Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea Magistrale in Informatica

Esperimenti di integrazione ViewOS-Rump: lo stack di rete NetBSD

Tesi di Laurea in Progetto di Sistemi Virtuali

Relatore:
Chiar.mo Prof.
Renzo Davoli

Presentata da:
Paolo Perfetti

Sessione III
Anno Accademico 2012-2013

a Mips

Introduzione

L'attuale panorama delle tecniche di virtualizzazione e compartimentazione delle risorse disponibili su un sistema è molto ampio [4, 24] e le diverse soluzioni sviluppate sono motivate da esigenze differenti e parzialmente contrapposte.

I computer sempre più performanti dispongono di risorse che, se dedicate ad un singolo utente o servizio, sono destinate a rimanere largamente inutilizzate. I meccanismi di time-sharing e multiprogramming, già responsabili della condivisione delle risorse disponibili, non rappresentano un meccanismo d'isolamento sufficiente a garantire la robustezza di un sistema rispetto ad applicativi concorrenti in esecuzione. Bug o programmi maliziosi in esecuzione possono infatti pregiudicare la stabilità dell'intero sistema.

Il ramo della ricerca orientata all'ottenimento di performance migliori ed allo sfruttamento ottimale delle risorse ha portato alle tecniche di paravirtualizzazione [1,28] e partizione di sistemi basati su container [13,26].

Un differente approccio ha scelto di realizzare l'astrazione di un sistema a livello di Instruction Set Architecture (ISA) realizzando la virtualizzazione (o emulazione) completa di una architettura hardware [2, 20]. Con alcune penalità sulle performance si può così ottenere l'isolamento totale del sistema guest all'interno di un processo utente, facendo affidamento sui sistemi standard per la protezione del sistema operativo host.

Nel quadro precedente, la *(para)virtualizzazione parziale* rappresenta un tentativo di cogliere i vantaggi dei due diversi approcci. Il fine è quello di offrire l'isolamento e la flessibilità derivante dalla virtualizzazione effettuata

a livello utente evitando alcune delle penalità alle performance introdotte dalla replicazione delle funzionalità già offerte dal kernel del sistema host.

ViewOS, una delle componenti del progetto Virtual Square (V^2) [5], persegue questo obiettivo rompendo la *global view assumption* secondo cui ogni processo deve avere la stessa percezione del sistema host su cui è eseguito [9]. Basandosi sulla convenzione UNIX che "tutto è un file", lascia ad ogni processo la possibilità di modificare la propria visione del filesystem e quindi dell'ambiente d'esecuzione. In particolare anche lo stack di networking, tradizionale eccezione al namespace fornito dal filesystem, viene reso accessibile come file grazie all'estensione della Berkeley socket API introdotta con msocket [7].

Il progetto **rump** [15] rappresenta un tentativo di paravirtualizzazione parziale implementato in NetBSD. Nato originalmente dall'esigenza di sviluppare e testare driver in userspace, si presta molto bene alla realizzazione di reti virtuali grazie all'impatto minimo che ha sulle prestazioni. Lo sviluppo di questo strumento ha portato alla trasformazione di NetBSD nel primo *anykernel* e ha reso quindi possibile l'estrazione e l'esecuzione in userspace dei driver integrati nel kernel, tra cui lo stack TCP-IP.

Dalla disponibilità di questi strumenti ha avuto origine l'idea di sviluppare **umnetbsd**, un modulo di UMView che include al suo interno lo stack di rete NetBSD eseguito e reso accessibile tramite l'hypervisor rump per Linux. L'integrazione realizzata grazie a `umnetbsd` rende possibile eseguire un applicazione non modificata in ambiente UMView utilizzando uno stack di rete alternativo a quello del sistema host. A differenza della virtualizzazione totale le penalità di performance e l'occupazione di memoria sono minime e l'utilizzo dello stack di NetBSD offre garanzia di stabilità.

La realizzazione contemporanea di `libvdeif` permette inoltre il collegamento diretto di un kernel rump allo switch virtuale realizzato da `vde_switch`. Seppur creato come strumento funzionale al debugging di `umnetbsd`, da sola questa libreria apre nuovi scenari per l'utilizzo dei kernel rump nella creazione di reti virtuali con overhead minimo.

La presentazione del lavoro svolto è suddivisa in tre parti principali:

- nei capitoli 1, 2 e 3 vengono presentate le idee che hanno guidato lo sviluppo del software utilizzato e di `umnetbsd`;
- nella seconda parte, i capitoli 4 e 5, si scende nel dettaglio delle architetture di UMView e rump e degli strumenti a disposizione;
- nella terza parte, capitolo 6, viene presentato il risultato originale di questa tesi ed alcuni dettagli dell'implementazione di `umnetbsd`;

In chiusura, il capitolo 7 presenta un panoramica dei possibili sviluppi futuri ed il capitolo 8 alcune considerazioni sul lavoro svolto.

Infine l'appendice A offre maggiori dettagli sull'ambiente di sviluppo e test.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Virtualizzazione

In informatica il concetto di *virtualizzazione* si sovrappone e spesso sostituisce a quello di *astrazione*. Nell'ambiente dello sviluppo software, dove lo stesso "reale" oggetto di studio manca di una componente fisica, la distinzione tra virtuale e tangibile perde di significato in favore della contrapposizione tra *astrazione* ed *implementazione*.

L'aspetto e la forma esteriori che una componente software espone agli altri elementi di un sistema complesso vengono chiamate *interfaccia* ed essa è definita come l'insieme di caratteristiche osservabili ed interazioni possibili che la componente espone al contesto in cui essa è immersa. Stratificando questo approccio e definendo $A_0, A_1..., A_n$ come una sequenza di componenti in cui A_i utilizza l'interfaccia $I(A_{i-1})$ ed espone $I(A_i)$ verso il livello A_{i+1}, quello che si ottiene è una catena di astrazioni di cui A_n rappresenta il livello più alto, ovvero il meno dettagliato. In questa stratificazione è possibile definire la virtualizzazione come astrazione da realizzarsi sostituendo un elemento A_i con un nuovo A'_i, componente che espone la medesima interfaccia $I(A_i)$ e che ne realizza le stesse funzioni. I dettagli implementativi di A_i e A'_i sono completamente opachi verso i livelli superiori e le due componenti potrebbero essere banalmente equivalenti, contenute una nell'altra o anche completamente disgiunte.

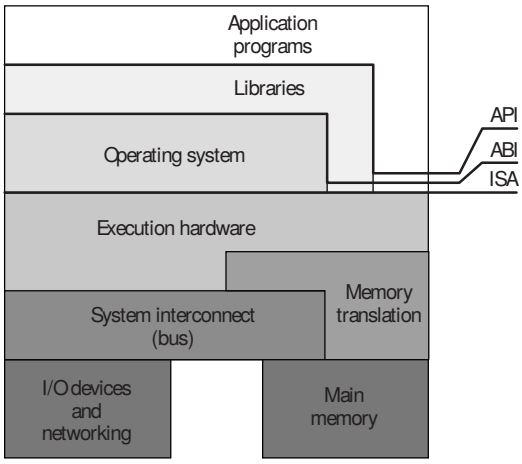
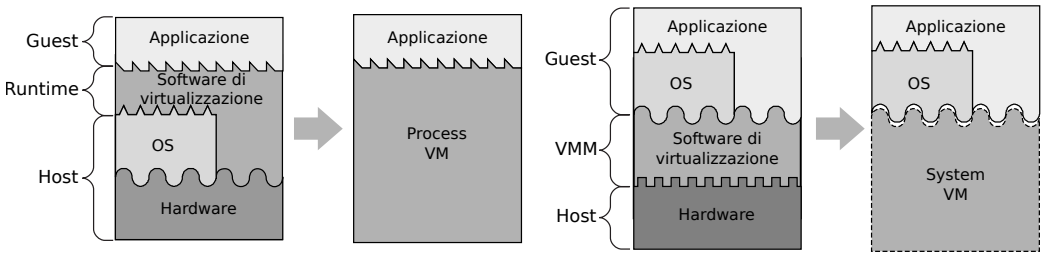


Figura 1.1: Architettura e interfacce di un sistema operativo

1.1 Macchine virtuali

Diversamente dall'astrazione, la virtualizzazione non mira a semplificare o astrarre i dettagli dei livelli sottostanti, ma piuttosto ad arbitrare l'accesso alle risorse limitate o a rendere disponibile un ambiente specifico in cui sia possibile eseguire una determinata classe di software. L'ambiente che si realizza in quest'ultimo caso prende il nome Macchina Virtuale (*Virtual Machine* o VM).

Una prima divisione e tassonomia [25] delle VM si basa sulla diversa vista che processi e sistemi operativi hanno dell'ambiente in cui vengono eseguiti. Per un processo l'ambiente d'esecuzione consiste nello spazio di memoria a disposizione, nelle librerie, nei registri della CPU e nelle system call messe fornite dal sistema operativo (SO) . L'interfaccia verso il livello inferiore è data dalla somma dell'Application Program Interface (API) delle librerie e dell'Appication Binary Interface (ABI) del SO (fig. 1.1). Un sistema operativo invece è concepito per avere accesso diretto all'hardware ed arbitrarne la disponibilità nell'ambiente messo a disposizione per l'esecuzione dei processi. L'interfaccia fornita dai device al SO è chiamata ISA (Instruction Set Architecture) ed è quella che definisce la vista che il sistema operativo ha del calcolatore su cui è in esecuzione.



(a) Process VM (b) System VM

Figura 1.2: Tipi di Virtual Machine

Da questa diversa visione che processi e sistemi operativi hanno della macchina su cui vengono eseguiti deriva la prima classificazione delle VM, divise tra *process VM* e *system VM*.

Process VM La virtualizzazione fornita da queste VM si colloca a livello dell'API e dell'ABI visibili ai programmi. Consiste nella creazione di uno specifico ambiente d'esecuzione, detto *runtime*, che nasce e muore con il processo. La compartimentazione dei processi ad opera del sistema operativo in ambiente multiprogramming può essere vista come la più basilare delle process VM. Le VM di esecuzione dei linguaggi ad alto livello (JVM, .NET, javascript, etc. . .) rientrano tutte in questo gruppo.

System VM Diversamente dalle process VM, l'ambiente fornito da questa classe di VM è persistente e virtualizza un sistema hardware completo, più precisamente un ISA, su cui è possibile l'esecuzione di un sistema operativo con tutte le sue componenti. Il software che si occupa della virtualizzazione delle risorse viene chiamato *monitor* (*Virtual Machine Monitor*, VMM) e può essere eseguito direttamente sull'hardware oppure all'interno di un sistema operativo *host*. L'ISA virtualizzata può essere la stessa dell'hardware fisico sottostante, nel qual caso si parla di virtualizzazione *omogenea*, oppure differente, ed allora si parla di virtualizzazione *eterogenea*.

In [5] viene introdotta un'ulteriore classificazione delle tecniche di virtualizzazione basata sul livello d'intrusività rispetto al sistema ospitante. Non tutti i monitor, inteso sia come runtime che come VMM, richiedono lo stesso livello di accesso alle risorse dell'host su cui vengono eseguiti e possono suddividersi in:

Accesso User Level Il monitor può essere eseguito da qualsiasi utente privo di accesso privilegiato, non può accedere direttamente all'hardware e l'interfaccia esposta verso la componente virtualizzata viene rimappata nell'ambiente non-privilegiato. Questa tecnica permette lo sviluppo di strumenti estremamente portabili, senza particolari restrizioni alle funzionalità virtualizzate ma può introdurre problemi di prestazioni;

Accesso Super utente Il monitor richiede i privilegi da super utente ed è quindi in grado di sfruttare a pieno le funzionalità fornite dal sistema operativo host. Questo livello di accesso permette il miglioramento delle prestazioni e l'applicazione di politiche di sicurezza diverse a VM differenti. Al contempo introduce però un isolamento imperfetto: l'esecuzione di codice con permessi da amministratore ha ovvie implicazioni sulla sicurezza e per questo l'accesso a questo tipi di strumenti è solitamente limitato ad utenti fidati;

Accesso kernel level Il monitor richiede che alcune funzionalità siano implementate, come modulo o come patch, all'interno del kernel del sistema host. Questa soluzione può introdurre un significativo miglioramento alle prestazioni ma il codice eseguito in kernel-mode dev'essere provato stabile e robusto per evitare l'insorgere di gravi problemi di sicurezza ed affidabilità;

Accesso nativo Il monitor stesso agisce in qualità di sistema host. L'accesso diretto all'hardware garantisce le prestazioni migliori e al contempo l'affidabilità e la sicurezza del sistema dipendono solamente dal monitor.

1.2 Paravirtualizzazione

Il vincolo principale imposto dalla virtualizzazione è quello di mantenere inalterata l'interfaccia esposta verso gli strati superiori della catena di astrazioni. Questo vincolo garantisce la compatibilità delle componenti software preesistenti ma al contempo limita lo sviluppo di nuove soluzioni che richiedono la presenza di un supporto specifico all'interno del software virtualizzato.

La tecniche di paravirtualizzazione ([1, 28]) rompono questa compatibilità introducendo delle modifiche all'ISA e alla sua semantica ed obbligando così i sistemi operativi e le applicazioni virtualizzate a prevedere un supporto specifico per il VMM. Tali modifiche mirano ad alleggerire lo strato di virtualizzazione anche mediante la semplificazione di alcune funzionalità (es. gestione della memoria virtuale) e delegando il controllo d'accesso ed arbitraggio delle risorse hardware al VMM.

Anche se gli obiettivi specifici dei vari progetti sono differenti, in entrambi i casi il risultato ottenuto dall'introduzione della paravirtualizzazione è un incremento significativo nelle performance ottenute dai sistemi virtualizzati. Inoltre la presenza di un supporto specifico permette ad un SO di rendersi conto della propria condizione di *sistema guest* e quindi di interagire opportunamente con il VMM in esecuzione.

Capitolo 2

Virtual Square

Virtual Square (V^2), nato inizialmente come progetto di ricerca, costituisce oggi un framework che raccoglie diversi strumenti dedicati allo studio delle tecniche di virtualizzazione ed alla realizzazione di sistemi virtuali.

