Alma Mater Studiorum · Università di Bologna

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
Corso di Laurea Specialistica in Informatica

DESIGN E IMPLEMENTAZIONE DEL NUOVO FRAMEWORK VIRTUAL DISTRIBUTED ETHERNET: INGEGNERIZZAZIONE E INTRODUZIONE NUOVI COMPONENTI

Tesi di Laurea in Progettazione di Sistemi Operativi

Relatore:
Renzo Davoli

Presentata da:
Filippo Giunchedi

Terza Sessione
Anno Accademico 2008/2009

Networking — only one letter away
from *not working*.

Anonimo

Indice

Indice **i**

Elenco delle figure **v**

Elenco dei listati **vii**

Introduzione **1**

1 Quadro di riferimento **3**

1.1 VDE 2 . . . 3

1.1.1 Le caratteristiche . . . 3

1.1.2 La struttura . . . 6

1.1.3 Il nostro contributo . . . 12

1.1.4 Limitazioni . . . 14

1.2 Virtual Square . . . 17

1.2.1 View-OS . . . 17

1.2.2 IPN . . . 18

1.2.3 Possibili utilizzi . . . 19

2 VDE 3 **23**
2.1 Architettura 24
2.1.1 Scelte architetturali 24
2.1.2 Visione d'insieme 26
2.1.3 Dettagli implementativi 28
2.2 Confronto con VDE 2 42
2.2.1 Comparazione funzionale 43
2.2.2 Reimplementazione con `libvde` 45
2.3 Analisi delle prestazioni 46
2.3.1 Ambiente di testing 46
2.3.2 Misurazioni effettuate 49
2.3.3 Analisi dei risultati 50
2.4 Ingegnerizzazione 65
2.4.1 Modello di sviluppo 65
2.4.2 Protocolli e formati 66
2.4.3 Generazione automatica del codice 68
2.4.4 Testing 70
2.4.5 Documentazione 74
2.4.6 Community 75
2.5 Nuovi componenti 77
2.5.1 Demone di controllo per STP 77
2.5.2 Anonimato e offuscamento del traffico 78
2.5.3 Logging eventi asincroni 78
2.5.4 Integrazione emulatore Cisco 79
2.5.5 Integrazione con libvirt 79
2.5.6 Interfacciamento con linguaggi ad alto livello 80

Conclusioni **81**

Sviluppi futuri 82

A Header rilevanti **85**

B Ambiente di testing **113**

Bibliografia **123**

Ringraziamenti **127**

Elenco delle figure

1.1 Esempio di collegamento con VDE 4
1.2 Diagramma dei context switch con wirefilter 16
1.3 Diagramma dei componenti di Virtual Square 21

2.1 Schema di un contesto VDE 3 27
2.2 Rappresentazione della gerarchia dei componenti 32
2.3 Diagramma di sequenza per una nuova connessione 36
2.4 Diagramma di sequenza per lo scambio di `vde_pkt` 38
2.5 Diagramma di sequenza della fase di autorizzazione 40
2.6 Schematizzazione dell'ambiente di testing 47
2.7 Utilizzo di bandwidth di vde2_hub e vde3_hub 51
2.8 Bandwidth e latenza delle reti VDE 2 e VDE 3 51
2.9 Utilizzo delle risorse di vde2_hub e vde3_hub 52
2.10 Utilizzo di bandwidth di vde2_wf e vde3_hub2hub 54
2.11 Utilizzo delle risorse di vde2_wf e vde3_hub2hub 55
2.12 Utilizzo di bandwidth di vde2_switch, vde3_hub e kvde . . 57
2.13 Bandwidth e latenza di vde2_switch, vde3_hub e kvde . . . 57
2.14 Utilizzo delle risorse di vde2_switch, vde3_hub e kvde . . . 58
2.15 Utilizzo di bandwidth di vde3_hub, kvde e bridge 60

2.16 Bandwidth e latenza di vde3_hub, kvde e bridge 60
2.17 Utilizzo delle risorse di vde3_hub, kvde e bridge 61
2.18 Utilizzo di bandwidth di kvde, bridge e virtio 63
2.19 Bandwidth e latenza di kvde, bridge e virtio 63
2.20 Utilizzo delle risorse di kvde, bridge e virtio 64

