tun0** è l'interfaccia virtuale utilizzata per modificare il flusso dati secondo logiche proprie di ciascuna architettura; l'installazione di TUN0 ha il duplice scopo di realizzare un collegamento virtuale con le applicazioni e di istanziare un descrittore reale.

5.1 Singolo proxy

Come si evince dalla figura 5.1 l'applicazione lanciata da umView vede come unica interfaccia di rete TUN0. Le interfacce reali vengono nascoste e non saranno in nessun modo raggiungibili dall'applicazione. In questo caso sarà il proxy a veicolare in traffico da TUN0 verso l'interfaccia reale più conveniente in quel momento. Il proxy dovrà inoltre controllare il traffico in ingresso in modo da inoltrare a TUN0 solo i pacchetti destinati all'applicazione.

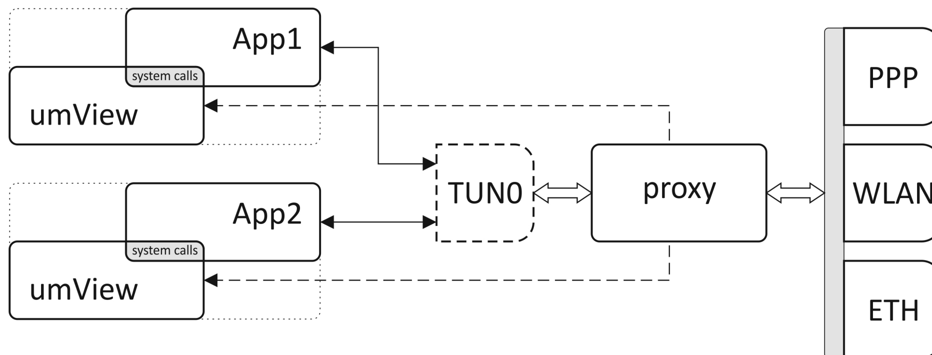

Figura 5.1: Architettura preliminare con singolo proxy

Anche se le figure 5.1 e 5.2 presentano alcune similitudini, la logica di funzionamento è totalmente diversa. Nella soluzione con singolo proxy si vuole implementare un modulo da collegare ad umView in grado di intercettare le socket call di tipo netlink e limitare la visione dell'applicazione alla sola interfaccia di rete TUN0. Invece, come vedremo nella sezione 5.2, la soluzione

con proxy dedicato userà prevalentemente socket INET.
L'analisi finale ha evidenziato alcuni punti di forza:

- architettura semplice: la disposizione lineare dei moduli riflette un flusso lineare di dati.
- intervento poco invasivo: le socket call da modificare sono un piccolo sottinsieme delle system call messe a disposizione dal kernel.

Sono stati però individuati anche alcuni punti deboli:

- possibile overload dovuto al passaggio di strutture di controllo tra umView e proxy; nell'ipotesi molto probabile che il sistema venga utilizzato su un dispositivo mobile, l'overload si tradurrebbe in un aumento dei consumi con conseguente calo di autonomia del dispositivo stesso.
- architettura non scalabile; per ogni applicazione bisogna attivare un'istanza di umView, mentre tutto il traffico continuerebbe ad essere veicolato attraverso l'unica interfaccia TUN0. A questo punto diventa difficile associare il flusso dati in ingresso all'applicazione corrispondente.
- la presenza di un singolo proxy rende difficile differenziare le politiche riferite alla qualità del servizio (QoS). In questo caso verebbe addottata un'unica politica per tutte le applicazioni in essere.

Le ultime due considerazioni ci hanno portato a scartare questa soluzione a favore dell'architettura descritta nella sezione 5.2 che prevede un proxy dedicato per ogni applicazione.

5.2 Proxy dedicato

La soluzione con proxy dedicato non è un semplice aggiornamento della versione con singolo proxy, tant'è che il principio di funzionamento è stato profondamente cambiato. In questo caso si vuole implementare un modulo da

collegare ad unView in grado di intercettare le chiamate di tipo sys_socketcall. L'applicazione controllata da umView sarà in grado di vedere tutte le interfacce di rete presenti sulla macchina, ma l'interazione con esse verrà mascherata dal modulo sopra citato.

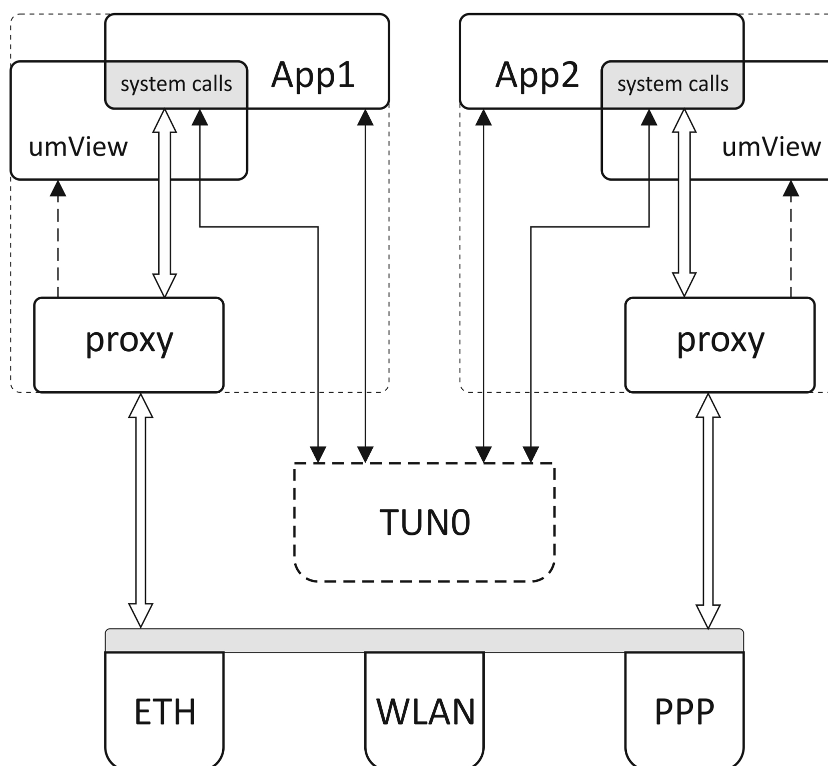

Figura 5.2: Architettura preliminare con proxy dedicato

Per semplicità in figura 5.2 sono state rappresentate due istanze di umView, ma lo stesso modello può essere esteso ad un maggior numero di istanze. Come evidenziato dalla linea tratteggiata, sia l'applicazione che il proxy sono processi figli di umView. Questo consente di realizzare con semplicità un canale di comunicazione interno[2] per lo scambio dei messaggi di controllo. Questa soluzione risolve il problema di possibile sovraccarico descritto nell'architettura con singolo proxy; inoltre umView potrebbe essere oppor-

[2]il canale può essere realizzato tramite pipe

tunamente modificato in modo da gestire il parametro relativo alla QoS da associare all'applicazione. Possiamo ipotizzare la seguente sintassi:

```
$umview -p umnet -qos x app1
$umview -p umnet -qos x app2
```

dove `qos` è la chiave che identifica il parametro e `x` è il valore associato ad una delle otto classi di QoS definite dall'ITU [9].
La presenza di un proxy per ogni istanza di umView risolve l'ultimo problema evidenziato nella precedente architettura relativo alla difficoltà di associare il flusso dati in ingresso all'applicazione corrispondente. In questo caso il problema non sussiste dal momento che umView, applicazione e proxy fanno parte dello stesso nucleo e quindi i riferimenti sono ben delineati.
Durante lo studio di fattibilità si è visto che le tecniche usate per mascherare le interfacce reali non sono risultate particolarmente efficaci sia da un punto di vista di risultati che da un punto di vista di stabilità di sistema. Venendo a mancare l'elemento centrale dell'architettura si è pensato di procedere con la soluzione descritta nella sezione 5.3 che ha come caratteristiche salienti quella di essere più lasca e modulare.

5.3 Soluzione definitiva

L'ultima versione dell'architettura ha visto una profonda ristrutturazione del proxy a cui è stato dato il nome *monitor* per caratterizzare meglio il ruolo che ricopre. Inoltre è stato aggiunto ad umView il modulo *umNETDIF* che ha il compito di gestire il traffico di rete dell'applicazione da lui controllata.
Si nota subito la diversa collocazione dei moduli: come da specifiche iniziali, umView e applicazione continuano a lavore in user space, mentre il *monitor*, avendo necessità di modificare le tabelle di routing e configurare le intefacce di rete, viene eseguito in kernel space.
Il *monitor* mette a disposizione un socket a cui *umNETDIF* si può collegare per ottenere informazioni sullo stato dei dispositivi di rete tutte le volte che sugli stessi si verificherà una variazione a livello network o datalink.

Per semplicità in figura 5.3 è stata rappresentata una sola istanza di

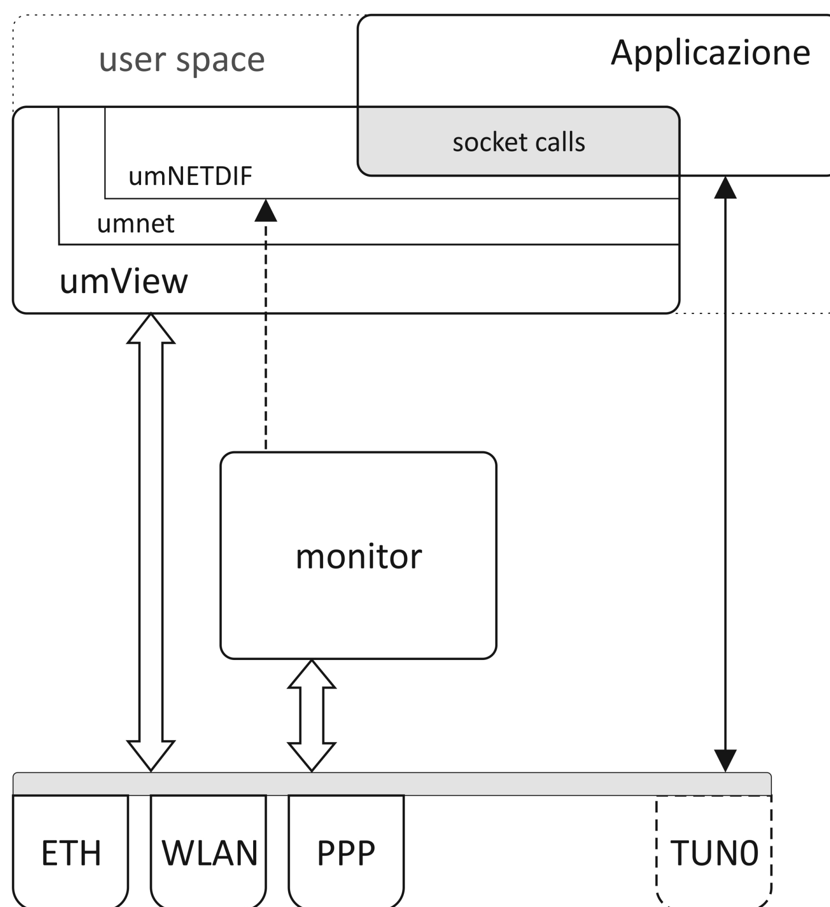

Figura 5.3: Architettura definitiva

umView. L'architettura consente la presenza contemporanea di più istanze di umView con relativa applicazione, mentre il *monitor* rimane comunque attivo su singola istanza.
La logica di instradamento dei pacchetti è la seguente:

- *monitor* e *umNETDIF* sono collegati e quindi *umNETDIF* conosce lo stato dei dispositivi di rete presenti sulla macchina. TUN0 è stata attivata e configurata come interfaccia di default.

- L'applicazione spedirà i pacchetti verso TUN0.

- umView intercetta l'operazione di scrittura e cede il controllo della system call a *umNETDIF* il quale provvederà ad incapsulare i pacchetti e a redirigerli verso l'interfaccia di rete più conveniente.
- I pacchetti in ingresso seguiranno il percorso inverso e l'applicazione avrà l'impressione di comunicare esclusivamente tramite TUN0.