Lo scopo è quello di fornire un ambiente unico che integri i tool disponibili, fornendo principi comuni e creando al contempo soluzioni semplici e modulari per una virtualizzazione parziale. In questo senso, il nome stesso Virtual Square (dove *square* assume il significato di *piazza*) indica l'obiettivo primario di realizzare un punto d'incontro e di dialogo tra gli strumenti esistenti.

I principi di V^2 sono in comune con molti altri progetti di Free Software e si basano sulla libertà del codice, il riutilizzo di strumenti standard, il design modulare e poche scelte architetturali rigide.

Basandosi sul riutilizzo degli elementi a disposizione e puntando ad interconnettere tecnologie pre-esistenti, V^2 si può definire come una "*virtualizzazione fatta di virtualizzazioni*", elevando a potenza il concetto stesso di virtualità (da cui il secondo significato del nome, dove *square* è tradotto come *quadrato*).

2.1 Vista globale e virtualizzazione parziale

In un sistema operativo tradizionale l'ambiente di esecuzione di un'applicazione viene definito e limitato dal kernel. Ogni processo può accedere liberamente solo alla propria memoria allocata mentre per interagire con il sistema operativo o accedere alle risorse disponibili, un programma può solamente ricorrere all'invocazione delle *system call*.

Questo design basato su dei punti d'accesso ristretti ed unificati è dettato dal bisogno di sicurezza ed affidabilità: il sistema operativo deve sia impedire che un processo malizioso pregiudichi la sicurezza globale, sia deve proteggere i singoli programmi dalle interferenze che potrebbero derivare dall'esecuzione concorrente di altri processi. Dal punto di vista del kernel le system call rappresentano il mezzo di controllo totale sulle attività di un processo, metodi astratti la cui unica implementazione sicura è quella messa a disposizione dal sistema operativo al fine di garantire la propria sicurezza ed integrità.

Ribaltando il punto di vista ed osservando con gli occhi di un processo, le system call rappresentano l'unico punto di accesso al mondo esterno, una sorta di "finestra" attraverso la quale è possibile osservare e parzialmente modificare il contesto d'esecuzione. Lo stesso processo in esecuzione su due host differenti e che invochi la medesima system call riceverà risposte tendenzialmente diverse (es. la lista dei filesystem, l'indirizzo di rete, la tabella di routing, ...). Da questa considerazione si può dire che il mondo osservato attraverso l'interfaccia definita dalle system call costituisce la "vista" (*view*) che il processo ha del proprio contesto. Tutti i programmi le cui system call sono esaudite dallo stesso kernel condividono quindi la stessa vista, ovvero si ha una *global view assumption*, GVA [9].

Dall'assunzione forzata della GVA dipendono alcuni dei limiti e delle scelte effettuate nel design dei sistemi POSIX ed in particolare a quelle che potrebbero sembrare feature di sicurezza. Un esempio significativo è rappresentato dalla system call `mount` e dall'assunzione che tutti i processi condividono la visione di un unico namespace per il filesystem e di un'unica tabella di mount [23]. A queste condizioni risulta ovvio come l'operazione di `mount`,

con cui è possibile definire e sovrascrivere i rami del filesystem visibile dai processi, diventi particolarmente sensibile dal punto di vista della sicurezza. I file dedicati all'autenticazione e alla gestione dei permessi (`/etc/passwd` , `/etc/group`, ...) e i device accessibili sotto la directory `/dev/` sono solo alcuni esempi di come sia possibile modificare sostanzialmente e pericolosamente la vista che un processo ha del proprio ambiente d'esecuzione.

Al contempo la semantica che deriva dalla GVA risulta eccessivamente restrittiva in un ambiente dove ogni utente può legittimamente accedere a filesystem locali o remoti (CD, memorie USB, condivisioni NFS/CIFS/SSH, etc). Per far fronte a queste restrizioni sono state realizzate diverse eccezioni, alcune a livello di configurazione (l'opzione `user` in `/etc/fstab`) altre con utility dedicate (udisks).

Adattandosi alla visione unica imposta ai processi, di fatto sono stati realizzati strumenti che richiedono privilegi di amministratore ma che in realtà rendono possibile all'utente ciò che comunque è già in grado di fare. Un utente qualsiasi può eseguire l'istanza di una macchina virtuale completa (es. QEMU) definendo quali filesystem utilizzare: i processi al suo interno beneficeranno di una vista differente senza per questo compromettere la sicurezza del sistema operativo host o degli altri programmi in esecuzione.

2.2 ViewOS

Un tentativo radicale di modifica della semantica della GVA è costituito dall'idea di ViewOS [9]: garantire ad ogni processo la possibilità di modificare la propria vista dell'ambiente circostante modificando il comportamento delle system call effettuate. Come User Mode Linux (UML), ViewOS intercetta le system call e procede alla loro ridefinizione per adattarle agli scopi richiesti, ma diversamente da UML non impone l'utilizzo di un kernel completo con conseguente overhead per svolgere questo compito.

L'elemento principale di ViewOS è costituito da un hypervisor minimale a cui è possibile collegare i moduli che realizzano le diverse parti della virtualiz-

zazione. Il compito dell'hypervisor è quello di intercettare tutte le chiamate a sistema effettuate dal processo controllato e verificare se richiedono una funzionalità gestita da qualche modulo. In caso così non fosse (comportamento previsto di default dalle implementazione di ViewOS) il funzionamento è paragonabile a quello di un proxy che inoltra solamente le richieste al livello sottostante, sia esso un'altra istanza di ViewOS oppure il sistema host.

A differenza di altre tecnologie ViewOS non si pone come obiettivo primario l'isolamento totale del processo, ma piuttosto l'integrazione tra l'ambiente fornito dal sistema host e le variazioni effettuate dall'hypervisor alla *view* del processo. In questo senso la virtualizzazione realizzata da ViewOS può essere definita *parziale*, dove solo alcune componenti vengono sovrascritte a seconda del compito che si vuole realizzare, dei moduli caricati e delle opzioni utilizzate.

Molti tool diffusi si occupano già della virtualizzazione di alcune singole funzionalità ma nessuno di questi prevede un ambiente integrato in cui è possibile definire nel dettaglio quali aspetti della vista e quali chiamate a sistema devono essere gestite e come. ViewOS invece nasce all'interno del Virtual Square Framework (V^2) da cui riprende i principi di modularità, flessibilità e libertà.

2.3 Virtual Distributed Ethernet

Parte fondamentale di V^2 è costituita dal framework di virtualizzazione di reti ethernet VDE (*Virtual Distributed Ethernet*), una suite di applicativi che permette di interconnettere sistemi reali e virtuali.

L'obiettivo di VDE è rappresentare una soluzione unica, geograficamente distribuibile, utilizzabile da utenti non amministratori, compatibile con la rete ethernet reale e facilmente collegabile ad ogni tipo di device e software di virtualizzazione. Per questa sua centralità costituisce uno dei primi elementi di V^2 e certamente ne rappresenta bene i principi di apertura e interoperabilità.

Per l'interconnessione delle sue componenti VDE si basa sull'astrazione UNIX per cui la semantica utilizzata per comunicazioni ed oggetti è rispettivamente quella di stream di input/output e di file/filesystem. I diversi programmi, e quindi i device virtuali di rete, possono essere eseguiti su host differenti e gli stream possono essere instradati su strumenti standard come ssh e per questo si può definire un framework distribuito.

Nel rispetto della filosofia UNIX, la suite è costituita da piccoli programmi dedicati ad un compito preciso ed ognuno di essi può essere eseguito per virtualizzare uno specifico device o implementare un comportamento definito. Alcuni dei componenti principali sono:

vde_switch realizza l'emulazione software di uno switch ethernet "manageable". Il canale di comunicazione verso gli altri elementi della rete virtuale viene realizzato da un socket UNIX specificabile da riga di comando e l'accesso può essere gestito attraverso il sistema standard dei permessi. `vde_switch` accetta le connessioni dei `vde_plug` che a loro volta possono essere collegati a `vde_switch`, device virtuali o stream. Lo switch virtuale può essere gestito direttamente o attraverso `vdeterm` un altro applicativo che agisce come console di manageament. Tra i protocolli supportati da `vde_switch` appaiono le VLAN (IEEE802.1Q) e lo Fast Spanning Tree Protocol [1].

vde_plug rappresenta l'equivalente software di un plug ethernet ed è il principale e più semplice strumento di connessione ad una porta dello switch virtuale. Si occupa di inoltrare quel che riceve sullo standard input verso lo switch e, viceversa, il traffico in arrivo dal `vde_switch` a cui è collegato sullo standard output. Intuitivamente un cavo di collegamento tra due switch virtuali è realizzato semplicemente connettendo tra loro due `vde_plug` attraverso strumenti che supportino un forward full-duplex come `dpipe` [2].

[1] http://wiki.virtualsquare.org/wiki/index.php/Fast_Spanning_Tree_Protocol
[2] http://wiki.virtualsquare.org/wiki/index.php?title=VDE#dpipe

vde_plug2tap come `vde_plug` si connette ad una porta di `vde_switch`, ma invece di redirigere il traffico da/per lo stdin/stdout stabilisce una corrispondenza con un interfaccia di rete virtuale `tap`. Questo tool è quello che permette di stabilire un collegamento tra una rete virtuale realizzate con VDE ed un interfaccia reale presente sul sistema host.

slirpvde agisce come un gateway dall'interno della rete VDE verso la rete della macchina ospite su cui viene eseguito. Sebbene non si tratti dell'unico strumento che consente di interconnettere la rete virtuale costruita con il framework VDE è l'unico che permette di effettuare routing e masquerading senza permessi di super-utente. `slirpvde` permette di tradurre gli indirizzi dalla classe di indirizzamento interna a quella esterna, fornisce un servizio di assegnamento dinamico degli indirizzi e permette di inserire semplici regole di port-forwarding e di re-direzione delle richieste DNS.

libvdeplug a differenza delle componenti elencate sinora, programmi completi e stand-alone, questa libreria fornisce un'API semplice e funzionale (figura 2.1) per connettere macchine virtuali ed applicazioni ad un `vde_switch`. È già utilizzata da molte tecnologie di virtualizzazione tra cui QEMU, KVM, User-Mode Linux e VirtualBox e nel capitolo 6 si vedrà come sia stata fondamentale anche nello sviluppo di questo lavoro.

```
#define vde_open(vde_switch,descr,open_args)
   vde_open_real((vde_switch),(descr),
   LIBVDEPLUG_INTERFACE_VERSION,(open_args))
VDECONN *vde_open_real(char *vde_switch,char *descr,int
   interface_version, struct vde_open_args *open_args);
ssize_t vde_recv(VDECONN *conn,void *buf,size_t len,int
   flags);
ssize_t vde_send(VDECONN *conn,const void *buf,size_t
   len,int flags);
int vde_datafd(VDECONN *conn);
int vde_ctlfd(VDECONN *conn);
int vde_close(VDECONN *conn);
```

Figura 2.1: API implementata da `libvdeplug`

Capitolo 3

L'architettura anykernel

I sistemi operativi attualmente più diffusi hanno come elemento strutturale comune la scelta di utilizzare un'architettura monolitica per i loro kernel. Tale architettura prevede che tutto il codice che implementa le funzionalità fornite dal kernel sia eseguito in un unico dominio privilegiato piuttosto che suddiviso tra processi differenti che comunicano attraverso message passing.

L'esistenza di un unico dominio di fatto permette ad ogni componente del kernel di accedere direttamente a tutta la memoria allocata, ad ogni funzione e device, senza imporre una struttura precisa da rispettare. Tuttavia, vista la complessità raggiunta dal kernel e la necessità di uno sviluppo ordinato, anche nell'ambito di kernel monolitici si tende a rispettare convenzioni ed uno sviluppo strutturato.

Nonostante il loro largo impiego, alcuni problemi caratterizzano lo sviluppo e l'utilizzo dei kernel monolitici:

Insicurezza e vulnerabilità L'utilizzo di un unico dominio per l'esecuzione di tutte le componenti ed i driver rappresenta una fonte di criticità dato che un singolo problema può pregiudicare la stabilità dell'intero sistema. Oltre ai bug, questo approccio unificato può rendere il kernel oggetto di attacchi che mirano all'esecuzione di codice in modalità protetta sfruttando vulnerabilità presenti in driver che accedono a filesystem insicuri [30] .

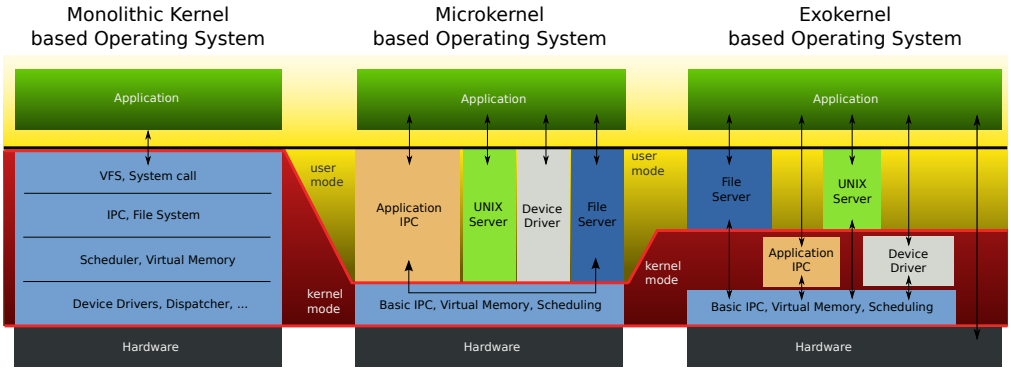


Figura 3.1: Categorie di kernel

Limitato riutilizzo del codice e complessità dello sviluppo La mancanza di strutturazione tipica dei kernel monolitici rende difficile l'estrazione di particolari funzionalità immerse nel codice. A causa della difficoltà nella risoluzione delle dipendenze e nell'identificazione di singole componenti, spesso viene richiesto un lungo lavoro di adattamento del codice o addirittura la scrittura da zero per poter eseguire in user-space driver altrimenti funzionanti in kernel-mode. Come conseguenza di questo fenomeno si ha un ulteriore complicazione al modello di sviluppo del kernel, per il quale non è possibile testare il codice se non come parte di un intero sistema operativo. Tradizionalmente questo comporterebbe che le procedure di test avvengano installando il kernel direttamente sull'hardware ma alcune soluzioni di virtualizzazione (es. QEMU) possono aiutare. Nonostante questo l'overhead introdotto è significativo in quanto il test di un singolo componente comporta il bootstrap di un intero kernel, con conseguente spreco di tempo e memoria.