Elenco dei listati

2.1 Un hub di esempio creato con `libvde` 29
2.2 La struttura vde_module e le funzioni specifiche 31
2.3 La struttura vde_event_handler 31
2.4 Descrizione JSON di un comando esportato 69
2.5 Esempio di documentazione nel codice 75
2.6 Esempio di markup RST 76
A.1 Header vde3.h 85
A.2 Header module.h 91
A.3 Header context.h 94
A.4 Header component.h 96
A.5 Header transport.h 102
A.6 Header engine.h 104
A.7 Header connection.h 104
B.1 Script principale per la gestione dei test 113
B.2 Script per l'esecuzione di un test e la raccolta dei dati 114
B.3 Esempio di test 116
B.4 Estratto dello script di management delle Virtual Machine . 116

Introduzione

La *virtualizzazione* pervade molti aspetti dell'informatica, dai linguaggi di programmazione ai sistemi operativi. L'idea di poter disporre di risorse virtuali non è nuova, basti pensare al sistema CP/CMS presente nei *mainframe* IBM System/360 sin dai primi anni '60. Oggi tuttavia la ricerca in questo campo trova rinnovato fervore poiché il forte incremento delle capacità di calcolo dei dispositivi di utilizzo comune e la diffusione delle reti hanno introdotto nuove necessità di sicurezza e di utilizzo delle risorse.

Virtual Distributed Ethernet (VDE) si inserisce in questo scenario per fornire un punto di contatto tra varie tipologie di reti virtuali consentendone l'interoperabilità in modo flessibile e trasparente. Il progetto, arrivato al suo sesto anno di sviluppo, nutre un crescente numero di utilizzatori e viene ampiamente impiegato in svariati campi quali l'interconnessione di macchine virtuali e la simulazione di reti a scopi didattici.

Il lavoro presentato in questa tesi è frutto di una collaborazione con Luca Bigliardi e nasce dalla volontà di entrambi di risolvere alcuni limiti da noi individuati nel corso degli anni durante il nostro coinvolgimento nello sviluppo.

La nuova architettura mira ad aumentare la flessibilità del progetto ed al tempo stesso renderne facili l'espansione e la manutenzione. A questo scopo è stata implementata una libreria asincrona basata su un modello ad eventi e su un sistema di moduli dinamici che forniscono varie tipologie di componenti.

Il capitolo 1 presenta inizialmente le caratteristiche di VDE e descrive le varie parti che lo compongono. Successivamente illustra il nostro contributo al progetto ed analizza i limiti individuati nell'architettura di VDE 2.

Il capitolo 2 tratta in modo estensivo la nuova architettura proposta offrendo dapprima una panoramica concettuale e una descrizione dei dettagli implementativi. In seguito effettua un confronto tra VDE 2 e VDE 3 sia sul piano funzionale che sul piano prestazionale suggerendo soluzioni per la piena retrocompatibilità e analizzandone le performance. Successivamente discute alcune scelte progettuali ed il processo di sviluppo seguito durante il lavoro. Infine presenta nuovi componenti che è possibile realizzare grazie alla nuova architettura e i benefici che possono portare.

Lo studio e l'implementazione sono il risultato di un lavoro congiunto. Nello sviluppo di questa tesi mi sono concentrato maggiormente sull'ingegnerizzazione del progetto e sullo studio di nuovi componenti, mentre Luca Bigliardi ha lavorato sull'analisi delle prestazioni e la retrocompatibilità.

Il capitolo conclusivo riassume il lavoro svolto inquadrando la nostra proposta all'interno del progetto VDE nella sua interezza e descrive nuovi problemi emersi durante lo sviluppo ipotizzandone possibili soluzioni.

Capitolo 1

Quadro di riferimento

1.1 VDE 2

Il progetto VDE [7], arrivato nella sua corrente implementazione alla versione 2.2.3, è stato di fondamentale ispirazione per il lavoro di tesi. È pertanto giusto analizzarne le caratteristiche a cui abbiamo fatto riferimento e che siamo riusciti a cogliere grazie ad un'assidua partecipazione allo sviluppo.

1.1.1 Le caratteristiche

L'autore, Renzo Davoli, definisce VDE come «*a swiss army knife for emulated networks*»[1] per sottolineare la flessibilità del progetto nel creare topologie di reti virtuali tra macchine fisiche, macchine virtuali, emulatori ecc. in modo trasparente.