Come già anticipato, l'architettura non è priva di difetti: citiamo quelli più evidenti:

- L'applicazione è in grado di vedere tutte le interfacce di rete, ma in assenza di indicazioni specifiche utilizzarà TUN0.
- Il *monitor* imposta delle politiche di instradamento globali che inevitabilmente riguarderanno anche le applicazioni non coinvolte in questo tipo di gestione.

Nonostante tutto, questa soluzione è sembrata essere la più conveniente poiché presenta il giusto equilibrio tra prestazioni e difficoltà di sviluppo.

Capitolo 6

Architettura di umNETDIF

Come precedentemente accennato, umview intercetta le system call di una certa famiglia e le redirige al relativo modulo. Ciò significa che una applicazione lanciata all'interno di umview vuoto, quindi senza aver caricato nessuno dei suoi moduli, funziona normalmente, altrimenti le system call vengono redirette al relativo modulo. Nel nostro caso, il modulo `umnet` si occupa soltanto di intercettare le system call di rete (socket call) e di redirigerle ad uno dei sottomoduli. In mancanza di sottomoduli per umnet (ma questo vale per qualsiasi altro modulo), il modulo restituisce -1 settando `errno` con codice d'errore `EINVAL`.

Il sottomodulo sviluppato in questo progetto prende il nome di `umNETDIF` (`umnet Double Interface`), che sta ad indicare l'utilizzo contemporaneo di due interfacce di rete, nonostante le due siano nascoste da una interfaccia virtuale[1].
A livello logico, le componenti principali di `umNETDIF` sono:

- **Module Main Thread** (*MMTh*): thread principale che si occupa di virtualizzare le system call intercettate da `umnet` utilizzando direttamente la *Socket API* del sistema operativo sottostante; è responsabile anche dell'inizializzazione e della terminazione del sottomodulo; il codice sorgente si trova in `umnetdif.c`;

[1] `tun0`, non ancora aggiunta

- **Poll Thread** (*PTh*): thread secondario che si occupa di monitorare lo stato dei socket descriptor affinché sia possibile effettuare operazioni di Input/Output; utilizza la chiamata a `poll()` del sistema operativo sottostante e comunica con *MMTh* attraverso due *pipe*; il codice sorgente si trova in `umnetdif.c`;

- **Socket Hash Table** (*SockHT*) è la struttura dati condivisa fra i due thread che gestisce i socket descriptor e mappa i socket virtuali (`generic_fd`) sui socket reali (`real_fd[0]` e `real_fd[1]`); la struttura è implementata come *hash table* nel file `sock_htab.c` e utilizza il wrapper `hsearch_rw`[2] per le funzioni fornite direttamente da `libc`[3];

- **Poll Utils**: wrapper per la chiamata a `poll()` e altre funzioni per la gestione di un array dinamico opaco di file descriptor; il codice sorgente si trova in `poll_utils.c`;

- **umNETDIF Utils**: utilities condivise fra i due thread; il codice sorgente si trova in `umnetdif_utils.c`.

[2] https://bitbucket.org/nshou/hsearch_rw/

[3] vedi `man 3 hsearch`

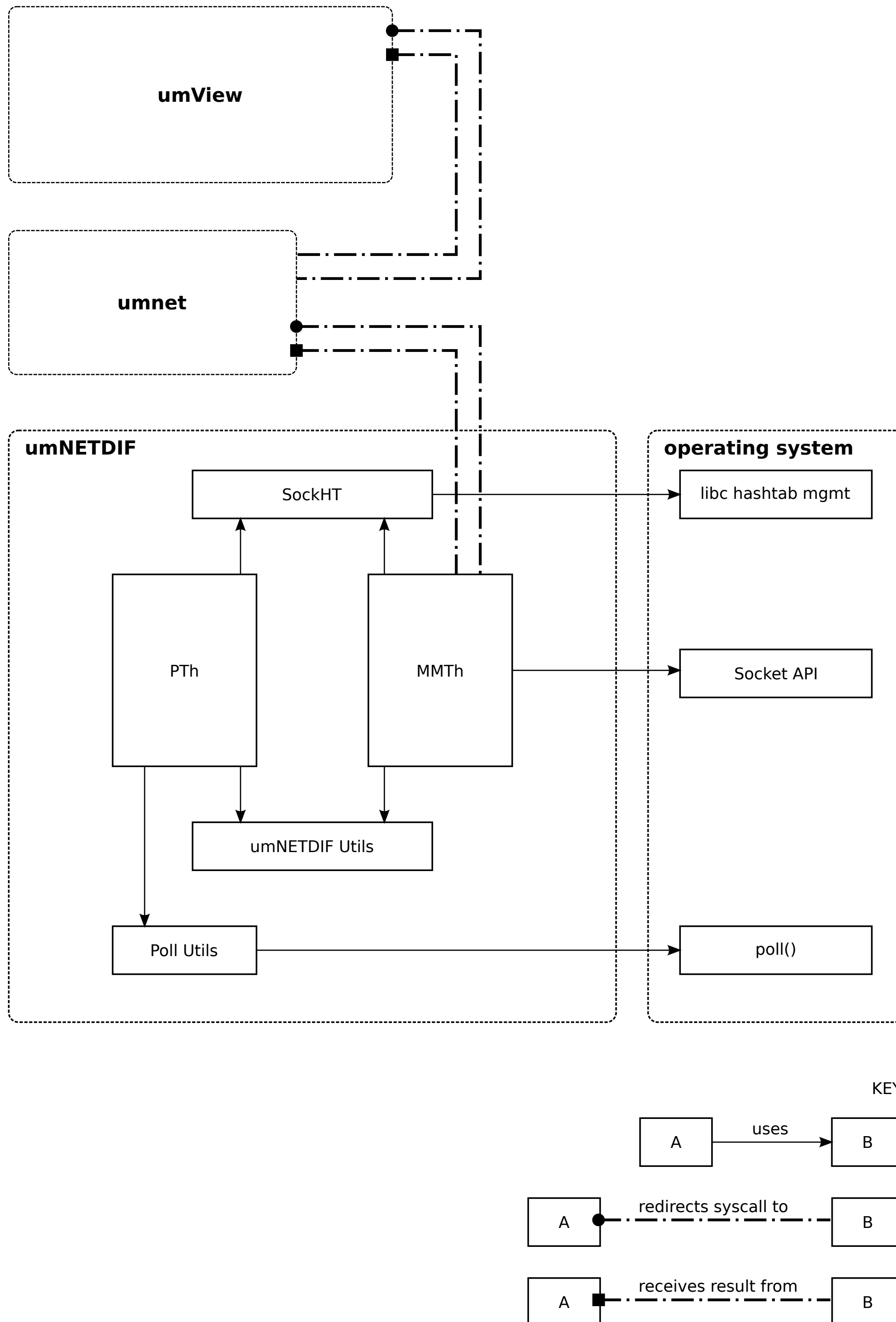

Figura 6.1: Architettura software del sottomodulo umNETDIF

Capitolo 7

Implementazione

Il sottomodulo, così come tutti i sottomoduli per `umview`, si presenta sotto forma di *libreria condivisa*[1] che può essere caricata a runtime tramite una estensione del comando `mount` (vedi sezione 7.1). Esistono sottomoduli di umnet già pronti, come `umnetnull` che intercetta la chiamata a una funzione e restituisce un errore, negando quindi l'accesso alla rete, oppure come `umnetcurrent` che semplicemente esegue la funzione chiamata. Quest'ultimo non apporta cambiamenti al funzionamento di un processo.

Quando si lavora con la Socket API, si ha normalmente a che fare con dei **file descriptor** (che identificano i socket aperti da una applicazione) e con **indirizzi** che, a seconda della famiglia, individuano vari tipi di end-point della comunicazione. In questo caso ci occupiamo solo di indirizzi appartenenti alla famiglia `AF_INET`, quindi indirizzi **IPv4**, tralasciando il supporto per indirizzi IPv6 o per altri protocolli di livello Transport.

[1]shared library, secondo la nomenclatura standard dei sistemi Unix-like

7.1 `msocket()` e il comando `mstack`

Una delle caratteristiche principali dei sistemi operativi UNIX-like è rappresentata dall'espressione *"tutto è un file"*[2], in quanto svariati tipi di risorse come documenti, directory, dispositivi di memorizzazione, periferiche, interfacce di rete vengono rappresentati come dei *file* all'interno del *filesystem*, quindi come stream di I/O. Anche i meccanismi di comunicazione hanno la loro controparte nel *filesystem* come gli stack di rete.

La Berkeley Socket API supporta diverse famiglie di protocolli di comunicazione ma non permette l'utilizzo di più stack di rete diversi per la stessa famiglia di protocolli, ad esempio più stack TCP/IP. All'interno del progetto View-OS è stata introdotta una estensione[4] alla Berkeley Socket API, definita da `msocket()` che supera questa limitazione:

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

Questa funzione, oltre a mantenere i parametri formali tipici di `socket()`, introduce un nuovo parametro `char *stack` in cui è possibile specificare il path di uno *special file* rappresentante un determinato stack di rete. I campi d'applicazione sono molteplici, infatti attraverso questa estensione è possibile definire permessi diversi per l'accesso alla rete utilizzando due diversi stack e assegnando i permessi tramite `chmod` o ACL, definire diversi livelli di sicurezza, oppure sperimentare nuove caratteristiche di accesso alla rete come Internet of Threads[3]. Fornisce inoltre *compatibilità all'indietro* per i programmi scritti usando la normale Socket API, attraverso la ridefinizione di `socket()` in termini di `msocket()`

```
int socket(int domain, int type, int protocol) {
    return msocket(NULL, domain, type, protocol);
}
```

Infatti, se il parametro `stack == NULL`, viene considerato lo stack di default per quel processo.

[2]traduzione dall'inglese "everything is a file"

Attraverso i sottomoduli di `umnet` è possibile definire degli stack di rete utilizzando il comando `mount`. Il seguente esempio mostra la definizione di due stack all'interno di umview, uno a partire dal sottomodulo `umnetnull` che nega l'accesso alla rete, l'altro a partire dal sottomodulo `umnetcurrent` che indirizza le chiamate allo stack di default, e l'esecuzione del comando `ip addr` utilizzando i diversi stack tramite il comando `mstack`:

```
$ um_add_service umnet
umnet init
$ mount -t umnetnull none /dev/net/null
$ mount -t umnetcurrent none /dev/net/current

[1] $ ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 16436 qdisc noqueue
        state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: tap0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc
        pfifo_fast state DOWN qlen 500
    link/ether 22:49:2b:d4:06:27 brd ff:ff:ff:ff:ff:ff
    inet 10.0.5.16/24 brd 10.0.5.255 scope global tap0
3: tap1: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc
        pfifo_fast state DOWN qlen 500
    link/ether 32:d5:0e:4c:04:a1 brd ff:ff:ff:ff:ff:ff
    inet 10.0.5.17/24 brd 10.0.5.255 scope global tap1
```

```
[2] $ mstack /dev/net/current
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 16436 qdisc noqueue
        state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: tap0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc
        pfifo_fast state DOWN qlen 500
    link/ether 22:49:2b:d4:06:27 brd ff:ff:ff:ff:ff:ff
    inet 10.0.5.16/24 brd 10.0.5.255 scope global tap0
3: tap1: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc
        pfifo_fast state DOWN qlen 500
    link/ether 32:d5:0e:4c:04:a1 brd ff:ff:ff:ff:ff:ff
    inet 10.0.5.17/24 brd 10.0.5.255 scope global tap1

[3] $ mstack /dev/net/null
Cannot open netlink socket: Address family not supported
        by protocol
```

`mstack` è un comando che permette di selezionare lo stack di rete. Questo comando ridefinisce lo stack di default e invoca il comando specificato tramite `exec()`. Nel comando `[1]` viene invocato `ip addr` senza usare `mstack`, nel comando `[2]` viene invocato usando lo stack `/dev/net/current` (infatti i due output coincidono), mentre nel comando `[3]` viene invocato usando lo stack `/dev/net/null`.