3.1 Categorie di kernel

Alcune architetture alternative sono state proposte per lo sviluppo dei kernel:

Microkernel Vengono eseguite in modalità privilegiata solo poche funzionalità dedicate alla comunicazione tra i processi, allo scheduling dei thread ed alla gestione della memoria virtuale mentre tutti i driver dei device, gli stack di rete, i filesystem vengono delegati a server in userspace. Tra i vantaggi di questa architettura ci sono la facilità di sviluppo e la quantità minima di codice in esecuzione in modalità privilegiata (meno di 6000 righe per MINIX [10]). Questi vantaggi comportano però un prezzo importante in termini di prestazioni raggiunte.

Exokernel Simili ai precedenti microkernel, perseguono uno scopo differente, ovvero la riduzione della distanza e dell'astrazione imposta ai programmi in user space. Ad ogni applicativo viene lasciata la possibilità di scegliere quale livello di astrazione utilizzare rispetto ai device, permettendo così a software con esigenze particolari di raggiungere prestazioni più alte di quelle offerte dai kernel monolitici.

Le architetture alternative a quella monolitica hanno però un problema in comune: la complessità richiesta ad un sistema operativo che sia utilizzabile in produzione ed in casi reali è così alta che l'implementazione da zero di nuove soluzioni diventa sempre più difficile da realizzare. Microkernel come MINIX o HURD sono sviluppati da più di 20 anni ma ancora oggi non sono in grado di rispondere alle necessità reali di un sistema operativo general purpose.

3.2 anykernel e kernel Rump

Con lo scopo principale di risolvere i problemi sopra esposti, lo sviluppatore NetBSD Antti Kantee [15] ha affrontato la questione proponendo una soluzione efficace ed allo stesso tempo praticabile.

In contrapposizione con le architetture viste sinora, dove già a partire dal nome si prevedeva una determinata strutturazione del codice e delle funzionalità implementate, l'idea di Kantee è quella di definire un'interfaccia

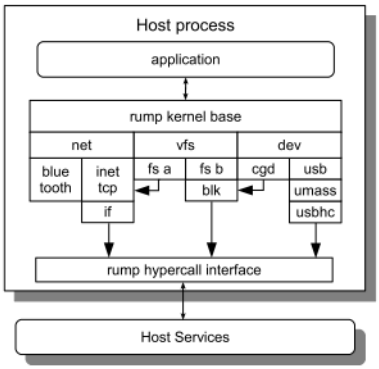


Figura 3.2: L'architettura anykernel

minimale ed uniforme su cui gli sviluppatori possono basarsi durante la programmazione di componenti per il kernel. Tale approccio prende il nome di *anykernel* e riflette bene la flessibilità che caratterizza questa soluzione, dove la modalità di esecuzione di una determinata componente può essere decisa a runtime. L'interfaccia definita è paragonabile a quella di un mini hypervisor che si occupa di fornire le funzionalità minime non scorporabili, una sorta di kernel parziale che viene detto *rump kernel*.

L'introduzione di questa stratificazione permette all'utente di scegliere dinamicamente dove collocarsi nel trade-off tra sicurezza, flessibilità e prestazioni. Qualora la priorità siano le performance in esecuzione ed il driver di interesse sia ben testato ed integrato nel kernel standard, allora sarà possibile mantenere l'approccio monolitico senza incorrere in alcuna penalità prestazionale. Se invece l'utente si sta occupando dello sviluppo di funzionalità aggiuntive per il kernel, oppure sta testando dei driver, o ancora vuole utilizzare alcune componenti isolandosi dal resto del sistema operativo e senza incorrere in problemi di sicurezza, allora è possibile eseguire un kernel rump che virtualizzi le funzionalità minime e che includa il codice oggetto di studio. Tale istanza può essere eseguita completamente in user-space e per questo non ha alcun impatto particolare sulla stabilità del sistema *host* su cui gira il kernel rump.

È possibile accedere alle funzionalità di un kernel rump sia come libreria

locale, sfruttando direttamente l'API esposta, sia come server remoto. In questo secondo caso il kernel rump è in esecuzione in un processo differente ospitato sullo stesso sistema oppure su uno diverso ed è possibile comunicarvi attraverso un socket UNIX oppure TCP/IP. L'API fornita è la stessa utilizzabile all'interno del kernel NetBSD ed applicazioni scritte appositamente possono essere eseguite in qualunque delle modalità kernel sopra esposte. Grazie alla libreria `librumphijack` è inoltre possibile eseguire applicazioni non modificate dirottando le system call in modo che vengano soddisfatte da un'istanza di un kernel rump.

Un kernel rump può essere eseguito su qualunque piattaforma POSIX come processo utente senza privilegi e, grazie a questo approccio minimalista, introduce un overhead minimo alle operazioni richieste dai driver e dalle componenti che lo utilizzano (nell'ordine di 1MB di memoria e 10ms di tempo di boot [15]).

3.3 Paravirtualizzazione parziale in Rump

Lo scopo dei kernel rump è quello di eseguire in userspace i driver inclusi in un un kernel anykernel senza che sia necessario apportarvi modifiche e con il minor overhead possibile. Una soluzione praticabile per lo sviluppo ed il test delle componenti del kernel sarebbe stata quella di ricorrere alla virtualizzazione completa di un sistema operativo, ma questo avrebbe portato a delle penalità di performance troppi pesanti per essere applicata. Prestazioni migliori possono essere ottenute utilizzando i containers ([3], [26]) ma in questo caso il contesto di esecuzione dei driver è condiviso con quello del sistema host e quindi viene meno il requisito di isolamento: un bug in uno dei containers comporterebbe l'instabilità dell'intero sistema host.

Nella virtualizzazione totale di un kernel emergono diverse problematiche dovute alla duplicazione dei compiti spettanti al sistema operativo. Si prenda ad esempio un sistema host che ospiti differenti kernel guest all'interno dei quali siano in esecuzione diverse applicazioni concorrenti. In uno scenario

standard lo scheduler host designerà uno dei kernel guest come attivo. A sua volta, il kernel guest invocherà il proprio scheduler per decidere quale dei processi interni (invisibili all'host) dovrà prendere il controllo del processore. Chiaramente questo doppio livello di indirezione non può che penalizzare le performance introducendo una complessità non richiesta.

Nei kernel rump invece sono virtualizzate solamente le componenti strettamente necessarie, delegando al sistema host per quanto non esplicitamente implementato all'interno del driver in considerazione. Inoltre, trattandosi di virtualizzazione e non solamente di ridefinizione di namespace, il contesto di esecuzione di un kernel rump è esterno a quello del kernel host. I privilegi di un kernel rump dipendono solamente dall'utente con cui viene eseguito ed il suo isolamento e controllo sono quelli di un qualsiasi altro processo in esecuzione con i medesimi privilegi.

3.4 Struttura dei kernel rump

Nello sviluppo di rump il codice del kernel NetBSD è stato suddiviso in tre livelli principali:

base Indipendentemente dalle componenti che poi verranno fornite qui sono contenute le funzionalità comuni a tutti i kernel rump come i meccanismi di lock e l'allocazione della memoria. Questo livello dipende solamente dalle hypercall definite dall'interfaccia anykernel e rappresenta la dipendenza necessaria per ogni kernel rump;

factions Questa ripartizione suddivide i sottosistemi di riferimento dei driver e le risorse richieste all'esecuzione degli stessi. Le factions sono 3: *devices*, *network* e *(virtual) filesystem*. Sono richieste essere ortogonali tra di loro, ovvero di poter funzionare indipendentemente l'una dall'altra, al fine di minimizzare l'overhead richiesto per soddisfare le dipendenze dei singoli componenti;

drivers I driver rappresentano tutte le componenti che forniscono accesso a protocolli e device disponibili sul sistema. Dal punto di vista dell'utente sono le componenti che forniscono i servizi esternamente interrogabili di un sistema operativo. Nella suddivisione introdotta possono richiedere la presenza di una o più faction sottostanti ed anche dipendere da altri driver.

Capitolo 4

La virtual machine UMView

L'idea portata da ViewOS è stata implementata in due diverse soluzioni usualmente raggruppate con il nome di *MView:

KMView L'intercettazione delle system call avviene tramite un modulo del kernel basato sulle funzionalità di debugging introdotte da *utrace()*[1] di Roland McGrath. Delle soluzioni implementate è quella con le performance migliori e non deve sottostare ai limiti imposti per gestione dei segnali, ma richiede i privilegi di root per l'installazione del modulo.

UMView Implementazione interamente in user space, si basa su *ptrace()*. Garantisce la flessibilità più elevata e non richiede alcun permesso speciale per essere eseguita, ma deve fare i conti con i limiti imposti dalla `ptrace()`. Questa system call è infatti progettata per il debugging e non esplicitamente per la virtualizzazione.

Come si è appena visto, *MView realizza l'idea proposta da ViewOS sfruttando i meccanismi di intercettazione delle system call messi a disposizione dal sistema operativo host. Formalizzando si può definire *MView una System Call Virtual Machine (SCVM) e secondo la tassonomia introdotta nel paragrafo 1.1 si tratta di una ProcessVM. Inoltre, dato che un elemento fondamentale di ViewOS è proprio quello di introdurre un framework unico

[1] https://sourceware.org/systemtap/wiki/utrace

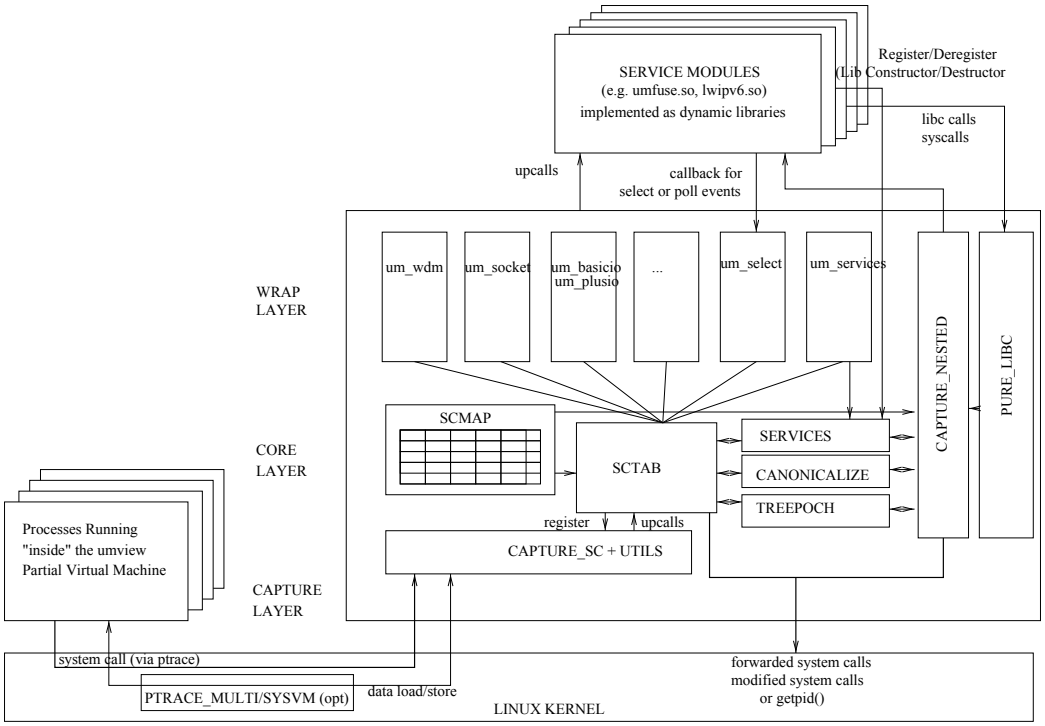


Figura 4.1: Architettura di UMView (immagine tratta da [5])

tramite cui definire quali sono le parti di un sistema che si vogliono virtualizzare, si può affinare la definizione parlando di Partial Process Virtual Machine.

Durante lo sviluppo del sottomodulo `umnetbsd`, risultato principale di questo lavoro di tesi, è stata posta particolare attenzione alla possibilità di effettuare una virtualizzazione parziale ed in particolare dello stack di rete con i soli privilegi di utente. Per questo durante il lavoro di ricerca si è scelto di utilizzare sempre l'implementazione di userspace di ViewOS, ovvero UMView (User-Mode ViewOS).

4.1 Architettura generale

Sin dal principio i sistemi UNIX-like hanno sempre seguito la filosofia secondo cui "tutto è un file". In particolare device e meccanismi di InterProcess Communication (IPC), le pipe, sono stati pensati in maniera da essere

Nome macro	Significato
STD_BEHAVIOR	Il kernel esegue la system call come abitualmente
SC_FAKE	Se presente l'ottimizzazione PTRACE_VM non viene eseguita alcuna sc, altrimenti viene sostituita con una `getpid()`
SC_CALLONXIT	Il kernel esegue la sc ma il risultato richiede un'ulteriore manipolazione in fase di ritorno

Tabella 4.1: Possibili comportamenti delle system call intercettate

compatibili ed utilizzabile tramite le stesse system call previste per la manipolazione dei file ordinari [23]. Un altro elemento innovativo emerso per la prima volta in seguito a tale assunzione, è stata la definizione di un unico filesystem su cui fosse possibile "attaccare" e "staccare" filesystem secondari tramite le operazioni di `mount` e `umount`. Questo filesystem globale, la cui radice è la root ('/') di fatto definisce l'unico namespace di riferimento per ogni processo in esecuzione su un sistema UNIX-like, e attraverso esso è possibile accedere a tutte le risorse disponili e alle strutture interne del kernel [17]. Come diretta conseguenza di questa scelta architetturale si ha che modificando il filesystem visibile da un processo si modifica anche la percezione che esso ha del sistema su cui è in esecuzione.

ViewOS in particolare estende la semantica del comando `mount` facendo coincidere con un operazione di questo tipo la modifica della SCVM parziale in esecuzione. Infatti la prima estensione alla semantica di `mount` riguarda proprio la possibilità per un singolo processo di modificare la propria visione del filesystem, operazione altrimenti non possibile per effetto della GVA. Quali sono le system call il cui comportamento viene modificato da una chiamata a questa nuova `mount` dipende dal *filesystemtype* passato come parametro, ovvero dal modulo associato a quel *filesystemtype*.

4.2 Core di UMView

Il mainloop degli eventi si occupa di gestire sia le chiamate a system call intercettate tramite `ptrace()` sia i segnali asincroni provenienti dai moduli. In questo primo strato vengono registrate le system call da intercettare e per cui deve essere generata l'*upcall* verso la funzione wrapper che si occupa di scansionare le tabelle *scmap*. Queste tabelle implementano la logica di smistamento del core: in esse i moduli registrano le proprie system call e vengono definiti i wrapper invocati prima e dopo l'esecuzione delle chiamate stesse. A seconda della system call invocata e dei moduli caricati, il wrapper di ingresso definisce quale comportamento dev'essere seguito inoltrando la system call al modulo responsabile per essa (tabella 4.1)

Un'altra responsabilità del livello di intercettazione è quella di copiare gli argomenti delle system call dalla memoria del processo a quella di UMView e viceversa: queste operazioni rappresentano una delle principali fonti di overhead ed il team di ViewOS ha sviluppato un'apposita patch (PTRACE_MULTI [2]) per evitare i molti context switch che si rendono necessari.

4.3 Moduli *MView

Il core principale di *MView si occupa solamente dell'intercettazione delle system call ma non apporta alcuna modifica al loro comportamento. All'interno dell'ambiente UMView "vuoto" il processo virtualizzato ha la stessa visione dell'host che avrebbe se non fosse virtualizzato e tutte le chiamate a sistema vengono direttamente inoltrate al sistema ospite.

Il compito di modificare la visione che un processo ha dell'ambiente spetta ai moduli. La loro implementazione avviene nella forma di librerie dinamiche ed il loro utilizzo è determinato a runtime dall'utente. Le due possibilità per caricare un modulo, da riga di comando o utilizzando i comandi messi a disposizione da *MView, sono esemplificante nella figura 4.2

[2] https://lkml.org/lkml/2008/6/16/69

```
$ umview bash -p umnet
$ um_ls_service
umnet: virtual (multi-stack) networking
```

(a) Caricamento di un modulo da riga di comando

```
$ umview bash
$ um_ls_service
$ um_add_service umnet
$ um_ls_service
umnet: virtual (multi-stack) networking
```

(b) Caricamento di un modulo usando `um_add_service`

Figura 4.2: Caricamento di un modulo in *MView

Comando	**Effetto**
`um_add_service`	Carica il modulo specificato (eventualmente nella posizione indicata)
`um_ls_service`	Elenca i moduli caricati
`um_del_service`	Rimuove il modulo dai servizi disponibili

Tabella 4.2: Comandi per la gestione dei moduli di *MView

Per semplificare il routing delle system call e lo sviluppo dei moduli, il core di *MView utilizza sempre i pathname assoluti e, quando possibile, "collassa" le chiamate a sistema alla più generica system call definita per quella famiglia di funzioni (es. `create()` viene tradotta in una chiamata a `open(...,O_CREAT|O_WRONLY|O_TRUNC)`).

I file descriptor ritornati dalla system call implementate all'interno di un modulo sono trattati come semplici valori interi e non vengono né inoltrati all'applicazione virtualizzata, né condivisi con altri moduli. Il core di *MView utilizza questi valori interi per effettuare un mapping a quelli restituiti all'utente e per le successive chiamate alle socket call.

Data la natura single-thread di *MView e la necessità di rimanere in attesa del verificarsi di eventi su domini differenti (es. file descriptor creati

da diversi moduli), le system call di attesa quali `select()`, `poll()` e simili richiedono un supporto particolare. Questo supporto è implementato dalla funzione `*_event_subscribe` i cui dettagli sono discussi nella sezione 6.3.2

4.4 msocket

Una significativa eccezione alla convenzione UNIX per cui "tutto è un file" è costituito dai servizi di networking. Infatti, mentre per i device hardware, la comunicazione IPC, i socket UNIX esiste una corrispondenza diretta con il nome di un file presente sul file-system, tale corrispondenza non esiste per i device e gli stack di rete.

Nel tentativo di unificare questa visione del namespace ed in particolare di aggiungere il supporto per multipli stack di rete all'interno dello stesso sistema operativo, il progetto Virtual Square ha esteso l'interfaccia Berkeley socket API, standard *de facto* prima [27] e POSIX poi [21], con la definizione della system call *msocket* [7]. La sintassi di msocket è la seguente:

```
#include <sys/types.h>
#include <msocket.h>

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

dove *domain*, *type* e *protocol* mantengono la stessa semantica definita per socket mentre *path* definisce la desiderata associazione con il filesystem. Il funzionamento di una `socket()` standard può essere ottenuto passando `NULL` come argomento *path*.

4.5 mstack

Insieme all'estensione delle Berkeley socket API, il framework V^2 fornisce il tool `mstack` con lo scopo di rendere retro-compatibile l'utilizzo di `msocket`. Tramite l'uso di `mstack` è possibile specificare a quale stack di rete il core

```
$ mstack  -h
Usage:
  mstack [-hv] [-o protocol_list] stack_mountpoint command
  protocol_list may include: all, unix, ipv4, ipv6, netlink,
  irda or #num
  protocols may be prefixed by + (add) - (delete)
```

Figura 4.3: Sintassi di `mstack`

di UMView deve inoltrare le socketcall intercettate. Un esempio è dato dal seguente listato, in cui l'applicazione intercettata è l'eseguibile standard `ip`:

```
$ mount -t umnetnull none /dev/net/null
$ mstack /dev/net/null ip link
Cannot open netlink socket: Address family not supported by protocol
```

`mstack` costituisce la soluzione più comoda per permettere a programmi scritti senza un supporto specifico per `msocket` di sfruttare l'ambiente multi-stack messo a disposizione da UMView. In alternativa tutte le applicazioni dovrebbero prevedere un supporto specifico oppure l'ambiente dovrebbe accontentarsi di avere un unico stack, seppur diverso da quello dell'host, a cui inoltrare tutte le socketcall. Così facendo pero', ci si scontrerebbe nuovamente lo stesso limite che rappresenta una delle motivazioni principali dietro allo sviluppo di ViewOS.

Le altre opzioni dell'eseguibile `mstack` (fig. 4.3) servono a definire quali famiglie di protocolli sono disponibili o devono essere filtrate dallo stack in oggetto e la sintassi è la stessa utilizzata usata nei comandi `um_add_service` e `mount` .

4.6 Il modulo umnet

In ViewOS il supporto per la nuova semantica estesa fornita da msocket è implementato all'interno del modulo `umnet` che, insieme all'utility `mstack`, permette di eseguire programmi standard non modificati intercettando tutte le socket call e gestendole attraverso uno stack non-standard. Attraverso

Opzioni	Address family
`all` o `nothing`	tutti i protocolli
`u` o `unix`	AF_UNIX
`4` o `ipv4`	AF_INET (ipv4)
`6` o `ipv6`	AF_INET6 (ipv6)
`n` o `netlink`	AF_NETLINK
`p` o `packet`	AF_PACKET
`b` o `bluetooth`	AF_BLUETOOTH
`i` o `irda`	AF_IRDA
`ip`	tutti i protocolli TCP-IP (AF_INET, AF_INET6, AF_NETLINK e AF_PACKET)

Tabella 4.3: Opzioni di `umnet`

le opzioni passate al comando `um_add_service` è anche possibile avere una granularità maggiore sui servizi di rete resi disponibili o negati dal modulo `umnet`. Per esempio la riga:

```
$ um_add_service umnet,-ipv4
```

nega l'utilizzo della famiglia di protocolli AF_INET mentre lascia disponibili le altre (es. AF_UNIX, AF_INET6). Gli argomenti disponibili sono elencati nella tabella 4.3 e possono essere usati insieme ai prefissi `+` e `-`. Le stesse opzioni possono essere utilizzate sia dal comando `mstack` sia in fase di `mount` di un sotto-modulo con lo scopo di inibire particolari funzionalità. Per esempio