`mstack` viene inoltre impiegato nel file di configurazione `difrc` del sottomodulo `umnetdif`. In questo caso, per ragioni di testing, lo stack individuato dal sottomodulo viene reso stack di default.

7.2 SockHT e gestione dei socket descriptor

umview, in quanto macchina virtuale parziale, può trovarsi nella condizione di dover gestire file descriptor relativi a file o socket sia reali che virtuali allo stesso tempo. Nel core di umview è presente una hash table che si occupa di mappare i file descriptor virtuali sui file descriptor realmente aperti. Infatti umview utilizza delle *named pipe*[3] per realizzare dei socket virtuali. Questa hash table memorizza una entry *chiave-valore* dove il valore corrisponde al file descriptor del socket realmente creato, mentre la chiave corrisponde al valore del file descriptor della named pipe. Il valore di questa chiave è quello restituito al processo da chiamate come `open()`, `dup()` o `socket()`.

Supponiamo di aver caricato il modulo `umnet` e di aver selezionato il sottomodulo `umnetcurrent`. Supponiamo inoltre che un processo effettui una chiamata a `socket()`. Per prima cosa, umnet intercetta la chiamata a `socket()` e la redirige verso `umnetcurrent` che crea un socket il cui file descriptor sarà `sm_sd`. `umnetcurrent`, tramite `umnet`, restituisce a umview questo valore che diverrà il valore della entry (`entry1`) nella sua hash table. A questo punto, umview apre una named pipe, il cui file descriptor (`m_sd`) diverrà la chiave di `entry1`. Solo in questo momento, umview restituisce al processo il file descriptor della named pipe, cioè `m_sd`, che corrisponderà al file descriptor del socket virtuale.

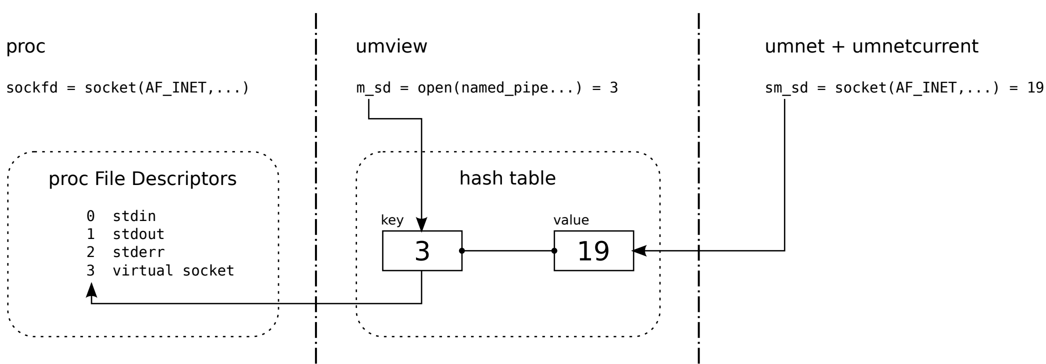

Figura 7.1: Creazione di un socket virtuale usando `umnetcurrent`

[3]conosciute anche come `FIFO`

Per quanto riguarda il nostro sottomodulo, i problemi non finiscono qui, infatti per utilizzare due interfacce di rete contemporaneamente si ha bisogno di mappare un socket virtuale su due socket reali (`real_fd[0]` e `real_fd[1]`), ognuno dei quali fa riferimento all'indirizzo IP di una interfaccia. Una domanda sorge spontanea: come identificare univocamente un socket virtuale senza preoccuparsi dei file descriptor reali? Affinchè la hash table di umview funzioni correttamente, è necessario fornirle un file descriptor valido. Non è possibile, ad esempio, fornire una codifica dei due valori secondo una funzione hash. La soluzione trovata consiste nel creare una seconda hash table, questa volta interna al sottomodulo e totalmente distinta da quella di umview, in cui la coppia chiave-valore è così memorizzata:

- il *valore* è un tipo di dato strutturato composto da due campi `real_fd[0]` e `real_fd[1]` contenenti i due file descriptor dei socket reali;

- la *chiave* `generic_fd` è un file descriptor relativo ad un file realmente aperto ma inutilizzato; a tal proposito viene aperto `/dev/null`.

La chiave della nostra coppia costituirà il *valore* della entry relativa alla hash table di umview. La scelta di aprire un generico file (nel nostro caso si è scelto arbitrariamente di aprire `/dev/null`) è dettata dalla necessità di fornire ad umview un file descriptor valido, e non un qualsiasi dato intero.

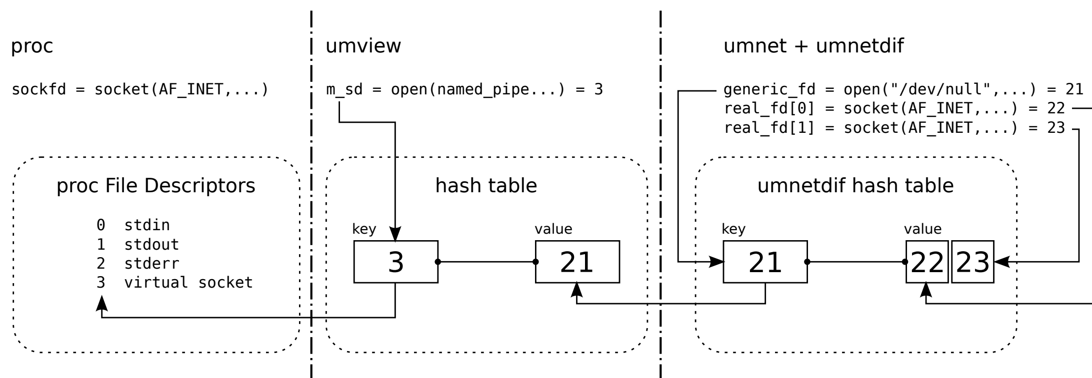

Figura 7.2: Creazione di un socket virtuale usando `umnetdif`

Questa metodo risolve eventuali conflitti tra i domini di file descriptor reali e virtuali, ma introduce una limitazione al numero massimo di socket aperti per processo (vedi sezione 8.6).

7.2.1 Socket virtuale

La struttura utilizzata per contenere le informazioni su un socket virtuale, e che costituisce il *valore* di una entry nella SockHT, è definita nel file `umnetdif_types.h` e corrisponde a:

```
struct sock_double_fd {
    int sfd[2];
    unsigned int flags;
    struct sockaddr *generic_local;
    struct sockaddr *generic_foreign;
    struct sockaddr *local[2];
    struct sockaddr *foreign[2];
    struct event_subs_arg *subscription;
};
```

Questa struttura, oltre a contenere i file descriptor dei socket reali, contiene un campo per i flag, dei puntatori di tipo `struct sockaddr*` e un puntatore di tipo `struct event_subs_arg*` che verrà definito nella sezione 7.6.1. Lo scopo dei puntatori di tipo `struct sockaddr*` è quello di tenere traccia delle operazioni di *binding* e di connessione sia del socket virtuale che dei due socket reali. La definizione dei campi è la seguente:

- `int sfd[2]`: array contenente i file descriptor dei due socket reali;

- `unsigned int flags`: campo contenente i flag associati al socket virtuale, definiti nella sezione 7.3.2;

- `struct sockaddr *generic_local`: punta a una struttura contenente indirizzo e porta *locali* del socket virtuale; questa struttura viene popolata in seguito ad una chiamata a `bind()` da parte del processo;

- `struct sockaddr *generic_foreign`: punta a una struttura contenente indirizzo e porta verso cui il socket virtuale è connesso; questa struttura viene popolata in seguito ad una chiamata a `connect()` da parte del processo;

- `struct sockaddr *local[2]`: array di puntatori, ognuno dei quali punta a una struttura contenente indirizzo e porta *locali* di ciascun socket reale;

- `struct sockaddr *foreign[2]`: array di puntatori, ognuno dei quali punta a una struttura contenente indirizzo e porta *remoti* verso cui ogni socket reale è connesso;

- `struct event_subs_arg *subscription`: puntatore a struttura contenente le informazioni circa l'evento atteso dal socket virtuale.

7.3 Module Main Thread

Il *Module Main Thread* (MMTh) è il thread principale del sottomodulo `umnetdif`. Il suo scopo è quello di *reimplementare* le system call della Socket API, nascondendo al processo in esecuzione la presenza delle due interfacce di rete. È inoltre responsabile della comunicazione con il secondario *Poll Thread* (PTh) e della gestione della SockHT.

7.3.1 Inizializzazione e terminazione

`umnetdif_init()` rappresenta l'entry point del sottomodulo. In primo luogo vengono rilevate le interfacce di rete presenti, successivamente viene assegnato un indirizzo IPv4 all'interfaccia virtuale (10.0.6.66 di default). Prima di generare e lanciare i due thread MMTh e PTh, si procede con l'inizializzazione di SockHT, del *mutex* per l'esecuzione concorrente dei thread e delle pipe di comunicazione tra MMTh e PTh.

Le pipe sono:

- `poll_command_wrpipe` e `poll_command_rdpipe`, rispettivamente write-end e read-end della Command Pipe, viene usata da MMTh per inviare il comando *Request for termination* a PTh, provocando la terminazione di PTh prima della terminazione dell'intero sottomodulo;

- `poll_data_wrpipe` e `poll_data_rdpipe`, rispettivamente write-end e read-end della Data Pipe, viene usata da MMTh per inviare a PTh una *Request for operation* (vedi sezione 7.6.1).

`umnetdif_fini()`, invece, si occupa semplicemente della terminazione del sottomodulo nel momento in cui viene chiuso umview, inviando il comando **Request for termination** a PTh. Inoltre, si occupa della distruzione di SockHT, del mutex e di PTh.

7.3.2 Flag dei socket

Ad ogni socket virtuale vengono associati dei *flag* che ne determinano il tipo, l'utilizzo e l'indirizzo su cui è stata chiamata la `bind()`. Le tre categorie e i rispettivi flag sono:

- `TYPE`: distingue il tipo di socket virtuale
  - `SOCKTYPE_DGRAM` per il tipo *datagram*
  - `SOCKTYPE_STREAM` per il tipo *stream*
- `USAGE`: indica su quali socket reali effettuare operazioni di I/O
  - `USE_FIRST` per usare solo il primo (`real_fd[0]`)
  - `USE_SECOND` per usare solo il secondo (`real_fd[1]`)
  - `USE_BOTH` per utilizzarli entrambi
- `BIND`: indica su quale indirizzo IP è stata chiamata `bind()`
  - `LOCAL_BOUND` binding effettuato su indirizzo `localhost 127.0.0.1`
  - `ANY_BOUND` binding effettuato su indirizzo dell'interfaccia di rete virtuale `10.0.6.66` oppure usando `INADDR_ANY`

Ogni categoria possiede le proprie macro per la manipolazione dei rispettivi flag, e le operazioni consentite sono `GET`, `SET`, `RESET`. Tutte le definizioni dei flag e delle macro si trovano nel file `umnetdif.h`.

	Macro		
cat	**SET op**	**GET op**	**RESET op**
`TYPE`	`SETTYPEFLG(sf,f)`	`GETTYPEFLG(sf)`	`RESETTYPEFLG(sf,f)`
`USAGE`	`SETUSGFLG(sf,f)`	`GETUSGFLG(sf)`	`RESETUSGFLG(sf,f)`
`BIND`	`SETBNDFLG(sf,f)`	`GETBNDFLG(sf)`	`RESETBNDFLG(sf,f)`

Tabella 7.1: Tabella riassuntiva delle operazioni sui flag

I flag della categoria `TYPE` vengono settati durante la creazione del socket virtuale, in base al parametro `type` della chiamata a `socket()`, e non cambiano durante la vita del socket virtuale. Per quanto riguarda la categoria

BIND, i flag vengono settati in fase di binding del socket virtuale, in base all'indirizzo IP fornito da bind(). Anche in questo caso, i flag sono costanti. Gli unici flag variabili sono quelli della categoria USAGE, che variano in base alla disponibilità della relativa interfaccia di rete.

7.4 Manipolazione delle system call

Come già detto, il core di umview intercetta una chiamata a system call tramite ptrace() e la redirige verso il modulo opportuno. Il modulo, a sua volta, redirige la chiamata al sottomodulo caricato che avrà il compito di reimplementare la system call. Avendo a disposizione due interfacce di rete, la reimplementazione prevede *due* chiamate alla system call in oggetto. I risultati delle due chiamate vengono poi analizzati per decidere quale risultato inviare al chiamante.

Generalmente, il codice di ogni reimplementazione prevede tre fasi:

1. *fase iniziale* in cui vengono dichiarate le variabili e le strutture necessarie; si procede ottenendo la struttura struct sock_double_fd relativa al socket virtuale in oggetto; nel caso non fosse possibile trovare il socket virtuale, il blocco di codice termina restituendo -1 e settando errno a EBADF; altrimenti si procede con l'inizializzazione delle variabili e delle strutture;

2. *fase centrale* in cui viene effettivamente richiamata la system call; quest'ultima viene chiamata una o due volte sia in base allo stato delle interfacce disponibili che al flag USAGE;

3. *fase finale* in cui vengono analizzati i risultati delle due chiamate in fase centrale per decidere quale risultato restituire al processo chiamante; il risultato non viene restituito direttamente ma deve passare attraverso il modulo superiore (umnet) per poi essere consegnato al processo tramite il core di umview;

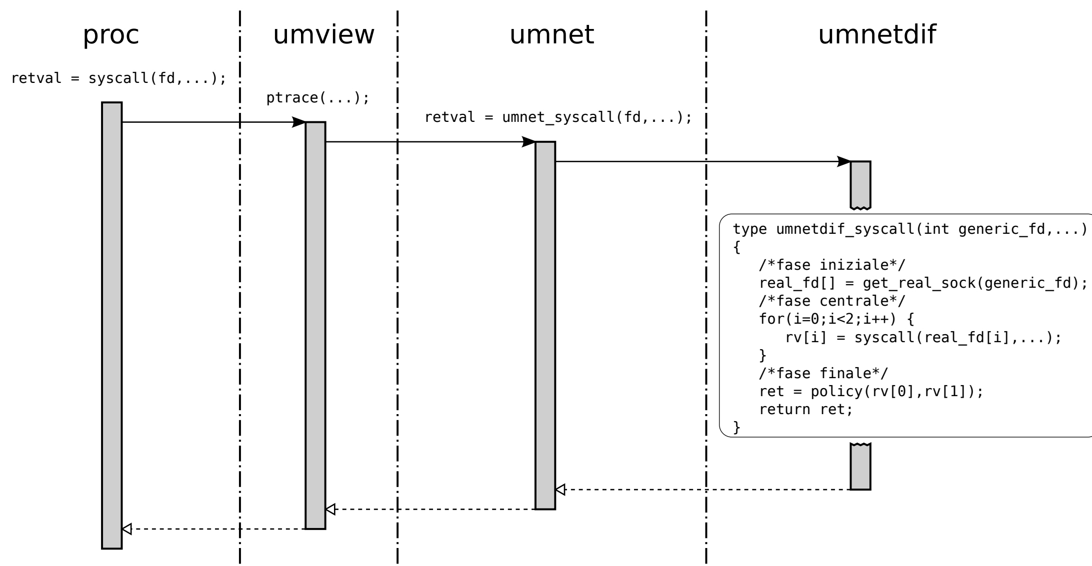

Figura 7.3: Sequenza di redirezione delle system call. I brani di codice, scritti in linguaggio pseudo-C, non rispecchiano la reale implementazione

7.4.1 socket()

La funzione (vedi sezione 7.1)

```
int umnetdif_msocket(int domain, int type, int protocol,
                     struct umnet *nethandle);
```

reimplementa la system call `socket()`.

fase iniziale:
viene allocata in memoria una struttura
`struct sock_double_fd`;

fase centrale:
viene prima aperto il file `/dev/null` ottenendo il file descriptor generico (`generic_fd`) (vedi sezione 7.2), poi vengono creati i due socket reali tramite una chiamata doppia a `socket()` ottenendo i file descriptor reali (`real_fd[]`);
infine vengono settati i flag `USAGE` che assumeranno il valore `USE_BOTH`[4];

fase finale:
in questa fase vengono prima settati i flag `TYPE` che dipendono dal parametro `int type` fornito dal processo, successivamente viene inserita in SockHT la entry *chiave-valore* dove

- chiave ← `generic_fd`;
- valore ← puntatore a `struct sock_double_fd` allocata in fase iniziale.

Infine viene restituito `generic_fd`.

[4] in seguito, questi flag verranno modificati in base allo stato delle interfacce

7.4.2 bind()

La funzione

```
int umnetdif_bind(int sockfd, const struct sockaddr *addr,
                  socklen_t addrlen);
```

reimplementa la system call `bind()`. Sono previsti tre funzionamenti diversi, in base all'indirizzo IP su cui il processo vuole effettuare la `bind()`:

1. `10.0.6.66` oppure `INADDR_ANY`

   fase iniziale:
   si cerca nella SockHT la struttura relativa al socket virtuale il cui file descriptor coincide con il parametro `int sockfd` fornito dal processo; se il processo ha specificato un *numero esatto* di porta su cui effettuare il binding del socket, si procede con la fase centrale, altrimenti si cerca di determinare un numero di porta **random** compreso tra 1024 e 65535 che sia libero per entrambe le interfacce di rete tramite la utility `port_is_free()` (vedi sezione 7.5), simulando il comportamento del sistema operativo nel caso venga fornito un numero di porta pari a 0;

   fase centrale:
   viene chiamata due volte `bind()`, una per ogni coppia *indirizzo:porta*; se almeno una delle due chiamate fallisce, anche `umnetdif_bind()` termina riportando lo stesso errore;

   fase finale:
   in questa fase viene settato per la prima volta il flag della categoria `BIND` col valore `ANY_BOUND`, mentre il flag di categoria `USAGE` mantiene il suo valore di default `USE_BOTH` settato da `umnetdif_msocket()`; inoltre vengono allocate le strutture puntate da `struct sockaddr *local[2]` che conterranno le informazioni sul binding dei due socket reali, da `struct sockaddr *generic_local` (vedi sezione 7.2.1) che conterrà le informazioni sul binding del socket virtuale; la funzione termina con successo.

2. `INADDR_LOOPBACK`

   fase iniziale:
   si cerca nella SockHT la struttura relativa al socket virtuale il cui file descriptor coincide con il parametro `int sockfd` fornito dal processo; viene poi determinata la porta su cui il processo vuole effettuare il binding (come nel caso precedente);

   fase centrale:
   viene chiamata la `bind()` solo una volta per default sul primo socket reale; se la chiamata fallisce, anche `umnetdif_bind()` termina riportando lo stesso errore;

   fase finale:
   il flag `BIND` viene settato con il valore `LOCAL_BOUND`, mentre il flag `USAGE` viene resettato al valore `USE_FIRST`, avendo scelto il primo socket reale come default; come per il caso precedente, vengono allocate le strutture puntate da `local[0]` (ma non `local[1]`), `generic_local` e la funzione termina con successo.

3. altri indirizzi IP: termina restituendo -1 settando `errno` con codice `EADDRNOTAVAIL`.

7.4.3 getsockopt()

La funzione

```
int umnetdif_getsockopt(int sockfd, int level, int optname,
                        void *optval, socklen_t *optlen);
```

reimplementa la system call `getsockopt()`. `socklen_t *optlen`, essendo un parametro passato per *valore-risultato*, deve puntare allo stesso valore dopo le due chiamate a `getsockopt()`.

fase iniziale:

viene allocato un array `socklen_t temp_optlen[2]` che servirà per memorizzare il risultato del parametro `optlen` per ognuna delle due chiamate a `getsockopt()`, e un array di puntatori `void *temp_optval[2]`; si cerca poi nella SockHT la struttura relativa al socket virtuale il cui file descriptor coincide con il parametro `int sockfd` fornito dal processo;

fase centrale:

in base al flag `USAGE` si sceglie fra tre strade: se abbiamo `USE_FIRST` oppure `USE_SECOND`, si procede chiamando una sola volta `getsockopt()` per l'unico socket reale utilizzabile: in questo caso, le fasi centrale e finale sono banali e la funzione termina restituendo il risultato della chiamata; se il flag è `USE_BOTH`, copiamo il valore puntato da `optlen` nei due elementi dell'array `temp_optlen` e allochiamo memoria per i due puntatori dell'array `temp_optval`. A questo punto si possono effettuare le due chiamate a `getsockopt()` in questa maniera:

```
rv[i] = getsockopt(sdfd->sfd[i], level, optname,
                   temp_optval[i], &temp_optlen[i]);
```

otteniamo così due valori puntati da `temp_optlen[0]` e `temp_optlen[1]` che potrebbero anche essere diversi tra loro;

fase finale:

in base alla politica scelta, la chiamata a `umnetdif_getsockopt()` termina con successo se e solo se

1. i valori restituiti dalle due chiamate sono entrambi diversi da -1;
2. `temp_optlen[0] == temp_optlen[1]`, quindi i due buffer coincidono in dimensione;
3. i due buffer puntati da `temp_optval` coincidono nel contenuto;

in questo caso, il parametro `optval` fornito dal processo viene aggiornato con uno dei due `temp_optval` (essendo uguali, non c'è bisogno di fare distinzioni) e il parametro `optlen` prende uno dei due `temp_optlen`; se anche una delle condizioni sopra elencate non è verificata, la chiamata termina restituendo un errore.

7.4.4 `setsockopt()`

La funzione

```
int umnetdif_setsockopt(int sockfd, int level, int optname,
                        const void *optval, socklen_t optlen);
```

reimplementa la system call `setsockopt()`.

fase iniziale:
si cerca in SockHT la struttura relativa al socket virtuale il cui file descriptor coincide con il parametro `int sockfd` fornito dal processo;

fase centrale:
in base al flag `USAGE` si sceglie fra tre strade: se abbiamo `USE_FIRST` oppure `USE_SECOND`, si procede chiamando una sola volta `setsockopt()` per l'unico socket reale utilizzabile: in questo caso, le fasi centrale e finale sono banali e la funzione termina restituendo il risultato della chiamata; se il flag è `USE_BOTH`, si effettuano le due chiamate a `setsockopt()` terminando con successo se e solo se le due chiamate terminano anch'esse con successo;

fase finale:
viene restituito il risultato.

7.4.5 `listen()`

La funzione

```
int umnetdif_listen(int sockfd, int backlog);
```

reimplementa la system call `listen()`.

fase iniziale:
si cerca in SockHT la struttura relativa al socket virtuale il cui file descriptor coincide con il parametro `int sockfd` fornito dal processo; affinché sia possibile mettere un socket in stato di `LISTENING`, si deve aver effettuato precedentemente il binding del socket virtuale su uno degli indirizzi permessi; osservando il puntatore `generic_local` è possibile determinare se sia stato effettuato il binding oppure no, infatti se il puntatore è diverso da `NULL` vuol dire che è stato effettuato, quindi si procede con la fase centrale, altrimenti viene richiamata `umnetdif_bind()` sul socket virtuale in oggetto.

fase centrale:
se il flag `BIND` è settato a `ANY_BOUND`, viene chiamata due volte `listen()`, una per ogni socket reale (terminando con successo se e solo se entrambe le chiamate terminano anch'esse con successo), altrimenti (flag settato a `LOCAL_BOUND`) viene chiamata `listen()` solo per il primo socket reale;

fase finale:
viene restituito il risultato.

7.4.6 accept()

La funzione

```
int umnetdif_accept(int sockfd, struct sockaddr *addr,
                    socklen_t *addrlen);
```

reimplementa la system call `accept()`.

Questa system call termina con successo restituendo il file descriptor di un nuovo socket, perciò una nuova entry deve essere aggiunta a SockHT.