```
$ mount -t umnetcurrent none /dev/net/copia
$ mstack -o -ip /dev/net/copia bash
```

virtualizza una shell a cui è interdetto l'utilizzo dei protocolli della famiglia AF_INET mentre continua ad avere accesso ai protocolli ipv6 messi a disposizione dallo stack del sistema host.

4.7 Sottomoduli di `umnet`

I sottomoduli attualmente disponibili per umnet sono quattro e di questi solo `umnetlwipv6` implementa uno stack alternativo a quello disponibile sul

sistema host. Nel capitolo 6 viene esaminato lo sviluppo e lo stato attuale del sottomodulo `umnetbsd`, il primo tentativo di includere uno stack completo ed ampiamente testato come quello di NetBSD tra le funzionalità rese disponibili da UMView.

`umnetcurrent` Permette l'accesso allo stack messo a disposizione da sistema operativo host. Questo modulo è in particolare utile per bypassare eventuali overloading effettuati da `umnetlink` e `umnetnull`.

`umnetlink` Il funzionamento di questo modulo è simile a quello dei link simbolici in un filesystem standard UNIX. Stabilisce un ulteriore file associato ad uno stack ed a cui possono essere applicati dei filtri in fase di mount.

`umnetlwipv6` Questo sottomodulo crea uno stack LWIPv6 in userspace e lo associa al file indicato da riga di comando. Questo secondo stack risulta completamente indipendentemente da quello standard di sistema e può essere interamente configurato dall'utente. In particolare può (comportamento di default) connettersi ad uno `vde_switch` in esecuzione ed accedere così alla rete virtuale configurata su tale device virtuale.

`umnetnull` È un semplice modulo, didattico per lo più, che nega tutte le funzionalità di rete.

Capitolo 5

Il kernel rump

Uno dei requisiti da soddisfare durante la scrittura del kernel rump e l'implementazione dell'architettura anykernel era quello di mantenere al minimo le modifiche effettuate al kernel preesistente.

La struttura monolitica di NetBSD fa affidamento sull'inclusione dinamica delle componenti, che avviene in fase di linking. Per questo è necessario procedere all'inclusione ed alla riscrittura delle componenti rispettando una granularità a livello di file sorgente.

Nel suo lavoro Antti Kantee ha seguito due differenti tecniche per assicurarsi questo risultato:

Estrazione Dove possibile questo procedimento è sempre stato preferito e consiste nel riutilizzo del codice già presente nel kernel standard. Per sfruttare al meglio le componenti presenti in certi casi sono state comunque necessarie alcune modifiche, per esempio quando funzionalità appartenenti a factions differenti apparivano nello stesso file. In fase di compilazione del kernel rump, questi sorgenti vengono direttamente inclusi dalla loro posizione originale (`/sys/`) nel tree del codice NetBSD, senza ulteriori modifiche.

Implementazione Nel caso di funzionalità non supportate nel kernel regolare o con semantica diversa rispetto al kernel rump, la via seguita è

Funzionalità	Libreria
Hypervisor rump	`librumpuser`
Libreria base	`librump`

Librerie specifiche per le factions	`librumdev`
	`librumpnet`
	`librumpvfs`

Librerie specifiche per i driver	`librump<faction>_<driver>` `librumpnet_inet`, `librumpdev_cgd`, .. La componente `<faction>` *non* indica una dipendenza stretta, ma solamente la faction di riferimento

Tabella 5.1: Alcune librerie fornite da rump

stata quella della (re-)implementazione. Il codice scritto appositamente per rump si trova nella sotto-directory `/sys/rump/` di NetBSD

Per minimizzare l'overhead introdotto ed il peso in memoria, le componenti di rump sono suddivise in librerie dinamiche modulari (tabella 5.1) lasciando all'utente la scelta di quali funzionalità includere a runtime al fine di garantire il corretto funzionamento dell'applicazione virtualizzata.

Nell'estrarre il codice dal kernel per renderlo eseguibile in userspace su qualsiasi host POSIX sono emersi due problemi particolari. Il primo è dovuto al conflitto dei simboli nel namespace C per cui una chiamata a `printf()`, per esempio, deve esser risolta su due simboli differenti a seconda che sia richiamata da un'applicazione (`printf()` fornita da `libc`) o dal kernel (implementazione nel kernel). Per ovviare a questo problema, Kantee ha adottato una tecnica di renaming sintattico dei simboli effettuato con `objcopy`: ogni simbolo utilizzato all'interno delle librerie rump il cui nome non inizi con `rump` o `RUMP` viene rinominato aggiungendo il prefisso `rumpns_` . Questa soluzione garantisce che le chiamate generate dal kernel rump siano sempre gestite internamente, senza però intercettare anche le chiamate provenienti da applicazioni esterne.

Il secondo problema è dovuto all'incompletezza dello standard POSIX, per cui anche se `stat(const char *path, struct stat *sb)` è definita e

standard, così non è per la rappresentazione binaria della `struct stat`. Questo secondo aspetto ha richiesto l'introduzione dell'hypervisor (`librumpuser`) il cui scopo è facilitare la procedura di porting del kernel rump su altri host POSIX.

La configurazione iniziale di un kernel rump dipende da quali moduli si sono linkati in fase di *bootstrap*. L'implementazione fornisce supporto sia al linking delle librerie in fase di creazione degli eseguibili, sia l'utilizzo di `dlopen()` per il caricamento a runtime. Ogni componente, inclusa la base, può essere caricato in questo modo, garantendo la massima flessibilità per lo sviluppo di applicativi che utilizzino i rump kernel.

5.1 Networking di rump

L'accesso diretto ai driver dei device di rete, necessario per l'invio di pacchetti raw, è un'operazione privilegiata e quindi non concessa ad un kernel rump in esecuzione come utente. I permessi disponibili a tutti prevedono che l'accesso avvenga tramite la Berkeley socket API e che il traffico a basso livello venga gestito dallo stack di rete del sistema operativo host. Nell'implementazione di rump sono supportati tre modelli di accesso allo stack ed alla rete host differenti e rappresentati in figura 5.1.

- **Stack di rete completo con accesso raw alla rete host**
  Lo standard per questo tipo di utilizzo è rappresentato dall'uso dei device file-like `tap`. I device di questo tipo possono essere configurati in bridge con i device (fisici o virtuali) già presenti nel sistema operativo host e rendere quindi disponibile al kernel rump l'accesso raw alla rete. L'utilizzo di `tap` richiede però i permessi di amministratore. **libvirtif** è una libreria fornita con rump che permette la creazione e la gestione trasparente di interfacce `tap` e rappresenta anche il modello di riferimento utilizzato per la creazione di `libvdeif` (sezione 6.2).

- **Stack di rete completo senza accesso alla rete host**
  Per ovviare al problema dei permessi richiesti dall'utilizzo dei device

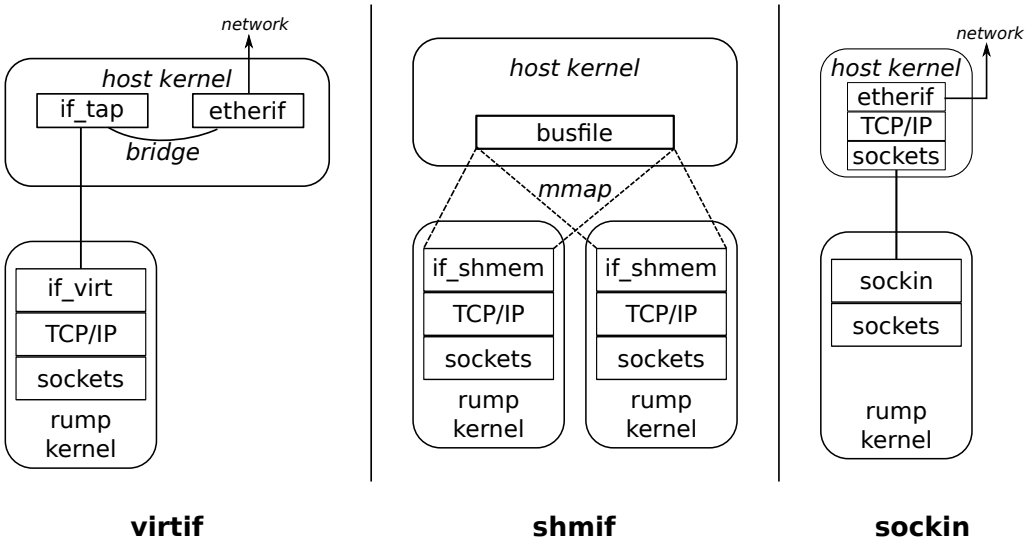


Figura 5.1: Networking per rump

`tap`, è stato implementata **shmif** un'interfaccia virtuale basata su un file mappato in memoria e condiviso tra i processi nel ruolo di bus ethernet. Questa implementazione (discussa anche in 6.1) si presta bene alla fase di test e sviluppo, ma soffre di alcune evidenti limitazioni che la rendono inadatta all'utilizzo in produzione.

- **Accesso utente alla rete host**
  Volendo evitare di ricorrere a Slirp, Kantee ha implementato **sockin** una soluzione alternativa che eviti il doppio processamento dei pacchetti da parte dei due stack di rete (guest e host). Il protocollo `sockin` sostituisce il dominio AF_INET e mappa i comandi definiti dall'utente direttamente sulle socket call. Si tratta però di un'implementazione mutualmente esclusiva rispetto allo stack standard TCP/IP e l'accesso raw viene comunque eseguito dallo stack del sistema host e non dal kernel rump.

Nel capitolo 6 si vedrà come la libreria `libvdeif`, parte del lavoro di questa tesi, offra un nuovo strumento per connettere i kernel rump ad una rete

virtuale VDE e grazie agli strumenti a disposizione permetta la realizzazione di tutte le modalità sopra descritte.

5.2 Client rump remoti

Per client rump si intende qualsiasi applicazione che richieda dei servizi ad un kernel rump. Come introdotto nel capitolo 3 le modalità con cui un client può accedere ad un server rump sono tre:

Client locale Il kernel rump, avviato invocando `rump_init()` di `librump`, viene incluso come libreria dinamica all'interno del processo. Offre le prestazioni migliori e consente l'accesso alle routine interne di librump ma obbliga l'utente a provvedere alla completa configurazione del kernel rump come parte nella logica dell'applicazione. Nel caso oggetto di questa tesi, ovvero l'esecuzione di uno stack di rete, questo comporterebbe l'inserimento di tutto il codice necessario alla configurazione delle interfacce, del routing, etc.

Client remoto Rispetto all'applicazione in esame, il kernel rump risiede in un processo esterno eseguito sullo stesso host o su uno diverso. Il bootstrap del server avviene tramite la routine `rump_init_server()` mentre il client è implementato nella libreria `librumpclient` e viene inizializzato da `rumpclient_init()`.

Microkernel Le chiamate vengono gestite dal kernel host che si occupa di inoltrarle a un kernel rump residente in un processo differente e che fornisce il servizio richiesto. La logica è simile a quella del client remoto ma in questo caso il routing delle system call non viene gestito dall'applicazione client, ma dal microkernel host.

L'utilizzo di kernel rump remoti è la modalità più interessante per gli scopi di questo lavoro e garantisce la flessibilità necessaria per l'utilizzo degli strumenti disponibili. La comunicazione tra client e server può avvenire

tramite socket UNIX o su rete TCP/IP e, anche se non sono previsti sistemi automatici di ricerca dei kernel rump disponibili, l'uso della variabile d'ambiente RUMP_SERVER rappresenta una soluzione anche per le applicazioni senza supporto specifico e che sfruttano `librumphijack`.

L'implementazione di `librumpclient` supporta applicazioni sia singlethread che multithread e le operazioni di sincronizzazione avvengono all'interno della libreria. La connessione tra client e server non è però condivisibile tra più processi e per questo libreria e protocollo di comunicazione includono un supporto speciale per alcune syscall. Nel caso della `fork()`, per esempio, per garantire il mantenimento dei filedescriptor aperti, memorizzati nel server rump, viene stabilita un associazione basato su *cookie* tra processo padre e figlio anche se i canali di comunicazione utilizzati sono differenti.

Un esempio di system call remota è riportata nella tabella 5.2.

5.3 Portabilità e compilazione di rump

La portabilità di rump dipende dai conflitti che possono emergere in corrispondenza delle due interfacce esposte:

- **L'interfaccia verso il sistema host**: definita dalle hypercall, questa interfaccia è abbastanza contenuta ma soprattutto di nuova implementazione e quindi liberamente strutturabile. Per evitare problemi di portabilità tra sistemi POSIX, Kantee ha definito le hypercall utilizzando solo tipi chiaramente standardizzati e definiti su tutti gli host.

- **L'interfaccia verso i client**: dato il namespace piatto del linguaggio C e l'impossibilità di ridefinire i tipi senza incorrere in conflitti, la soluzione adottata in questo caso è parziale e "sintattica". Lo script `sys/rump/include/rump/makerumpdefs.sh` è stato incluso nella procedura di compilazione con il compito di creare l'header `rumpdefs.h` tramite una sequenza di comandi `sed`. Così facendo le definizioni utili di macro e costanti vengono estratte dall'albero dei sorgenti di NetBSD e rinominate aggiungendo il prefisso `RUMP_` per il loro successivo utilizzo.

Syscall diretta	Syscall rump remota
`open(/dev/null,O_RDWR)`	`rump_sys_open(/dev/null,O_RDWR)`
libc esegue il trap	`librumpclient` effettua il *marshalling* degli argomenti e trasmette la richiesta di syscall sul canale di comunicazione. Il thread chiamante viene sospeso in attesa del ritorno della syscall
l'handler chiama `sys_open()`	il kernel rump riceve la richiesta, l'associa al thread, lo shedula e procede alla chiamata di `sys_open()`
la routine di lookup richiama `copyinstr()`	la routine di lookup invia una richiesta di `copyinstr()` al client, il client risponde il pathname, il server riceve la risposta e la copia in un buffer locale
la routine di lookup ritorna ed alloca il file descriptor collegato al file `/dev/null`	la routine di lookup ritorna ed alloca il file descriptor collegato al file `/dev/null`
la system call setta l'`errno` e restituisce il valore di ritorno	il server spedisce il valore di ritorno e l'`errno` al client, questo sveglia il thread sospeso in attesa che setta l'errno e restituisce il valore di ritorno

Tabella 5.2: Confronto tra system call locale e remota

La compilazione di rump estende le sue dipendenze in profondità nei sorgenti di NetBSD. Il kernel rump dipende infatti dalla libreria `rumpuser` che implementa le hypercall e che a sua volta dipende da `libc`. Per questo si rende necessaria la compilazione di parte delle librerie userspace di NetBSD a prescindere da quali saranno i moduli di rump utilizzati successivamente.

Fortunatamente insieme ai sorgenti di NetBSD viene distribuito build.sh [19], un tool prezioso per la (cross)compilazione del kernel e del sistema base. Grazie a questo strumento è possibile effettuare la build su una qualsiasi architettura supportata per qualsiasi altra architettura *target* con un singolo comando: il tool si occupa di creare tutto l'ambiente necessario, compresa l'indispensabile toolchain BSD [1].

5.3.1 `buidrump.sh`

Al fine di semplificare la compilazione dei componenti rump su e per piattaforme diverse da NetBSD sono forniti alcuni script, di cui `buildrump.sh` è il principale [16]. In linea di principio `build.sh` sarebbe sufficiente per svolgere il lavoro necessario, ma l'uso diretto richiederebbe una conoscenza approfondita delle procedure interne di compilazione di NetBSD. Per questa ragione Kantee ha realizzato `buildrump.sh` che, utilizzando anche altri script esterni, si occupa di:

- scaricare tutti e i soli sorgenti necessari dal tree di NetBSD;
- compilare tutta la toolchain (BSDmake, ...);
- individuare le opzioni da passare al compilatore ed al linker per una corretta esecuzione dei passi successivi. Infatti, nel codice di rump sono molte le direttive per preprocessore che fanno affidamento sulle definizioni che avvengono a questo livello ("`-Dlinux`", etc...);
- configurare i percorsi di destinazione di header, librerie ed eseguibili.

[1] La corretta compilazione del sistema base NetBSD richiede l'utilizzo di BSDmake, incompatibili con GNU Make

```
Usage: ./buildrump.sh [-h] [options] [command] [command...]
supported options:
 -d: location for headers/libs.  default: PWD/rump
 -T: location for tools+rumpmake.
 -s: location of source tree.
 -k: only kernel components (no hypercalls).
 -N: emulate NetBSD, set -D__NetBSD__ etc.
supported commands (default => checkout+fullbuild+tests):
 checkoutgit:   fetch NetBSD sources to srcdir from github
 checkoutcvs:   fetch NetBSD sources to srcdir from anoncvs
 checkout:      alias for checkoutgit
 tools:         build necessary tools to tooldir
 build:         build everything related to rump kernels
 install:       install rump kernel components into destdir
 tests:         run tests to verify installation is functional
 fullbuild:     alias for "tools␣build␣install"
```

Figura 5.2: Usage (estratto) di buildrump.sh

- compilare ed installare i programmi di test ed esempio.

Gli avanzamenti avvenuti in questo tool sono stati fondamentali per mantenere lo sforzo ed i tempi di sviluppo ragionevoli. Basti pensare che nelle prime versioni era necessario scaricare l'intero tree di NetBSD per capire il risparmio di risorse realizzato con questo strumento.

5.3.2 `rumprun` e `rumpremote`

Il codice estratto dal kernel NetBSD ed in esecuzione su host differenti fa affidamento sull'hypervisor e l'unica libreria linkata direttamente verso l'ambiente host è `librumpuser`. Questo crea un ambiente omogeneo all'interno delle librerie rump, per cui le factions ed i driver possono eseguire non modificati e con la certezza di utilizzare dati e funzioni compatibili.

Diversamente, compilando del codice C scritto per NetBSD e includendo e linkando solo header e librerie compatibili, quel che si ottiene è un binario che non può essere eseguito in un sistema diverso da NetBSD stesso.

```
objcopy --redefine-syms=extra.map ${OBJDIR}/tmp1_${2}.o
objcopy --redefine-syms=rump.map ${OBJDIR}/tmp1_${2}.o
objcopy --redefine-syms=emul.map ${OBJDIR}/tmp1_${2}.o
```

(a) Esempio dell'utilizzo di `objcopy` nel `Makefile` fornito con `rumprun`

```
read    rumprun_read_wrapper
write   rumprun_write_wrapper
open    rump___sysimpl_open
close   rump___sysimpl_close
link    rump___sysimpl_link
unlink  rump___sysimpl_unlink
chdir   rump___sysimpl_chdir
fchdir  rump___sysimpl_fchdir
```

(b) Estratto di `rump.map`

Figura 5.3: La manipolazione sintattica di `rumprun`

Per risolvere questo problema è nato il progetto `rumprun`, un wrapper di `buildrump.sh` il cui obiettivo è creare un ambiente in cui si possano eseguire degli strumenti standard NetBSD ridirigendo tutte le system call non compatibili verso un kernel rump in esecuzione.

La tecnica utilizzata è quella delle manipolazione sintattica dei simboli, suddivisa in due step:

- utilizzando `objcopy` vengono rinominate le system call nel loro corrispondente rump (fig. 5.3);
- il file oggetto risultante viene caricato dinamicamente da un wrapper minimale, `rumprun` o `rumpremote`, che ridirige le system call verso un kernel rump in esecuzione rispettivamente nel processo locale o in uno remoto.

Questi strumenti rendono la configurazione di un kernel rump facile quanto quella del kernel NetBSD ed evitano all'utente l'overhead notevole di dover provvedere alla configurazione scrivendo appositamente del codice all'interno dei propri applicativi (fig. 5.4).