Sono previsti due funzionamenti diversi, in base al flag `BIND` del socket virtuale:

1. `LOCAL_BOUND` quindi binding effettuato su `INADDR_LOOPBACK`

   fase iniziale:
   come per `umnetdif_msocket()`, viene aperto `/dev/null` per ottenere un file descriptor valido (indicato come `generic_retfd`) che rappresenterà il socket della connessione appena stabilita; in seguito viene allocata la struttura `struct sock_double_fd` da inserire in SockHT, che fa riferimento al nuovo socket;

   fase centrale:
   viene chiamata `accept()` una sola volta, e il risultato di questa chiamata costituirà il primo socket reale di `generic_retfd`;

   fase finale:
   dopo aver ottenuto il socket reale, vengono settati i flag del nuovo socket virtuale; in particolare

   - `TYPE` viene settato a `SOCKTYPE_STREAM` in quanto `accept()` viene chiamata solo per connessioni TCP;
   - `USAGE` viene settato a `USE_FIRST`;
   - `BIND` viene settato a `LOCAL_BOUND` dato che la connessione in ingresso proviene da `localhost`;

   vengono poi allocate le strutture `struct sockaddr_in` relative al nuovo socket virtuale; in particolare

- `struct sockaddr *generic_local` e
  `struct sockaddr *local[0]` erediteranno le stesse informazioni del socket virtuale fornito dal processo (parametro `sockfd` di `umnetdif_accept()`);
- `struct sockaddr *generic_foreign` e
  `struct sockaddr *foreign[0]` prenderanno le informazioni su *indirizzo:porta* dell'host remoto da cui proviene la connessione;

infine, il nuovo socket virtuale viene inserito in SockHT e il suo file descriptor restituito al processo;

2. `ANY_BOUND` quindi binding effettuato su `10.0.6.66` oppure `INADDR_ANY` in questa situazione è possibile restare in attesa di una o due connessioni sulle due interfacce di rete utilizzando `select()` per monitorare i due socket reali; nel caso di una sola connessione, le tre fasi coincidono al caso precedente (l'unica differenza è che il flag `BIND` del nuovo socket viene settato a `ANY_BOUND`), altrimenti

   fase iniziale:
   come per `umnetdif_msocket()`, viene aperto `/dev/null` per ottenere un file descriptor valido (indicato come `generic_retfd`) che rappresenterà il socket della connessione appena stabilita; in seguito viene allocata la struttura `struct sock_double_fd` da inserire in SockHT, che fa riferimento al nuovo socket;

   fase centrale:
   viene chiamata `accept()` due volte, una per ogni socket reale; le informazioni sulle due connessioni accettate, quindi *indirizzo:porta* da cui provengono le connessioni, vengono salvate nelle strutture puntate da `struct sockaddr *foreign[2]`; se una sola delle due chiamate ad `accept()` fallisce, la funzione termina restituendo errore; per procedere con la fase finale, gli indirizzi IP dei due host remoti **devono coincidere**, secondo la politica di default;

fase finale:

vengono settati i flag del nuovo socket virtuale; in particolare

- `TYPE` viene settato a `SOCKTYPE_STREAM` in quanto `accept()` viene chiamata solo per connessioni TCP;
- `USAGE` viene settato a `USE_BOTH`;
- `BIND` viene settato a `ANY_BOUND`;

viene allocata la struttura puntata da
`struct sockaddr *generic_foreign` in cui verranno salvate le informazioni su uno degli host remoti; per default si è scelto di salvare le informazioni del primo host, che sarà l'host remoto effettivamente visto dall'applicazione all'interno di umview; infine il nuovo socket virtuale viene aggiunto a SockHT e il suo file descriptor viene restituito all'applicazione.

7.4.7 connect()

La funzione

```
int umnetdif_connect(int sockfd, const struct sockaddr *addr,
                     socklen_t addrlen);
```

reimplementa la system call connect().

Se per il socket virtuale non è stata effettuata alcuna operazione di binding, quindi il puntatore struct sockaddr *generic_local è uguale a NULL, prima di procedere con le tre fasi viene richiamata umnetdif_bind() specificando l'indirizzo di default INADDR_ANY.

Sono previsti due funzionamenti diversi, in base al flag BIND del socket virtuale:

1. LOCAL_BOUND quindi binding effettuato su INADDR_LOOPBACK

   fase iniziale:
   si cerca in SockHT la struttura relativa al socket virtuale il cui file descriptor coincide con il parametro int sockfd fornito dal processo; i flag della categoria USAGE vengono resettati completamente;

   fase centrale:
   essendo il flag USAGE settato con LOCAL_BOUND, è possibile connettere questo socket solo a localhost, quindi viene chiamata connect() solo una volta per il primo socket reale, mentre il secondo viene scartato; se la connessione va a buon fine, USAGE flag viene settato con il valore USE_FIRST;

   fase finale:
   viene allocata la struttura puntata da struct sockaddr *generic_foreign e la struttura puntata da struct sockaddr *foreign[0] (tralasciando foreign[1]), in cui verranno salvate le informazioni circa l'host a cui il socket virtuale è connesso; infine il risultato viene restituito al processo;

2. `ANY_BOUND` quindi binding effettuato su `10.0.6.66` oppure `INADDR_ANY`

   fase iniziale:

   (stessa del caso precedente)

   fase centrale:

   viene fatta un'ulteriore distinzione, questa volta però sull'indirizzo IP di destinazione:

   - `INADDR_LOOPBACK`, questa situazione coincide al caso `LOCAL_BOUND`
   - altri indirizzi, in questo caso viene chiamata `connect()` due volte, una per ogni socket reale, e il flag `USAGE` viene settato con il valore `USE_BOTH`;

   fase finale:

   viene allocata la struttura puntata da `struct sockaddr *generic_foreign` salvando *indirizzo:porta* di destinazione fornita dal processo, inoltre vengono allocate le strutture puntate da `struct sockaddr *foreign[2]` con *indirizzo:porta* di destinazione di ogni singolo socket reale; infine viene restituito il risultato al processo; in questo caso basta che almeno una connessione sia stata instaurata per far terminare la funzione con successo.

7.4.8 `getsockname()` e `getpeername()`

Le funzioni

```
int umnetdif_getsockname(int sockfd, struct sockaddr *addr,
                         socklen_t *addrlen);
int umnetdif_getpeername(int sockfd, struct sockaddr *addr,
                         socklen_t *addrlen);
```

reimplementano, rispettivamente, le system call `getsockname()` e `getpeername()`.

Il funzionamento è semplicissimo e sfrutta il *caching* delle informazioni fornito da SockHT, infatti queste due funzioni rappresentano gli unici casi in cui non vengono richiamate le relative system call del sistema operativo sottostante.

Nel caso di `getsockname()` il processo riceve, tramite il parametro formale `addr`, l'indirizzo su cui è stata chiamata `bind()` per il socket virtuale. Con lo stesso meccanismo, nel caso di `getpeername()`, il processo riceve l'indirizzo a cui il socket virtuale è connesso.

7.4.9 write() e send*()

Le funzioni

```
ssize_t umnetdif_write(int fd, const void *buf,
                       size_t count);
ssize_t umnetdif_send(int sockfd, const void *buf,
                      size_t len, int flags);
ssize_t umnetdif_sendto(int sockfd, const void *buf,
                        size_t len, int flags,
                        const struct sockaddr *dest_addr,
                        socklen_t addrlen);
ssize_t umnetdif_sendmsg(int sockfd, const struct msghdr *msg,
                         int flags);
```

reimplementano, rispettivamente, le system call `write()`, `send()`, `sendto()` e `sendmsg()`[5].

Queste system call condividono la stessa struttura, perciò verrà esposta solo la reimplementazione di `sendto()`.

Questa classe di funzioni fa uso di `select()` per testare la possibilità di inviare dati attraverso le diverse interfacce di rete. Sono previsti due funzionamenti diversi, in base al flag `BIND` del socket virtuale:

1. `LOCAL_BOUND` quindi binding effettuato su `INADDR_LOOPBACK`

   fase iniziale:
   viene inserito il file descriptor del primo socket reale, tralasciando il secondo, nell'insieme dei file monitorati da `select()`; in questo caso `select()` può restituire soltanto 0 se l'invio è bloccante, oppure 1 per indicare che l'unico file monitorato è pronto per inviare dati;

   fase centrale:

[5] implementata solo parzialmente

- se il risultato di `select()` è uguale a 0, non è possibile inviare dati e la funzione termina restituendo -1;
- altrimenti viene richiamata `sendto()` specificando solo il file descriptor del primo socket reale;

fase finale:

viene restituito al processo il risultato di `sendto()`.

2. `ANY_BOUND` quindi binding effettuato su `10.0.6.66` oppure `INADDR_ANY`

fase iniziale:

in base al flag `USAGE`, vengono inseriti uno oppure entrambi i socket reali nell'insieme dei file monitorati da `select()`; in questo caso `select()` può restituire anche il valore 2, indicando la possibilità di inviare dati utilizzando entrambi i socket reali;

fase centrale:

- se il risultato di `select()` è uguale a 0, non è possibile inviare dati e la funzione termina restituendo -1;
- se il risultato di `select()` è uguale a 1, solo uno dei due socket reali è pronto per inviare dati, quindi `sendto()` verrà richiamata solo per quel socket;
- se il risultato di `select()` è uguale a 2, entrambi i socket reali sono pronti per inviare dati, quindi `sendto()` verrà richiamata due volte, inviando lo stesso buffer di dati attraverso i due socket reali;

fase finale:

solo nel caso in cui `select` abbia restituito 2, la politica di default prevede che la funzione termini con successo se *almeno una* delle due chiamate a `sendto()` sia terminata con successo; nei restanti casi, il valore restituito corrisponde a quello dell'unica chiamata a `sendto()`.

7.4.10 read() e recv*()

Le funzioni

```
ssize_t umnetdif_read(int fd, void *buf, size_t count);
ssize_t umnetdif_recv(int sockfd, void *buf,
                      size_t len, int flags);
ssize_t umnetdif_recvfrom(int sockfd, void *buf,
                          size_t len, int flags,
                          struct sockaddr *src_addr,
                          socklen_t *addrlen);
ssize_t umnetdif_recvmsg(int sockfd, struct msghdr *msg,
                         int flags);
```

reimplementano, rispettivamente, le system call `read()`, `recv()`, `recvfrom()` e `recvmsg()`[6]. Queste system call condividono la stessa struttura, perciò verrà esposta solo la reimplementazione di `recvfrom()`.

Anche questa classe di funzioni fa uso di `select()`, questa volta per testare la possibilità di ricevere dati dalle diverse interfacce di rete. Sono previsti due funzionamenti diversi, in base al flag `BIND` del socket virtuale:

1. `LOCAL_BOUND` quindi binding effettuato su `INADDR_LOOPBACK`

   fase iniziale:
   viene inserito il file descriptor del primo socket reale, tralasciando il secondo, nell'insieme dei file monitorati da `select()`; in questo caso `select()` può restituire soltanto 0 se la lettura è bloccante, oppure 1 per indicare che l'unico file monitorato è pronto per leggere dati;

   fase centrale:

   - se il risultato di `select()` è uguale a 0, non è possibile leggere dati e la funzione termina restituendo -1;

[6] implementata solo parzialmente

- altrimenti viene richiamata `recvfrom()` specificando solo il file descriptor del primo socket reale;

fase finale:

viene restituito al processo il risultato di `recvfrom()`.

2. `ANY_BOUND` quindi binding effettuato su `10.0.6.66` oppure `INADDR_ANY`

fase iniziale:

in base al flag `USAGE`, vengono inseriti uno oppure entrambi i socket reali nell'insieme dei file monitorati da `select()`; in questo caso `select()` può restituire anche il valore 2, indicando la possibilità di ricevere dati da entrambi i socket reali;

fase centrale:

- se il risultato di `select()` è uguale a 0, non è possibile ricevere dati e la funzione termina restituendo -1;
- se il risultato di `select()` è uguale a 1, solo uno dei due socket reali è pronto per ricevere dati, quindi `recvfrom()` verrà richiamata solo per quel socket;
- se il risultato di `select()` è uguale a 2, entrambi i socket reali sono pronti per ricevere dati, quindi `recvfrom()` verrà richiamata due volte;

fase finale:

solo nel caso in cui `select` abbia restituito 2, la politica di default è leggermente più complessa del solito, infatti la funzione termina con successo se:

- entrambe le chiamate a `recvfrom()` restituiscono un risultato maggiore o uguale a 0;
- i buffer contenenti i dati ricevuti coincidono;
- gli indirizzi IP di provenienza coincidono[7].