```
$ rumprun ifconfig
lo0: flags=8049<UP,LOOPBACK,RUNNING,MULTICAST> mtu 33648
  inet6 ::1 prefixlen 128
  inet6 fe80::1%lo0 prefixlen 64 scopeid 0x1
  inet 127.0.0.1 netmask 0xff000000
rump kernel halting...
syncing disks... done
unmounting file systems...
unmounting done
halted
```

Figura 5.4: L'esecuzione di `ifconfig` per NetBSD grazie a `rumprun`

Capitolo 6

Lo stack NetBSD/TCP-IP in UMView

L'obiettivo finale di questi esperimenti è la possibilità di eseguire uno stack di rete completo e ampiamente testato come quello di NetBSD all'interno dell'ambiente parzialmente virtualizzato fornito da UMView.

Analizzando la struttura di UMView nel capitolo 4 si è già visto come le funzionalità di networking siano un'eccezione alla convenzione "tutto è un file" vigente in UNIX e come questo renda difficile la gestione di interfacce e stack di rete da parte di un utente senza privilegi. L'estensione della semantica della Berkeley socket API realizzata da `msocket` rende possibile il superamento di questo limite e lo sviluppo di `umnetbsd` fornisce lo stack alternativo configurabile a livello utente.

Il lavoro si basa sui risultati ottenuti da Antti Kantee nel dimostrare la possibilità di eseguire lo stack di rete NetBSD in userspace [14].

Le sperimentazioni iniziali, paragrafo 6.1, sono avvenute sfruttando l'interfaccia `shmif` vista in 5.1. Alcuni strumenti sono stati sviluppati per accedere dal sistema host Linux al bus ethernet utilizzato con particolare attenzione alle funzionalità di debug.

Successivamente si è passati all'analisi dell'implementazione di `virtif`, già presentata nel capitolo 5.1. La libreria `libvirtif` è stata quindi mo-

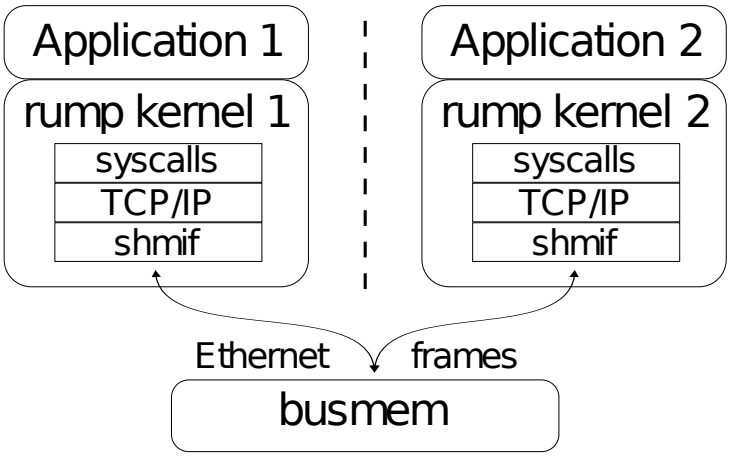


Figura 6.1: Funzionamento di `shmif`

dificata affinché fosse possibile collegarsi direttamente ad uno switch VDE invece di creare un device `tap`. Il risultato è stato ottenuto con `libvdeif`, presentata in 6.2.

Infine in 6.3 si presenta `umnetbsd`, il sottomodulo di `umnet` che realizza l'integrazione tra UMView e rump, risultato principale di questo lavoro di ricerca.

6.1 Primi tentativi: `libshmif` per Linux

La necessità di avere un'interfaccia virtuale configurabile dall'utente, accessibile in modalità raw e che potesse comunicare con altri kernel rump ha portato alla creazione di `shmif`. In questa interfaccia il canale di comunicazione ethernet è emulato da un file mappato in memoria ed interfacce associate allo stesso file si comportano come se fossero collegato allo stesso segmento di rete.

Il file che simula il canale ethernet è detto *buspath* ed è mappato come indicato nella figura 6.2. L'area di memoria puntata da `shm_data` realizza un buffer circolare in cui vengono trascritti i pacchetti ethernet esattamente come si trattasse di un device `tap`. L'accesso condiviso tra più interfacce viene arbitrato dalla variabile `shm_lock` tramite le operazioni atomiche di tipo *compare-*

```
struct shmif_mem {
        uint32_t shm_magic;
        uint32_t shm_version;
        uint64_t shm_gen;
        uint32_t shm_first;
        uint32_t shm_last;
        uint32_t shm_lock;
        uint32_t shm_spare[1];
        uint8_t  shm_data[0];
};
```

Figura 6.2: Struttura del canale condiviso shmif

and-swap messe a disposizione dal kernel host (__sync_val_compare_and_swap su Linux, atomic_cas_32 su NetBSD). Per ricevere notifica di nuovo traffico disponibile sul canale vengono utilizzate i sottosistemi del kernel inotify() o kqueue() sfruttando la duplice natura del file mappato in memoria.

In NetBSD il supporto per shmif è stato inserito non solo in rump, ma anche nel sistema base. Questo significa che gli strumenti standard di configurazioni quali ifconfig [1] supportano la nuova opzione necessaria linkstr mentre in Linux questo supporto non esiste.

Il codice del driver shmif presente nel source tree di NetBSD è perfettamente portabile in quanto parte del kernel rump ma, come discusso nel capitolo 5.3, la sua compilazione richiede particolare attenzione. Per ovviare a questi problemi attuando al contempo una separazione più netta delle componenti funzionali della libreria, le parti rilevanti di codice sono state estratte per essere incluse nel processo di compilazione di due strumenti usati nella prima parte dello sviluppo, shmif_dumpbus_live e vde_plug2shmif. Il primo permette di effettuare il dump in formato pcap [12] del traffico in transito sul *buspath*, mentre il secondo permette di collegare un interfaccia shmif ad un vde_switch.

Per quanto questa interfaccia risulti funzionale alcuni aspetti la rendono problematica dal punto di vista delle performance:

[1] http://mail-index.netbsd.org/tech-net/2010/11/09/msg002335.html

```
$ vde_plug2shmif
Usage: vde_plug2shmif [OPTION]... shmif_pathname
  -p, --port=portnum          Port number in the VDE switch
  -s, --sock=socket           VDE switch control socket or dir
```

Figura 6.3: Estratto dall'help del comando `vde_plug2shmif`

- se un processo termina (o crasha) mentre detiene il lock del buffer nessun processo è più in grado di accedervi e la comunicazione è bloccata;

- il doppio livello di indirezione imposto per la comunicazione tra uno stack TCP-IP attivo in un kernel rump e il `vde_switch`: un singolo pacchetto deve infatti essere scritto sul buffer circolare per poi essere letto da un'istanza di `vde_plug2shmif` che lo inoltrerà al `vde_switch`;

- la segnalazione del traffico tramite i sottosistemi `inotify` e `kqueue`: sia in Linux che in NetBSD questi sottosistemi non sono in grado di segnalare la modifica avvenuta in memoria di un file mappato. Affinché la segnalazione avvenga è necessario effettuare una `write()` sul file utilizzando il file descriptor associato. La soluzione adottata prevede di accorpare un unica `write()` alla fine di ogni trasmissione costituita da più pacchetti, ma rimane evidente come un accesso a disco con conseguente context switch risulti più lento del semplice accesso alla memoria.

6.2 La libreria `libvdeif`

Per risolvere i problemi introdotti con gli strumenti descritti sopra si è preferito quindi scrivere una versione modificata di `libvirtif` che faccia uso di `libvdeplug` per connettersi direttamente ad un `vde_switch`.

Grazie all'uso massiccio di macro nel codice di rump ed in genere di NetBSD (fig. 6.4) , l'implementazione è risultata veloce e contenuta: l'intero file `rump/sys/rump/net/lib/libvdeif/rumpcomp_user.c` conta 190 righe

comprensive di commenti e spazi bianchi. Un esempio è dato in figura 6.5, dove si può notare come la logica specifica di VDE sia ridotta alla chiamata della funzione `vde_open()`.

In generale il compito di questa libreria è stabilire un ponte tra il layer ethernet del kernel rump (`sys/net/if_ethersubr.c` nei sorgenti di NetBSD) ed un `vde_switch`. La corrispondenza tra le funzioni risulta pressoché biunivoca e, per esempio, nella funzione `vdeif_receiver()`, se si escludono controllo degli errori e meccanismi di lock, un'invocazione di `VIFHYPER_RECV()` è seguita banalmente da `ether_input()`, ovvero la procedura di inoltro dei pacchetti ethernet al driver corrispondente.

L'utilizzo di `libvdeplug` nello sviluppo di questa libreria ha comportato l'inclusione dell'header `libvdeplug.h` nei sorgenti di `libvdeif` introducendo così una dipendenza a compile time tra rump e VDE. Fortunatamente `libvdeplug.h` a sua volta include solamente `sys/types.h`, presente anche nell'albero dei sorgenti di NetBSD e pienamente compatibile con lo standard POSIX.

Il supporto introdotto da `libvdeif` apre nuovi scenari di sperimentazione con rump in Linux, rendendo disponile ai kernel rump una connessione diretta con l'ambiente di networking virtuale di VDE. Unendo gli strumenti offerti dal framework V^2 è quindi possibile accedere alla rete esterna reale utilizzando lo stack di rete NetBSD:

```
$ export RUMP_SERVER=unix:///tmp/rump_alone
$ export VDE_SOCKET=/tmp/vde0
$ vde_switch -s ${VDE_SOCKET} -m 666 -d
$ slirpvde -s ${VDE_SOCKET} -H 10.9.2.254/24 -N 10.9.2.253 -d
$ rump_allserver ${RUMP_SERVER}
$ rumpremote ifconfig vde0 create
$ rumpremote ifconfig vde0 10.9.2.60 netmask 0xffffff00
$ rumpremote ifconfig vde0 up
$ rumpremote route add default 10.9.2.254
$ nettest_simple rump_httpclient $RUMP_SERVER 46.246.93.70
Client, using RUMP_SERVER unix:///tmp/rump_alone
```

```
RUMPCOMP_USER_CPPFLAGS+= -DVDEIF_BASE=vde
```

(a) Estratto da `rump/sys/rump/net/lib/libvdeif/Makefile`

```
#ifndef VDEIF_BASE
#error Define VDEIF_BASE
#endif
#define VIF_CONCAT(x,y) x##y
#define VIF_CONCAT3(x,y,z) x##y##z
#define VIF_BASENAME(x,y) VIF_CONCAT(x,y)
#define VIF_BASENAME3(x,y,z) VIF_CONCAT3(x,y,z)
#define VIFHYPER_CREATE VIF_BASENAME3(rumpcomp_,
   VDEIF_BASE, _create)
```

(b) Estratto da `/rump/sys/rump/net/lib/libvdeif/if_vde.h`

Figura 6.4: Utilizzo di macro parametriche in rump

```
Received:
<html><body>
<h3>paolo.perfetti.name</h3>
<p>Welcome on coppelia!</p>
</body></html>
```

Ovviamente per uscire dall'host non si può evitare di utilizzare entrambi gli stack di rete, quello di NetBSD messo a disposizione da rump e quello del sistema host, con `slirpvde` nel ruolo di router connesso al `vde_switch`.

6.3 Il modulo `umnetbsd`

Gli strumenti elencati sin qui ci pongono ad un solo passo di distanza dalla possibilità di eseguire applicazioni non modificate in UMView utilizzando lo stack TCP-IP di NetBSD per accedere ad una rete VDE. L'anello mancante è rappresentato da un sottomodulo di `umnet` che intercetti le socketcall richieste dalle applicazioni e le mappi sulle syscall `rump_sys_*` messe a disposizione dal kernel rump.

```
int VIFHYPER_CREATE(int devnum, struct vdeif_user **viup) {
        struct vdeif_user *viu = NULL;
        void *cookie; int rv;

        #define SOCKETNAME "/tmp/vde0"
        cookie = rumpuser_component_unschedule();
        viu = malloc(sizeof(*viu));
        if (viu == NULL)
          { rv = errno; goto out; }

        viu->vde_socketname = malloc(sizeof(SOCKETNAME));
        strncpy(viu->vde_socketname, SOCKETNAME, \
          sizeof(SOCKETNAME));
        (viu->open_args).port=0;
        (viu->open_args).group=NULL;
        (viu->open_args).mode=0700;

        viu->vde_conn = vde_open(viu->vde_socketname, \
          "rumpvde:", &viu->open_args);

        if (viu->vde_conn == NULL) {
                rv = errno;
                free(viu);
                goto out;
        }
        viu->vde_dying = 0;
        rv = 0;
  out: rumpuser_component_schedule(cookie);
       *viup = viu;
       return rumpuser_component_errtrans(rv);
 }
```

Figura 6.5: Estratto di libvdeif/rumpcomp_user.c

Il modulo `umnetbsd` implementa esattamente questa funzionalità: basandosi sullo strato di intercettazione già previsto da `umnet` ridefinisce le syscall ed inoltra le richieste al kernel rump. Una volta caricato `umnetbsd` e utilizzando `mstack` per redirigere le system call generate, un applicazione può accedere allo stack di rete NetBSD incluso nel kernel rump eseguito senza bisogno di alcuna modifica.

Al pari degli altri sottomoduli, `umnetbsd` è implementato in forma di libreria dinamica il cui nome corrisponde al *filesystemtype* passato all'opzione "-t" del comando `mount`. Contestualmente è possibile definire dei filtri sui protocolli accessibile così come discusso nel capitolo 4.

Per registrare le funzioni interne al modulo da invocare quando avviene all'intercettazione di una system call si inizializza una struttura `struct umnet_operations` in cui viene definita la corrispondenza desiderata (figura 6.6). Le funzioni così definite vengono richiamate dal modulo `umnet` che si occupa di mantenere l'associazione tra mountpoint e relativo stack di rete.

6.3.1 L'inizializzazione del modulo: `umnetbsd_init`

La funzione di inizializzazione di `umnetbsd_init` si suddivide in quattro parti:

1. Avvio di un kernel rump (`rump_init()`) come *application library* interna di UMView.

2. Avvio di un server rump (`rump_init_server()`) per la gestione di system call provenienti da client remoti.

3. Apertura di una pipe tramite `rump_sys_pipe()` per la comunicazione tra il main thread di UMView ed il thread dedicato a rump. Questa pipe è utilizzata dalla funzione `umnetbsd_event_subscribe()` per la segnalazione al thread `umnetbsd_rw` di nuove richieste provenienti dall'applicazione.