[7] questa condizione non viene considerata nel caso di `read()` e `recv()`

7.4.11 close() e shutdown()

Le funzioni

```
int umnetdif_close(int sockfd);
int umnetdif_shutdown(int sockfd, int how);
```

reimplementano, rispettivamente, le system call `close()` e `shutdown()`.

Il funzionamento è molto semplice, infatti `umnetdif_close()` si limita a chiudere i socket reali, chiudere il socket virtuale (chiudendo `/dev/null`) e liberare la memoria allocata per le strutture puntate da `struct sock_double_fd`. Infine viene rimossa la entry relativa al socket virtuale da SockHT.

Per quanto riguarda `umnetdif_shutdown()`, in base al flag `USAGE` viene richiamata `shutdown()` una o due volte chiudendo le connessioni eventualmente aperte.

7.5 Umnetdif Utils

Il file `umnetdif_utils.c` contiene alcune funzioni utili per lo sviluppo del sottomodulo:

```
char *int_to_string(int fd);
```

restituisce il file descriptor `fd` in formato stringa, utile per SockHT.

```
int get_virt_if_ipv4_addr(struct in_addr *inp);
```

scrive l'indirizzo IPv4 numerico dell'interfaccia di rete virtuale all'interno della struttura `struct in_addr *inp`. Restituisce -1 in caso di errore, oppure un valore maggiore di 0 in caso di successo. Al momento questa funzione usa un indirizzo IP hard-coded deciso a priori (10.0.6.66) (vedi capitolo 9).

```
void get_interface_ipv4_addr(char *ifname, char *ipv4_addr_ptr);
```

restituisce, tramite il parametro `char *ipv4_addr_ptr` passato per valore-risultato, l'indirizzo IPv4 dell'interfaccia di rete dal nome `ifname` nel formato *dot-decimal notation*, chiamando `ioctl()`.

```
uint16_t get_rand_port();
```

restituisce un numero di porta random nell'intervallo [1024, 65535].

```
int port_is_free(uint32_t addr, uint16_t port, int type);
```

usando `bind()`, determina se la porta `port` è libera per l'indirizzo IP `addr` distinguendo tra `SOCK_DGRAM` e `SOCK_STREAM` tramite il parametro `type`. Restituisce -1 in caso di errore, 0 in caso di porta (o coppia di porte) occupata, 1 in caso di porta (o coppia di porte) libera.

7.6 Poll Thread

Umview si presenta come processo single-thread, perciò è stata necessaria l'introduzione di un secondo thread (Poll Thread o PTh), basata sulla soluzione fornita da [8], il cui scopo è quello di monitorare gli eventi asincroni di I/O su un determinato insieme di file descriptor. Infatti, una chiamata a *system call bloccante*, ad esempio `read()`, in assenza di dati pronti per la lettura porta al blocco totale del core di umview (quindi anche dei processi virtualizzati al suo interno).

PTh sfrutta un loop infinito in cui vengono effettuate due operazioni principali, cioè *gestione delle richieste* e *monitoraggio* dei file descriptor. La terminazione del thread avviene in fase di terminazione del sottomodulo e consiste nell'invio di una *Request for termination* da parte di MMTh, sottoforma di comando a singolo carattere inviato attraverso `poll_command_wrpipe`.

7.6.1 Event subscription e system call bloccanti

La soluzione adottata durante lo sviluppo di umview si basa sulla funzione

```
static long event_subscribe(void (* cb)(),
                            void *arg,
                            int fd, int how);
```

che richiama la sua implementazione presente nel sottomodulo caricato (nel nostro caso `umnetdif`), dove

`cb` è l'indirizzo di una *callback* da richiamare nel caso in cui si verifichi l'evento atteso;

`arg` è l'indirizzo agli *argomenti* da passare come parametro a `cb`;

`fd` è il *file descriptor* da monitorare;

`how` è la *maschera degli eventi* da monitorare, codificata secondo l'analogo parametro della chiamata `poll()`[8].

[8] `man 2 poll`

Quando un processo chiama una system call bloccante, e prima che quest'ultima venga rediretta verso il modulo, umview richiama questa funzione per testare lo stato del file descriptor in questione, quindi per determinare se la system call sarà bloccante oppure no. Se la risposta è no, la system call viene invocata normalmente, altrimenti il processo chiamante viene messo in attesa. Umview risveglierà il processo quando il sottomodulo richiamerà la callback. La callback notifica al core di umview che il file descriptor in oggetto è pronto e che la system call può finalmente essere chiamata senza alcun rischio di blocco. Il compito di monitorare i file descriptor e di tenere traccia delle richieste è affidato a PTh.

La funzione

```
int umnetdif_event_subscribe(void (* cb)(),
                             void *arg,
                             int fd, int how);
```

viene richiamata da `umnet` attraverso `umnet_event_subscribe()`. Questa funzione richiama il wrapper `umnetdif_poll()` (vedi sezione 7.6.2) per testare immediatamente se il file descriptor è pronto per il dato evento parametrizzato da `how`. Se la risposta è affermativa, la funzione restituisce 1 ad umview e il processo può chiamare la system call senza bloccarsi; altrimenti, viene inviata una richiesta (*Request for operation*) a PTh per monitorare il socket.

In base al valore del parametro `cb`, il core di umview può richedere due operazioni:

- se `cb != NULL`, è richiesta la *registrazione* del file descriptor `fd`; PTh memorizzerà questa richiesta, aggiungerà il file descriptor nell'array `struct pollfd sock_set[]` (vedi sezione 7.6.5) e richiamerà `cb(arg)` quando si verificherà l'evento codificato in `how`;

- se `cb == NULL`, è richiesta la *deregistrazione* del file descriptor `fd`; PTh rimuoverà la richiesta e il file descriptor dall'array `struct pollfd sock_set[]`.

Le richieste vengono inviate a PTh tramite la pipe `poll_data_wrpipe` e vengono lette dal relativo read-end `poll_data_rdpipe` (vedi sezione 7.3.1). Il formato della richiesta è rappresentato dalla seguente struttura

```
struct event_subs_arg {
    voidfun cb;
    void *arg;
    int fd;
    int how;
};
```

in cui vengono copiati i parametri effettivi passati a
`umnetdif_event_subscribe()`.

Dopo l'invio della richiesta, MMTh si mette in attesa su una *condition variable*, mentre PTh procede alla registrazione/deregistrazione.

7.6.2 Wrapper per la chiamata a `poll()`

La system call `poll()` permette di monitorare un determinato insieme di file descriptor. Nel nostro caso abbiamo a che fare con dei socket virtuali, ognuno dei quali dipende da due socket reali individuati da altrettanti file descriptor. Non è possibile tenere sotto controllo in maniera *atomica* i due file descriptor dei socket reali, se non gestendo l'insieme dei file descriptor in maniera adeguata. La soluzione proposta in questo progetto prevede la creazione di un *wrapper*[9] per la system call `poll()` utile al monitoraggio simultaneo dei due file descriptor.

Il wrapper in questione è

```
int umnetdif_poll(struct pollfd *fds, nfds_t nfds, int timeout);
```

che mantiene gli stessi parametri formali di `poll()`.

A partire dall'array `fds` passato come parametro, in cui ogni elemento fa riferimento a un file descriptor di un socket virtuale, si costruisce un array

[9]codice sorgente presente nel file `poll_utils.c`

bidimensionale `struct pollfd my_fds[2][nfds]` in cui la prima e la seconda riga fanno riferimento, rispettivamente, al primo e al secondo socket reale di ogni socket virtuale, secondo lo schema di figura 7.4.

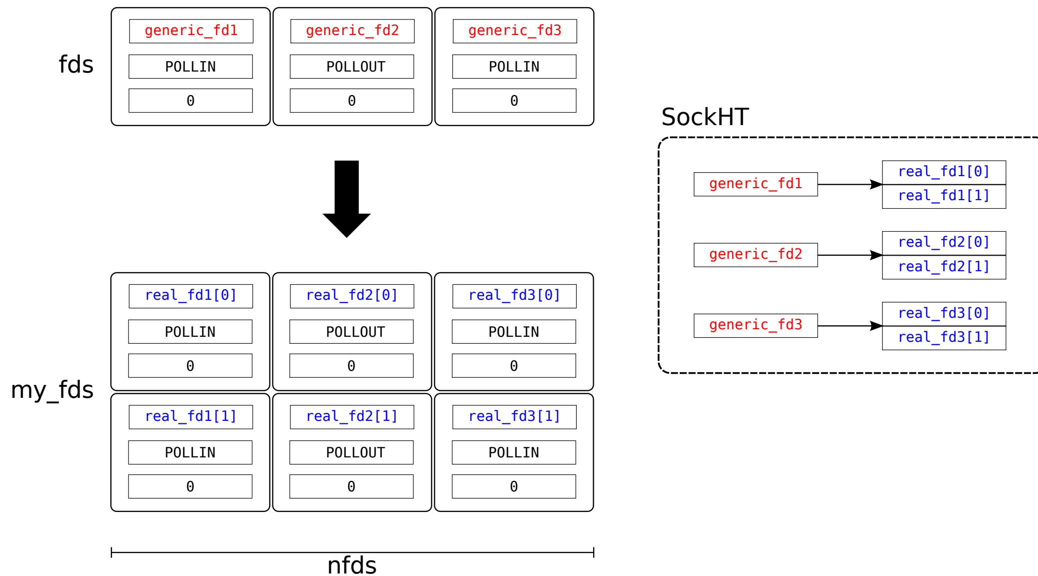

Figura 7.4: Costruzione dell'array bidimensionale per `umnetdif_poll()`

A questo punto viene invocata la `poll()` del sistema host per due volte, una per ogni riga dell'array, e si passa alla *proiezione dei risultati* secondo il seguente codice

```
for(i=0; i<nfds; i++) {
    /*projection of results*/
    fds[i].revents = my_fds[0][i].revents | my_fds[1][i].revents;

    /*count for matches*/
    if((fds[i].events & fds[i].revents) == fds[i].events)
        count++;
}
```

La scelta dell'operatore `bitwise OR` nella proiezione è giustificata dalla politica secondo cui il socket virtuale è pronto per I/O *se e solo se* lo è almeno uno dei due socket reali associati.

Il passo finale è rappresentato dal conteggio degli eventi verificati, il cui valore sarà restituito al chiamante.

7.6.3 Gestione delle richieste

Alla ricezione della richiesta, PTh acquisisce il lock su `poll_mutex`, ottiene la struttura con i parametri `event_req` tramite una `read()` sulla pipe e procede con l'operazione richiesta:

registrazione : si cerca il socket virtuale che corrisponde a `event_req.fd` in SockHT; se il socket è presente, al campo `struct event_subs_arg *subscription` viene assegnato il puntatore ad un'area di memoria in cui vengono copiati i parametri della richiesta; successivamente, viene aggiunto un elemento all'array `sock_set`[10] che indica il file descriptor da monitorare;

deregistrazione : si cerca il socket virtuale che corrisponde a `event_req.fd` in SockHT; se il socket è presente, l'area di memoria puntata da `struct event_subs_arg *subscription` viene rilasciata tramite una `free()`, il puntatore viene messo a `NULL` e dall'array `sock_set` viene rimosso l'elemento corrispondente a `event_req.fd`.

Dopo aver eseguito registrazione o deregistrazione, viene effettuata una `signal` sulla condition variable e il mutex viene rilasciato, permettendo ai due thread di riprendere la propria esecuzione.

7.6.4 Monitoraggio dei socket

Oltre a gestire le richieste, PTh si occupa di monitorare in background i file descriptor dei socket virtuali per cui è stata fatta richiesta. Non appena viene rilasciato il lock, PTh chiama `umnetdif_poll()` su tutto l'array `sock_set` e, in base al risultato ottenuto, passa in rassegna tutti gli elementi dell'array alla ricerca di eventi verificati.

[10] `struct pollfd` è definita in `man 2 poll`

Non appena si incontra un elemento dell'array in cui il campo `revents` fa match con il campo `how` della relativa richiesta, PTh invoca la callback `cb(arg)` notificando al core di umview la verifica dell'evento richiesto. Questa operazione va avanti finché l'array non è stato esaminato totalmente.

7.6.5 Poll Utils

Il file `poll_utils.c` contiene anche il codice di alcune funzioni utilizzate per inizializzare e manipolare l'array `sock_set` dei socket monitorati. Queste funzioni rendono l'array *dinamico* nella posizione degli elementi, nonostante l'array resti semplicemente statico nella dimensione. Il motivo che ha reso necessaria l'implementazione di queste funzioni è quello di aggiornare automaticamente il numero di elementi *effettivamente* inseriti che è diverso dal numero di elementi allocati in memoria. Il risultato è una struttura dati simile ad una *lista* implementata con un vettore utilizzabile da `poll()` senza la necessità di una conversione lista ↔ array. In questo paragrafo ci si riferisce alla *dimensione dell'array* come al numero di elementi effettivamente inseriti, mentre la dimensione in memoria dell'array verrà indicata come *dimensione massima*.

Le funzioni sono

```
nfds_t poll_array_init(struct pollfd *fds, int n);
nfds_t poll_array_add(struct pollfd *fds,
                      struct pollfd entry,
                      nfds_t dim);
nfds_t poll_array_remove(struct pollfd *fds,
                         int fd, nfds_t dim);
void poll_array_print(struct pollfd *fds, nfds_t dim);
```

dove

`poll_array_init()` inizializza l'array `fds` di *dimensione massima* `n` e restituisce 0 come *dimensione dell'array* iniziale;

poll_array_add() inserisce l'elemento `entry` in coda all'array `fds` (`dim` è la *dimensione dell'array* prima della chiamata a questa funzione) e restituisce la nuova dimensione, cioè `dim+1`;

poll_array_remove() rimuove l'elemento associato al file descriptor `fd` dall'array `fds`, la cui dimensione prima della chiamata è `dim`, e restituisce la nuova dimensione, cioè `dim-1`;

poll_array_print() funzione di debug che stampa (inviando a `stderr`) tutti gli elementi dell'array `fds` di dimensione `dim`.

Per un corretto funzionamento di questa struttura dati, è necessario notare che la dimensione dell'array **non deve** essere modificata manualmente, ma solo attraverso queste funzioni per evitare gravi effetti collaterali.

Capitolo 8

Test

8.1 Ambiente di testing

Tutti i test sono stati effettuati all'interno di una VLAN creata con Virtual Distributed Ethernet secondo la descrizione della figura 8.1.

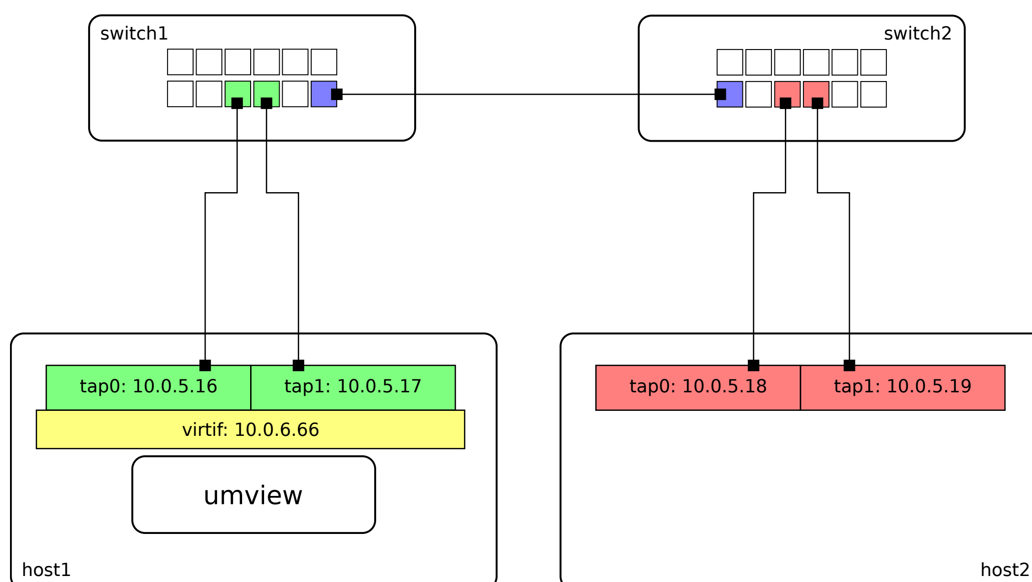

Figura 8.1: Rete locale per il testing di `umnetdif`

8.2 Configurazione

Per ragioni di testing sono state utilizzate due interfacce di rete virtuali `tap`. La creazione avviene tramite il comando `tunctl` fornito dal software `User-mode Linux`.

Su `host1`

```
# tunctl -u username // crea tap0
# tunctl -u username // crea tap1
# ifconfig tap0 10.0.5.16/24
# ifconfig tap0 up
# ifconfig tap1 10.0.5.17/24
# ifconfig tap1 up
```

Su `host2`

```
# tunctl -u username // crea tap0
# tunctl -u username // crea tap1
# ifconfig tap0 10.0.5.18/24
# ifconfig tap0 up
# ifconfig tap1 10.0.5.19/24
# ifconfig tap1 up
```

Successivamente, va configurata la rete locale tramite `vde`.

Su `host1`

```
$ vde_switch -s /tmp/switch1 -M /tmp/mgmt1 -d
# vde_plug2tap -s /tmp/switch1 tap0 -d
# vde_plug2tap -s /tmp/switch1 tap1 -d
```

Su `host2`

```
$ vde_switch -s /tmp/switch2 -M /tmp/mgmt2 -d
# vde_plug2tap -s /tmp/switch2 tap0 -d
# vde_plug2tap -s /tmp/switch2 tap1 -d
```

Su host1

```
$ dpipe vde_plug -s /tmp/switch1 = ssh username@host2
        vde_plug -s /tmp/switch2 &
```

Assicurarsi che tutti gli indirizzi della rete siano raggiungibili tra di loro eseguendo il comando `ping` verso ogni indirizzo.

Il file di configurazione di umview è `difrc`, in cui sono presenti i due comandi

```
um_add_service umnet
mount -t umnetdif -o tap0,tap1 none /dev/net/default
```

Il primo comando serve per caricare il modulo principale `umnet`, mentre il secondo serve per caricare il sottomodulo `umnetdif`. Il sottomodulo accetta come parametri i nomi delle due interfacce di rete utilizzate, tramite l'opzione `-o`. Montando il sottomodulo su `/dev/net/default` non ci sarà più bisogno di specificare lo stack di rete tramite il comando `mstack`.

Il comando

```
$ umview -f $DIFRC_DIR/difrc bash
```

avvia la shell `bash` all'interno di umview dopo aver specificato il file di configurazione `difrc`. Inserire al posto di `$DIFRC_DIR` la directory in cui si trova `difrc`.

8.3 Test di connessione multipla

8.3.1 Processo server interno a umview

Il seguente test dimostra la capacità del sottomodulo `umnetdif` di accettare due connessioni TCP per ogni socket virtuale, usando il client della directory `test/tcp/test1` e `netcat` come server. Il server deve essere avviato all'interno di umview su `host1`, mentre due istanze del client devono essere avviate da `host2` in parallelo:

host1 `$ umview -f $DIFRC_DIR/difrc bash`

host1 `$ nc -vvnl 10.0.6.66 35000`

host2 `$ ./test1_cl 10.0.5.18 40000 10.0.5.16 35000 &`
`./test1_cl 10.0.5.18 40001 10.0.5.17 35000 &`

Output di `netcat` **su** `host1`

```
$ nc -vvnl 10.0.6.66 35000
Listening on [10.0.6.66] (family 0, port 35000)
Connection from unknown port 35000 [tcp/*] accepted
(family 0, sport 0)
```

Output di `test1_cl host2`

```
$ ./test1_cl 10.0.5.18 40000 10.0.5.16 35000 &
        ./test1_cl 10.0.5.18 40001 10.0.5.17 35000 &
[1] 15883
[2] 15884
SOURCE 10.0.5.18:40001
DESTINATION 10.0.5.17:35000
SOURCE 10.0.5.18:40000
DESTINATION 10.0.5.16:35000
```

Output di `netstat` **su** `host1`

```
$ netstat -natp | grep umview
tcp 0 0 10.0.5.16:35000 10.0.5.18:40000 ESTABLISHED 20067/--umview
tcp 0 0 10.0.5.17:35000 10.0.5.18:40001 ESTABLISHED 20067/--umview
```

Output di `netstat` **su** `host2`

```
$ netstat -natp | grep test1_cl
tcp 0 0 10.0.5.18:40000 10.0.5.16:35000 ESTABLISHED 15883/test1_cl
tcp 0 0 10.0.5.18:40001 10.0.5.17:35000 ESTABLISHED 15884/test1_cl
```

8.3.2 Processo client interno a umview

Il seguente test dimostra la capacità del sottomodulo `umnetdif` di instaurare una connessione usando un socket virtuale, quindi sfruttando le due interfacce di rete `tap0` e `tap1`. In questo caso, il client avviato in umview è `netcat` mentre il server avviato su `host2` è `test1_srv`:

host2 `$ ./test1_srv 10.0.5.18 35000`

host1 `$ umview -f $DIFRC_DIR/difrc bash`

host1 `$ nc -vvn -s 10.0.6.66 10.0.5.18 35000`

Output di `test1_srv` **su** `host2`

```
$ ./test1_srv 10.0.5.18 35000
SOCKET BOUND TO 10.0.5.18:35000
Incoming connection from 10.0.5.16:16580
Incoming connection from 10.0.5.17:16580
```

Output di `netcat` `host1`

```
$ nc -vvn -s 10.0.6.66 10.0.5.18 35000
nc: can't set O_NONBLOCK - timeout not available:
```

```
        Invalid argument
Connection to 10.0.5.18 35000 port [tcp/*] succeeded!
```

Output di `netstat` **su** `host2`

```
$ netstat -natp | grep test1_srv
tcp 0 0 10.0.5.18:35000 0.0.0.0:* LISTEN 16351/test1_srv
tcp 0 0 10.0.5.18:35000 10.0.5.17:16580 ESTABLISHED 16351/test1_srv
tcp 0 0 10.0.5.18:35000 10.0.5.16:16580 ESTABLISHED 16351/test1_srv
```

Output di `netstat` **su** `host1`

```
$ netstat -natp | grep umview
tcp 0 0 10.0.5.16:16580 10.0.5.18:35000 ESTABLISHED 20067/--umview
tcp 0 0 10.0.5.17:16580 10.0.5.18:35000 ESTABLISHED 20067/--umview
```

8.4 Supporto flag SOCK_NONBLOCK e MSG_DONTWAIT

Il seguente test mostra come, allo stato attuale, `umview` non gestisca né il flag `SOCK_NONBLOCK` per rendere un socket non bloccante, né il flag `MSG_DONTWAIT` per rendere le chiamate `send*()/recv*()` non bloccanti.

Il programma si trova nella directory `test/tcp/test2` e, in base a un parametro da passare in fase di compilazione, genera tre varianti dello stesso:

`gcc -o test2 test2.c -DSNBLK`
: aggiunge il flag `SOCK_NONBLOCK` nella chiamata a `socket()` senza aggiungere il flag `MSG_DONTWAIT` nella chiamata a `recv()`;

`gcc -o test2 test2.c -DMSGDW`
: aggiunge il flag `MSG_DONTWAIT` nella chiamata a `recv()` senza aggiungere il flag `SOCK_NONBLOCK` nella chiamata a `socket()`;

`gcc -o test2 test2.c -DBOTHFLG`
: aggiunge entrambi i flag.