```
struct umnet_operations umnet_ops={
 .init=umnetbsd_init,
 .fini=umnetbsd_fini,
 .event_subscribe=umnetbsd_event_subscribe,
 .msocket=umnetbsd_msocket,
 .bind=umnetbsd_bind,
 .listen=umnetbsd_listen,
 .accept=umnetbsd_accept,
 .connect=umnetbsd_connect,
 .read=umnetbsd_read,
 .write=umnetbsd_write,
 .close=umnetbsd_close,
 .ioctl=umnetbsd_ioctl,
 .getsockname=umnetbsd_getsockname,
 .getpeername=umnetbsd_getpeername,
 .sendmsg=umnetbsd_sendmsg,
 .recvmsg=umnetbsd_recvmsg,
 .getsockopt = umnetbsd_getsockopt,
 .setsockopt = umnetbsd_setsockopt,
 .supported_domain = umnetbsd_supported_domain,
};
```

Figura 6.6: La struttura `umnet_ops`

4. Creazione del thread `umnetbsd_rw`, responsabile della gestione delle system call bloccanti dirette a file descriptor interni al kernel rump.

Per rendere comoda e flessibile la configurazione delle interfacce di rete si è scelto di utilizzare il kernel rump sia come libreria dinamica caricata dal processo, sia come server remoto. Così facendo le system call intercettate da UMView possono richiamare direttamente le corrispondenti messe a disposizione da rump senza incorrere nell'overhead determinato dalla comunicazione attraverso socket UNIX. Al contempo gli strumenti di configurazione, `ifconfig` per esempio, possono essere eseguiti tramite `rumpremote` come processi esterni garantendo all'utente la configurabilità del servizio a runtime (si veda la sezione 5.3.2).

6.3.2 La gestione asincrona degli eventi con `event_subscribe`

L'implementazione single-thread di UMView richiede un'accurata gestione degli eventi asincroni generati dallo stack di rete sottostante, indipendentemente dal fatto che si tratti di uno stack NetBSD, LWIPv6 o del sistema host. La soluzione adottata dal team V^2 fa affidamento sulla funzione `umnet_event_subscribe()` che richiama la funzione specifica definita per il sottomodulo con i seguenti parametri:

cb puntatore ad una funzione di *callback* da invocare al verificarsi di uno degli eventi monitorati;

arg puntatore agli argomenti della syscall originale chiamata dall'applicazione;

fd file descriptor oggetto della chiamata;

how maschera degli eventi interessanti, la cui semantica è la stessa definita per l'omonimo argomento della `poll()`.

La funzione `umnet_event_subscribe()` viene invocata prima di ogni system call bloccate per evitare che l'unico thread in esecuzione (e che quindi è responsabile anche del main loop degli eventi) si blocchi in attesa del verificarsi di qualche condizione. La logica della funzione è riassunta nel pseudo-codice di figura 6.7.

Questo metodo rende possibile la gestione di processi multipli in attesa di eventi diversi senza per questo bloccare l'esecuzione dell'intero core di UMView e conseguentemente di ogni programma virtualizzato al suo interno.

6.3.3 Il thread `umnetbsd_rw`

La creazione di un apposito thread è in contrasto con il paragrafo precedente e rappresenta una significativa eccezione rispetto all'attuale struttura single-thread di UMView. Questa soluzione si è resa necessaria a partire dall'osservazione che i file descriptor interni al sistema host e quelli creati

```
if ( umnetbsd_watch_shared->cb == NULL ) {
  if ( umnetbsd_watch_shared->arg == NULL ) {
    // testa quali eventi sono gi\'a avvenuti e
       ritorna
  } else{  // umnetbsd_watch_shared->arg != NULL
    // cancella le richieste pendenti sull'fd
} else { //  umnetbsd_watch_shared->cb != NULL
  if (rump_poll_rv == 0 ) {
    // aggiungo l'fd alla lista degli fd monitorati
  } else { //rump_poll_rv != 0
    // restituisco il risultato tornato dalla poll()
  }
}
```

Figura 6.7: Il funzionamento della `event_subscribe`

all'interno del kernel rump esistono in due domini completamente disgiunti. In particolare non può esistere una syscall che svolga lo stesso compito della `select()` o della `poll()` e che riesca a monitorare al contempo i due set di file descriptor.

Sicuramente sarebbe possibile definire una funzione che si occupi di associare ogni file descriptor nel dominio "virtuale" ad uno nel dominio "reale" e questo in effetti già avviene nel core di UMView. L'associazione così stabilita non rappresenterebbe però una soluzione al problema, in quanto richiederebbe comunque la creazione di un thread che rimanga in ascolto nel *dominio rump* per poi segnalare gli eventi sui filedescriptor corrispondenti nel *dominio host*. Inoltre realizzerebbe una sconveniente duplicazione delle funzionalità già implementati in altre componenti di UMView.

Il punto di contatto trai i due domini viene stabilito dalla pipe dedicata alla comunicazione tra i thread. Il thread `umnetbsd_rw` rimane in ascolto sul lato in lettura della pipe attendendo la segnalazione di nuove chiamate da gestire che avviene tramite l'invio di un singolo carattere predefinito. Questo canale di comunicazione viene infatti utilizzato solamente per la segnalazione di nuove chiamate e l'effettivo scambio di dati avviene tramite delle variabili condivise il cui accesso è arbitrato da meccanismi di lock standard (figura

6.8).

Nel modulo `umnetbsd` la funzione `umnetbsd_event_subscribe()` si occupa solamente di segnalare l'esecuzione di una nuova system call al thread `umnetbsd_rw` che si incarica di implementare la logica già spiegata nella sezione 6.3.2.

```
int umnetbsd_event_subscribe (voidfun cb, void *arg,
   int fd, int how) {
  /* dichiarazioni, gestione dei parametri,... */
  pthread_mutex_lock(&umnetbsd_rt_data.mutex);
  umnetbsd_watch_shared = umnetbsd_watch_new;
  umnetbsd_watch_shared_rv = 0;
  rump_sys_write(umnetbsd_cfd,"S",1);
  pthread_cond_wait(&umnetbsd_rt_data.cv,
     &umnetbsd_rt_data.mutex);
  rv = umnetbsd_watch_shared_rv;
  pthread_mutex_unlock(&umnetbsd_rt_data.mutex);
  return rv;
}
```

Figura 6.8: La sincronizzazione tra thread

6.3.4 I wrapper per le system call

La Berkeley socket API per il networking rappresenta lo standard per la gestione dei socket e da questo deriva una corrispondenza biunivoca tra le API Linux e NetBSD. Tutte le socketcall intercettate hanno la stessa semantica nei due stack e non richiedono una particolare logica di conversione, se non nella gestione dei valori delle macro definite per i flags e per le opzioni disponibili. Le situazioni possibili sono tre:

- Macro (o opzione o flag) presente in NetBSD ma non in Linux;
- Macro (o opzione o flag) presente in Linux ma non in NetBSD;
- Macro (o opzione o flag) presente in entrambi i sistemi ma con valori diversi.

```
#define RUMP2LINUX_ERRNO(err) rump2linux_errno(err)
#define R2L_AUX(a) a;
#define RETURN2LINUX(rv, F) \
   rv = R2L_AUX(F); if (rv < 0) \
   errno = RUMP2LINUX_ERRNO(errno);\
   return rv;

int rump2linux_errno(int n_errno) {
   switch(n_errno){
   case RUMP_EAGAIN: return EDEADLK;
   case RUMP_EDEADLK: return EAGAIN;
   case RUMP_EINPROGRESS: return EINPROGRESS;
   case RUMP_EALREADY: return EALREADY
   ....
```

Figura 6.9: Esempio di macro di conversione

Un esempio dell'ultimo caso è dato dalla famiglia di protocolli AF_INET6 macro il cui valore è 24 in NetBSD mentre in Linux è 10.

Per far fronte a queste incompatibilità sono state implementate alcune funzioni e macro (figura 6.9) dedicate ad effettuare le conversioni. L'overhead introdotto da queste conversioni è minimo ed ininfluente rispetto alla complessità dei programmi e trattandosi di operazioni a runtime non è possibile evitarlo. È invece significativo l'impegno necessario in fase di programmazione per elencare tutte le macro che necessitano una conversione e mantenere coerenti ed aggiornate le definizioni. Come già visto nel caso di rump sarebbe auspicabile una manipolazione sintattica a tempo di compilazione per evitare problemi dovuti alla difficoltà di mantenimento di procedure banali ma lunghe e ripetitive.

Lo stesso problema si presenta con i valori assegnati alla variabile `errno` per cui lo standard POSIX definisce il nome ed il significato delle macro ma non i valori corrispondenti che risultano differenti tra Linux e NetBSD. Di conseguenza si ha che se la variabile `errno` ritornata da una syscall `rump_*` viene interpretata dalle routine `perror()` o `strerror()` del sistema host si rischia di restituire un'informazione sbagliata fuorviante. Oltre a richiedere

una gestione metodica per tornare all'applicazione virtualizzata un'informazione consistente, questo aspetto ha richiesto particolare attenzione in fase di sviluppo e debug.

6.4 Un esempio di utilizzo

Dimostriamo ora come finalmente sia possibile eseguire applicativi standard non modificati, nell'ambiente parzialmente virtualizzato fornito da UMView ed utilizzando lo stack di rete NetBSD per la gestione delle socket call.

```
1  $ umview bash
2  $ um_add_service umnet
3  $ mount -t umnetbsd none /dev/net/bsd
4  $ rumpremote ifconfig vde0 create
5  $ rumpremote ifconfig vde0 inet 10.9.2.40 netmask
       255.255.255.0
6  $ rumpremote ifconfig vde0 up
7  $ rumpremote route add default 10.9.2.254
8  add net default: gateway 10.9.2.254
9  $ mstack /dev/net/bsd nettest_simple httpclient
       46.246.93.70
10 Client socket: 4
11 Received:
12 <html><body>
13 <h3>paolo.perfetti.name</h3>
14 <p>Welcome on coppelia!</p>
15 </body></html>
```

I passi necessari sono quelli già visti nel corso di questo documento:

riga 1 avvio del macchina virtuale UMView indicando in processo da virtualizzare;

riga 2 caricamento del modulo `umnet`;

riga 3 `mount` del sottomodulo `umnetbsd`;

righe 4-7 configurazione dell'interfaccia virtuale del kernel rump `vde0`;

riga 9 esecuzione dell'applicativo non modificato, intercettando le socket call tramite l'utilizzo di `mstack`

Una descrizione più dettagliata dell'ambiente di test è disponibile nell'appendice A.

Capitolo 7

Sviluppi futuri

L'implementazione attuale del modulo `umnetbsd` si può considerare a livello di *proof of concept* e la distribuzione ed integrazione in *MView richiede ancora molto lavoro di sviluppo e debug.

Sebbene siano utilizzabili le funzionalità di base, sono ancora richieste alcune modifiche agli script per la traduzione dei simboli di NetBSD (`makerumpdefs.sh`) per raggiungere il completo supporto delle socket call.

L'operazione di `mount` del sottomodulo `umnetbsd` ancora non accetta parametri di configurazione, mentre sarebbe auspicabile poter configurare il socket vde ed altri parametri in fase di caricamento, così come avviene per il modulo `umnetlwipv6`.

La libreria `librump` attualmente viene inclusa in fase di linking. In futuro si dovrebbe valutare la possibilità di utilizzare `dlopen()` per il suo caricamento dedicando particolare attenzione alla possibilità di caricarne istanze diverse in diversi thread. Se ciò risultasse fattibile si potrebbero avere diversi kernel rump, e quindi stack di rete NetBSD, in esecuzione all'interno di UMView. Un'alternativa, forse più complessa, sarebbe la modifica di `librump` affinché il supporto per multipli kernel rump concorrenti fosse incluso direttamente all'interno della libreria.

Alcune voci suggeriscono che una nuova versione multithread di *MView sia in fase di progettazione e sviluppo: se così fosse, la gestione asincrona degli

eventi basata su `*_event_subscribe` richiederebbe una radicale revisione, portando probabilmente a soluzioni più semplici.

Infine questa sperimentazione ha riguardato solo lo stack di rete ma è ragionevole pensare che l'integrazione tra rump e ViewOS possa avvenire in maniera molto più profonda. Un nuovo modulo di UMView `umrump` potrebbe infatti implementare l'interfaccia hypervisor richiesta dai kernel rump dando poi modo di caricare faction e driver come avviene sul layer base di rump.

Uno sviluppo in questa direzione aprirebbe nuovi scenari di virtualizzazione garantendo la possibilità di utilizzare tutti i driver NetBSD estraibili come moduli all'interno della virtualizzazione parziale di UMView.

Capitolo 8

Conclusioni

Il lavoro svolto ha permesso di dimostrare la fattibilità di un'integrazione tra la SCVM realizzata da UMView e il kernel rump di NetBSD. Anche se tuttavia limitato dai problemi di portabilità e di gestione di "domini" diversi in ambiente single-thread, `umnetbsd` fornisce agli utenti la possibilità di avviare e configurare uno stack di rete autonomo rispetto all'host e di comprovata affidabilità.

Pur sperimentando soluzioni diverse, questo lavoro compie un passo nella stessa direzione di altri software attualmente in rapido sviluppo e diffusione [11] e dedicati alla creazione di appliance software in cui un indirizzo di rete non è più assegnato ad un device, ma piuttosto ad un servizio [6].

In più, diversamente dalle altre tecnologie, `umnetbsd` offre la possibilità di gestire autonomamente il proprio stack di rete dove le altre soluzioni o fanno affidamento su uno stack unico, oppure impongono l'overhead di una virtualizzazione completa.

Durante lo sviluppo di questo lavoro si è dimostrata la possibilità d'integrazione tra piattaforme che differiscono per motivazione, sistema operativo di riferimento e tecnica di virtualizzazione.

Come ultima considerazione personale posso affermare che è stato un piacere contribuire all'incontro di due progetti di Software Libero che hanno così tanto da guadagnare dal reciproco scambio e collaborazione.

Appendice A

Ambiente di sviluppo

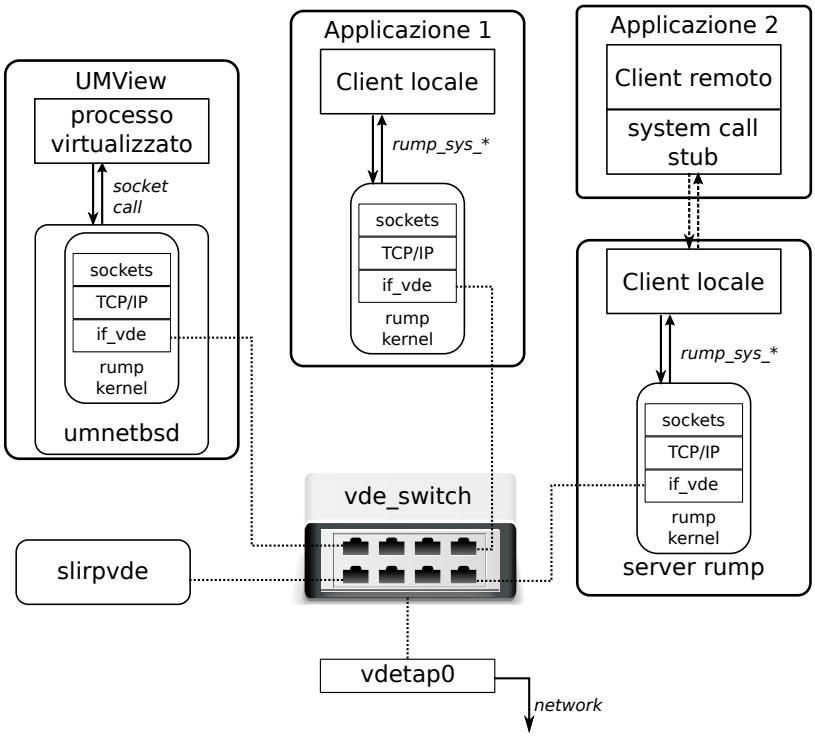


Figura A.1: Rete virtuale di test

L'intero processo di sviluppo è avvenuto su un portatile personale installato con sistema GNU/Linux Debian testing/unstable x86_64:

```
jigen@etah:~$ cat /etc/debian_version
```

```
jessie/sid
jigen@etah:~$ uname -a
Linux etah 3.12-1-amd64 SMP Debian 3.12.9-1
   (2014-02-01) x86_64 GNU/Linux
```

La scelta della distribuzione Debian rispetto ad altre è dovuta alla riconosciuta eticità della comunità di riferimento ed alla sterminata quantità di software disponibile (30.000 pacchetti [31]) ed architetture supportate.

L'ambiente creato per il testing del software prevede un `vde_switch` in esecuzione sul sistema host a cui vengono collegati le interfacce dei kernel rump, siano essi indipendenti o interni a UMView. Coerentemente con quanto sostenuto durante il documento tutta l'infrastruttura di rete è stata creata mantenendo i privilegi di utente:

```
$ vde_switch -s ${VDE_SOCKET} -m 666 -d
$ slirpvde -s ${VDE_SOCKET} -H 10.9.2.254/24 -N 10.9.2.253 -d
```

L'unica eccezione è rappresentata dal comando `vde_plug2tap` , non strettamente necessario ma utile per l'esecuzione di `tcpdump` e per comodità nell'esecuzione di alcuni test:

```
$ sudo vde_plug2tap -s ${VDE_SOCKET} ${VDETAP} -d
$ sudo ifconfig ${VDETAP} 10.9.2.2 netmask 255.255.255.0
$ sudo ifconfig ${VDETAP} up
```

Avendo bisogno di una shell da cui eseguire i successivi comandi di configurazione, si è scelto di virtualizzare all'interno di UMView l'emulatore di terminale `xterm`. Rispetto all'uso di una shell quale `bash`, `xterm` rende la possibile mantenere separati i messaggi di debugging, garantendo una maggior comprensione del comportamento dei programmi:

```
$ umview xterm
```

Una volta avviato un ambiente UMView il tracing delle system call è già attivo ma ancora non è caricato alcun modulo che si occupi di gestirne il comportamento. Per attivare le funzionalità si ricorre al comando `um_add_service`, come già visto nel capitolo 4:

```
$ um_add_service umnet
$ um_ls_service
umnet: virtual (multi-stack) networking
```

Caricato il modulo `umnet` UMView è istruito per intercettare le socket-call. Ancora però non vi è alcuna differenza rispetto l'ambiente standard in quanto tutte le chiamate vengono inoltrate allo stack di rete di default e quindi nessuna modifica è apportata al comportamento dei programmi. Si procede quindi all'attivazione del sottomodulo `umnetbsd` sul mountpoint che successivamente verrà indicato come argomento al comando `mstack`:

```
$ mount -t umnetbsd none /dev/net/bsd
```

Il modulo `umnetbsd` al caricamento si occupa di avviare un server rump linkato con la faction e le librerie che implementano le funzionalità di rete. È possibile quindi proseguire configurando un interfaccia che andrà a collegarsi al `vde_switch` avviato in precedenza e definire le regole di routing:

```
$ rumpremote ifconfig vde0 create
$ rumpremote ifconfig vde0 inet 10.9.2.40 netmask 255.255.255.0
$ rumpremote ifconfig vde0 up
$ rumpremote route add default 10.9.2.254
add net default: gateway 10.9.2.254
$ rumpremote ifconfig vde0
vde0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        address: b2:0a:4c:0b:0e:00
        inet 10.9.2.40 netmask 0xffffff00 broadcast 10.9.2.255
```

Il programma `nettest_simple` consiste in alcune semplici procedure che stabiliscono una connessione TCP e si scambiano una stringa di test. Agendo sui parametri passati da riga di comando è possibile decidere il ruolo tenuto dalla specifica istanza di `nettest_simple` che può comportarsi in diverse modi:

- server: utilizzando le socket call standard il programma si pone in ascolto sulla porta CONNPORT in attesa di connessioni;

- client: utilizzando le socket call standard il programma si connette alla porta CONNPORT dell'indirizzo IP passato come argomento;
- rump_server: il programma agisce come client remoto del RUMP_SERVER passato come argomento e utilizzando le system call rump si pone in ascolto sulla porta CONNPORT;
- rump_client: il programma agisce come client remoto del RUMP_SERVER passato come argomento e utilizzando le system call rump si connette alla porta CONNPORT dell'IP specificato;
- httpclient: il programma si connette alla porta 80 dell'indirizzo IP passato come argomento e richiede l'indice della directory principale ("`GET /`");
- rump_httpclient: come la precedente, ma utilizzando le chiamate rump ed inoltrandole a RUMP_SERVER.

Se si avvia `nettest_simple` in modalità server sul sistema host e nell'ambiente definito in precedenza, i seguenti comandi eseguiti all'interno di UMView si comportano alla stessa maniera:

Client `umnetbsd`

```
$ mstack /dev/net/bsd nettest_simple client 10.9.2.2
Client socket: 4
Received: Helo World, I'm a TEST STRING
```

In questo caso `nettest_simple` utilizzando le socket call standard, intercettate dal modulo `umnetbsd` di UMView ed inoltrate al server rump come in modalità "client locale"

Client rump remoto

```
$ nettest_simple rump_client unix:///tmp/rump_socket 10.9.2.2
Client, using RUMP_SERVER unix:///tmp/rump_socket
rumpclient_init: 0
```

```
Client socket: 0
Received: Helo World, I'm a TEST STRING
```

Specificando l'argomento rump_client, invece `nettest_simple` fa uso delle socket call rump in modalità "client remoto". Queste system call (`rump_sys_*`) non sono intercettate da UMView ma vengono direttamente inoltrate al server rump eseguito come parte del modulo `umnetbsd`.

Seguendo la stessa procedura e riprendendo l'esempio visto nella sezione 6.2 è possibile sfruttare il routing messo a disposizione dall'esecuzione di `slirpvde` per accedere ad un host su internet:

```
$ mstack /dev/net/bsd nettest_simple httpclient 46.246.93.70
Client socket: 4
Received:
<html><body>
<h3>paolo.perfetti.name</h3>
<p>Welcome on coppelia!</p>
</body></html>
```

```
export PRJ_ROOT=~/tesi
export SRC_PREFIX=${PRJ_ROOT}/src
export INSTALL_PREFIX=${PRJ_ROOT}/install
export RR_SRC=${SRC_PREFIX}/rumprun
export RR_DYN_INSTALL=${RR_SRC}/rumpdyn
CPPFLAGS="-I${INSTALL_PREFIX}/include -I${RR_DYN_INSTALL}/include"
LDFLAGS="-L${RR_DYN_INSTALL}/lib -L${INSTALL_PREFIX}/lib"
LD_LIBRARY_PATH=${INSTALL_PREFIX}/lib
LD_LIBRARY_PATH=${SRC_PREFIX}/tests/libshmif_linux:${LD_LIBRARY_PATH}
LD_LIBRARY_PATH=${INSTALL_PREFIX}/lib/umview/modules:${LD_LIBRARY_PATH}
LD_LIBRARY_PATH=${RR_DYN_INSTALL}/lib:${LD_LIBRARY_PATH}
LD_LIBRARY_PATH=${RR_SRC}:${LD_LIBRARY_PATH}
PATH="${INSTALL_PREFIX}/bin/":${RR_SRC}:${PATH}
export VDE_SOCKET=/tmp/vde0
export VDETAP=vdetap0
export RUMP_SOCKET=/tmp/rump_socket
export CPPFLAGS
export LDFLAGS
export LD_LIBRARY_PATH
export PATH
```

Figura A.2: Variabili d'ambiente

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. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[2] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[3] Eric Biederman and Linux Networx. Multiple instances of the global linux namespaces. In *Proceedings of the Linux Symposium*. Citeseer, 2006.

[4] Susanta Nanda Tzi-cker Chiueh and Stony Brook. A survey on virtualization technologies. *RPE Report*, pages 1–42, 2005.

[5] R. Davoli and M. Michael Goldweber. *Virtual Square: Users, Programmers & Developers Guide*. Disponibile a http://www.cs.unibo.it/ renzo/virtualsquare/.

[6] Renzo Davoli. Internet of threads. In *ICIW 2013, The Eighth International Conference on Internet and Web Applications and Services*, pages 100–105, 2013.

[7] 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.

[8] Kevin R Fall and W Richard Stevens. *Tcp/ip illustrated*, volume 1. Addison-Wesley Professional, 2011.

[9] Ludovico Gardenghi, Michael Goldweber, and Renzo Davoli. View-os: A new unifying approach against the global view assumption. In *Proceedings of the 8th international conference on Computational Science, Part I*, ICCS '08, pages 287–296, Berlin, Heidelberg, 2008. Springer-Verlag.

[10] Jorrit N Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S Tanenbaum. Minix 3: A highly reliable, self-repairing operating system. *ACM SIGOPS Operating Systems Review*, 40(3):80–89, 2006.

[11] Docker Inc. Docker: lightweight, portable, self-sufficient containers from any application. http://www.docker.io, 2014.

[12] V Jacobsen, Craig Leres, and Steven McCanne. Tcpdump/libpcap, 2005.

[13] Poul-Henning Kamp and Robert NM Watson. Jails: Confining the omnipotent root. In *Proceedings of the 2nd International SANE Conference*, volume 43, page 116, 2000.

[14] Antti Kantee. Environmental independence: Bsd kernel tcp/ip in userspace. *Proc. of AsiaBSDCon*, pages 71–80, 2009.

[15] Antti Kantee. Flexible operating system internals: The design and implementation of the anykernel and rump kernels. Doctoral Dissertation, Aalto Univerisity, Finland, 2012.

[16] Antti Kantee and Justin Cormack. Tools for building Rump Kernels. https://github.com/rumpkernel/buildrump.sh, 2012.

[17] Tom J Killian. Processes as files. In *USENIX Summer Conference Proceedings*, 1984.

[18] Butler W. Lampson. Hints for computer system design. *SIGOPS Oper. Syst. Rev.*, 17(5):33–48, oct 1983.

[19] Luke Mewburn and Matthew Green. build.sh: Cross-building NetBSD. In *BSDCon*, pages 47–56, 2003.

[20] VM Oracle. Virtualbox. *User Manual–2013*, 2013.

[21] POSIX.1-2008. The Open Group Base Specifications. Also published as IEEE Std 1003.1-2008, July 2008.

[22] Eric Steven Raymond, Thyrsus Enterprises, Copyright Eric, and S. Raymond. The art of unix programming. Addison-Wesley, 2003.

[23] Dennis M. Ritchie and Ken Thompson. The UNIX Time-sharing System. *Commun. ACM*, 17(7):365–375, jul 1974.

[24] Robert Rose. Survey of system virtualization techniques. 2004.

[25] James E Smith and Ravi Nair. The architecture of virtual machines. *Computer*, 38(5):32–38, 2005.

[26] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. *SIGOPS Oper. Syst. Rev.*, 41(3):275–287, March 2007.

[27] W Richard Stevens. *UNIX network programming*, volume 1. Addison-Wesley Professional, 2004.

[28] Andrew Whitaker, Marianne Shaw, Steven D Gribble, et al. Denali: Lightweight virtual machines for distributed and networked applications. Technical report, Citeseer, 2002.

[29] Gary R Wright and W Richard Stevens. *TcP/IP Illustrated*, volume 2. Addison-Wesley Professional, 1995.

[30] Junfeng Yang, Can Sar, Paul Twohey, Cristian Cadar, and Dawson Engler. Automatically generating malicious disks using symbolic execution. In *Security and Privacy, 2006 IEEE Symposium on*, pages 15–pp. IEEE, 2006.

[31] Stefano Zacchiroli. Legal issues from a radical community angle, FOSDEM 2014, February 2014. Original presentation and video available at https://fosdem.org.

[32] Marko Zec. Implementing a Clonable Network Stack in the FreeBSD Kernel. In *USENIX Annual Technical Conference, FREENIX Track*, pages 137–150, 2003.

Ringraziamenti

Agli uomini rinchiusi: passati, presenti e futuri.
Con ridondanza ai MeTA˜LAbS, laboratorio distribuito, sudditi del doge, testimonianza tangibile di lealtà ed amicizia.
Alla mia famiglia, che raramente mi vede ed oramai qualche volta si dimentica anche come sono fatto.
Alle colleghe ed ai colleghi che per anni mi hanno motivato e spesso ci hanno creduto più di me.
A chi per tanti anni mi ha sempre s[o|u]pportato, che mi trovassi molto molto vicino o molto molto lontano.
A `simonett`, senza di cui mi mancherebbero ancora n esami ed avrei qualche pezzo di fegato in più.
A `scacciag`, senza di cui sarei emigrato da un pezzo.
A chi, quando io non c'ero, ha fatto la propria parte e pure la mia.
A `renzo`, per la disponibilità e la fiducia.
A chi mi ha aiutato, in giorni difficili e confusi, in corse frenetiche, nei deliri e nel panico.
A quanti non sono inclusi nei ringraziamenti.