Il test può essere effettuato avviando il programma in umview su `host1` e avviando un server `netcat` su `host2`:

host2 `$ nc -vvnl 10.0.5.18 40000`

host1 `$ umview -f $DIFRC_DIR/difrc bash`

host1 `$ ./test2 10.0.6.66 35000 10.0.5.18 40000`

Nei casi `SNBLK` e `BOTHFLG` ci si aspetta la terminazione immediata del programma test con errore

```
connect: Operation now in progress
```

mentre nel caso di `MSGDW` ci si aspetta la terminazione immediata con errore

```
recv: Resource temporarily unavailable
```

Il risultato però non è quello atteso, infatti in tutti i casi nessuna delle chiamate `connect()` o `recv()` termina e umview richiama ugualmente `event_subscribe()`, indicando una mancata gestione dei due flag.

8.5 Test di concorrenza

Con i due test seguenti si cerca di studiare il comportamento di un processo multithread all'interno di umview, in cui ogni thread effettua chiamate bloccanti.

8.5.1 `connect()` test

Il processo è composto da due thread che, in maniera indipendente, tentano di instaurare una connessione verso due diversi indirizzi IP. Se i due thread vengono lanciati nello stesso momento[1], l'ordine con cui essi chiamano `connect()` dipende da come vengono schedulati dal sistema operativo.

Lo scenario è il seguente:

- `thread1`, che verrà lanciato per primo, intende stabilire una connessione con `10.0.5.18:40000`;

- `thread2` intende stabilire una connessione con `10.0.5.19:50000`;

- `iptables` blocca tutti i segmenti TCP in *uscita* verso `10.0.5.18` grazie al comando `iptables -I OUTPUT -p tcp -d 10.0.5.18 -j DROP`, rendendo l'host di destinazione inesistente o irraggiungibile;

Con questa regola per `iptables`, solo il secondo thread ha la possibilità di instaurare una connessione, mentre il primo dovrà attendere lo scadere del timeout di connessione prima di restituire l'errore `Connection timed out`. Essendo i thread concorrenti, il sistema operativo non attenderà lo scadere del timeout, perciò il secondo thread stabilirà la connessione prima dello scadere del timeout. Anche lanciando i due thread in maniera sequenziale, ad esempio fornendo un intervallo di tempo dell'ordine di 100000 ms[2] tra le due chiamate a `pthread_create()`, il secondo thread riuscirà ugualmente a stabilire la connessione prima dello scadere del timeout.

[1] ci sarà comunque un brevissimo intervallo di tempo tra un lancio e un altro

[2] tempo sufficientemente lungo

In umview la situazione sembra diversa, infatti in questo test le chiamate a `connect()` risultano essere *serializzate*. Il risultato è che `thread2` deve attendere lo scadere del timeout di `thread1` prima di poter stabilire la connessione. L'unica eccezione si ha quando i due thread vengono lanciati quasi contemporaneamente, ma `thread2` riesce a terminare tutte le operazioni prima che `thread1` chiami `connect()`.

Il test si trova nella directory `test/tcp/test3` e la sintassi è

```
Usage: ./test3 <local_ip> <local_port>
               <foreign_ip1> <foreign_port1>
               <foreign_ip2> <foreign_port2>
               <time interval millisec>
```

I passaggi per la riproduzione del test sono:

host2 in due diversi terminali

```
tty1 $ nc -vvnl 10.0.5.18 40000
tty2 $ nc -vvnl 10.0.5.19 50000
```

host1 `# iptables -I OUTPUT -p tcp -d 10.0.5.18 -j DROP`

host1 `$ umview -f $DIFRC_DIR/difrc bash`

host1 `$ ./test3 10.0.6.66 35000 10.0.5.18 40000 10.0.5.19 50000 100000`

8.5.2 `read()` test

Anche in questo caso il processo è composto da due thread. Ogni thread instaura una connessione verso un determinato host e resta in attesa di dati effettuando una chiamata a `read()`[3]. Mentre per il secondo thread ci sono dati già pronti per la lettura, per il primo i dati non sono presenti e questo

[3] stessa cosa per `recv*()`

costringe il primo thread a rimanere in attesa. Nel caso in cui le chiamate a `read()` fossero serializzate, questo scenario porterebbe a un *deadlock*.

A differenza del test precedente (vedi sezione 8.5.1) non vengono notati comportamenti anomali e il risultato è che le due chiamate concorrenti a `read()` non vengono serializzate.

Questo risultato è in netto contrasto rispetto al test precedente, quindi non tutte le system call bloccanti vengono serializzate in quanto il problema riguarda solo la system call `connect()`.

Il test si trova nella directory `test/tcp/test4` e la sintassi è

```
Usage: ./test4 <local_ip> <local_port>
               <foreign_ip1> <foreign_port1>
               <foreign_ip2> <foreign_port2>
               <timeout millisec>
```

Così come per il test precedente, è possibile inserire un intervallo di tempo tra il lancio di un thread e l'altro.
I passaggi per la riproduzione del test sono:

host2 in due diversi terminali

```
tty1 $ nc -vvnl 10.0.5.18 40000
tty2 $ nc -vvnl 10.0.5.19 50000
```

Per fornire dei dati al `thread2` prima che venga lanciato, è sufficiente inserire una stringa di testo in `tty2` come input per `netcat`;

host1 `$ umview -f $DIFRC_DIR/difrc bash`

host1 `$ ./test4 10.0.6.66 35000 10.0.5.18 40000 10.0.5.19 50000 100000`

8.6 Numero massimo di socket aperti

Un processo all'interno di umview, utilizzando il sottomodulo `umnetdif`, non può aprire più di 205 socket, rispetto ai 1021[4] di un processo esterno. Infatti per ogni socket virtuale, umview impiega (vedi sezione 7.2)

- due socket reali
- una named pipe aperta due volte, una in lettura e una in scrittura
- un file (`/dev/null`)

per un totale di 5 file descriptor occupati. Il test è presente nella directory `test/sock`.

[4]tre file descriptor sono riservati per `stdin`, `stdout`, `stderr`

Capitolo 9

Sviluppi futuri

Un passo fondamentale per il futuro di questo progetto è sicuramente il porting del sottomodulo `umnetdif` su architettura Android, in modo da proseguire ulteriormente il lavoro di porting già avviato dell'intera macchina virtuale `umview`[2, 11].

Per lo sviluppo di questo progetto è stato necessario effettuare delle semplificazioni, prima fra tutte quella relativa allo stato delle interfacce di rete. Infatti si assume che le interfacce siano sempre attive, mentre questo non è vero nella realtà. In questo senso è necessario integrare il presente progetto con il `monitor`[10] discusso nella sezione 5.3, e cercare una soluzione utile al ripristino delle connessioni di cui i socket reali ne rappresentano l'end-point.

Il progetto, inoltre, non è *scalabile* sul numero di interfacce di rete, quindi un ulteriore sviluppo sarebbe il supporto per un numero arbitrario di interfacce, eventualmente modificando la denominazione da `umnet Double Interface` a `umnet Multiple Interface` per una questione di chiarezza.

Ulteriori semplificazioni riguardano il mancato supporto per i socket `netlink` e l'aver adottato esclusivamente IPv4, mentre sarebbe auspicabile il supporto a IPv6.

A partire dai risultati dei test presentati nel capitolo 8, in particolare i test di concorrenza (sezione 8.5) e la gestione dei *non-blocking flag* (sezione 8.4), un possibile sviluppo futuro consiste nell'apportare modifiche al codice

di umview per far fronte alle due problematiche, cioè gestire più system call bloccanti in uno scenario multithread senza che esse vengano serializzate e fornire il supporto per i flag SOCK_NONBLOCK e MSG_DONTWAIT.

Scendendo nel particolare dell'implementazione, è necessario portare avanti la scrittura delle system call sendmsg() e recvmsg(), creare effettivamente l'interfaccia virtuale che nasconde quelle reali e fornire supporto per le chiamate a ioctl() relative a tale interfaccia.

Altre problematiche interessanti per futuri sviluppi sono legate alla *event subscription* e alle system call bloccanti, come la costruzione di un elenco di *callback* utile a distinguere le chiamate bloccanti.

Conclusioni

Il lavoro svolto ha dimostrato come sia possibile integrare in `umview` il principio alla base di ABPS, fornendo un supporto a livello utente.

Nonostante lo sviluppo di umview sia ancora in corso, questo progetto mette in luce ancora una volta le innumerevoli potenzialità del progetto View-OS e, più in generale, della virtualizzazione come soluzione alle problematiche legate alla mobilità.

Si è partiti discutendo sulle possibili architetture finali del progetto, scegliendo la creazione di un sottomodulo per `umview` affiancato da un *monitor* per le interfacce di rete. In seguito sono state affrontate alcune problematiche interne al sistema operativo, come la creazione e gestione di un socket virtuale attraverso l'utilizzo di una *hash table* (prendendo spunto proprio dalla soluzione adottata da `umview`) e la creazione di un modulo software sotto forma di *libreria dinamica* a doppio thread.

L'adozione di una soluzione *multi-thread* è stata necessaria per una corretta gestione delle system call bloccanti, quindi degli eventi di I/O, in quanto `umview` si presenta come processo *single-thread*. Proprio per questo, è stato necessario modificare il comportamento della system call `poll()` affinché potesse monitorare un insieme più complesso di socket.

Procedendo con lo sviluppo, sono state virtualizzate le system call di rete appartenenti alla Berkeley Socket API in modo da renderle compatibili con il socket virtuale, aggiungendo uno strato di virtualizzazione al livello *Transport* del modello ISO/OSI.

Uno dei limiti attuali, indicato come sviluppo futuro, è rappresentato dal

fatto che il progetto non è scalabile sul numero di interfacce di rete, infatti allo stato attuale è possibile gestire non più di due interfacce. Nonostante ciò, i test effettuati sul software dimostrano chiaramente come l'obiettivo principale sia stato raggiunto.

Bibliografia

[1] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, October 2003.

[2] Davide Berardi. Porting della macchina virtuale umView su sistema operativo Android ARM. Tesi di Laurea in Informatica, Università di Bologna, 2013.

[3] Renzo Davoli. Internet of Threads. Communication at the Conferenza Garr 2011 (in Italian), 2011.

[4] Renzo Davoli and Michael Goldweber. Msocket: Multiple Stack Support for the Berkeley Socket API. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 588–593, New York, NY, USA, 2012. ACM.

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

[6] Vittorio Ghini, Giorgia Lodi, and Fabio Panzieri. Always Best Packet Switching: the Mobile VoIP Case Study. *Journal of communications*, 4(9):700–713, 2009.

[7] Michael Goldweber and Renzo Davoli. VDE: An Emulation Environment for Supporting Computer Networking Courses. *SIGCSE Bull.*, 40(3):138–142, June 2008.

[8] Paolo Perfetti. Esperimenti di integrazione ViewOS-Rump: lo stack di rete NetBSD. Tesi di Laurea in Informatica Magistrale, Università di Bologna, 2013.

[9] Itu T. Recommendation. Y.1541: Network Performance Objectives for IP-Based Services. Technical report, International Telecommunication Union, 2003.

[10] Carlo Rimondi. Incapsulamento di applicazioni per il controllo del flusso dati: monitor. Tesi di Laurea in Informatica, Università di Bologna, 2014.

[11] Alessio Siravo. Esecuzione di applicazioni all'interno di una macchina virtuale si Android. Tesi di Laurea in Informatica, Università di Bologna, 2012.

[12] Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

Ringraziamenti

Ringrazio la mia famiglia che mi ha sempre sostenuto e che mi ha dato la possibilità di raggiungere tanti obiettivi.
Ringrazio i miei amici e i colleghi con cui ho condiviso questa esperienza di studio.
Inoltre, voglio ringraziare Carlo Rimondi per la collaborazione al progetto di tesi e il Prof. Vittorio Ghini per avermi dato la possibilità di svolgere questo progetto, per la sua infinita pazienza e disponibilità e per avermi trasmesso le conoscenze necessarie.