Come suggerito dal nome, *Virtual Distributed Ethernet* coinvolge più entità (*Distributed*) siano esse fisiche o virtuali, è completamente software (*Virtual*) e offre piena compatibilità con lo standard per reti LAN (*Ethernet*). Nel framework vengono utilizzate alcune analogie con il mondo fisico quali switch e cavi di rete, come si può vedere in figura 1.1.

[1] *Un coltellino svizzero per reti emulate.*

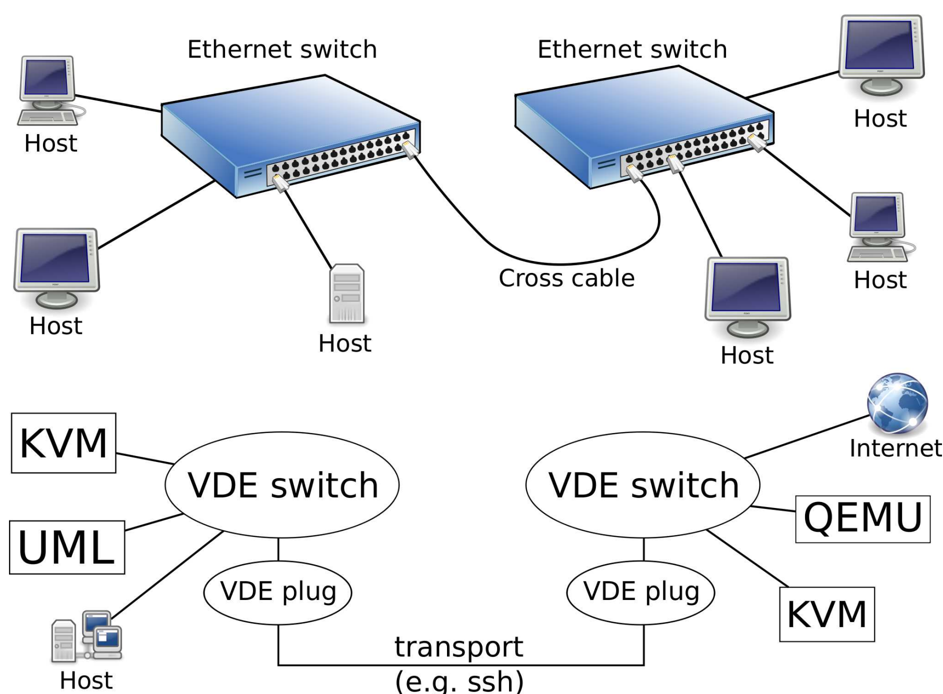


Figura 1.1: Esempio di collegamento con VDE

La maggior parte delle funzionalità di VDE sono disponibili senza privilegi di amministratore, questo consente di minimizzare le implicazioni di sicurezza e l'invasività del software: è possibile per ogni utente far girare uno o più *switch virtuali* indipendenti e ognuno con i controlli d'accesso sui file e sulle risorse di sistema dell'utente stesso.

La versatilità di VDE ha permesso di implementare molte soluzioni legate al *virtual networking*. Alcune di queste hanno controparti nel mondo del software commerciale ed è pertanto di particolare interesse avere implementazioni realizzate interamente da software libero:

VPN (esempio: *OpenVPN*[2]) Una *Virtual Private Network* è una rete di comunicazione privata spesso utilizzata per comunicare in modo sicuro attraverso una rete pubblica (ad esempio Internet). Ciò che caratterizza una VPN è l'essere punto-punto e il collegare due host fisici. Nel caso di VPN realizzate con VDE il concetto viene esteso a host

[2] http://openvpn.net

virtuali collegati in VPN tra loro, potenzialmente anche sulla stessa macchina fisica.

Overlay network (esempio: *Akamai*[3], *peer-to-peer software*) Una *overlay network* è una rete di computer costruita al di sopra di un'altra rete. I nodi di una overlay network possono essere connessi da link virtuali logicamente separati dalla rete sottostante. Ogni collegamento si può estendere attraverso più link della rete su cui l'overlay si poggia. Un esempio di overlay network è la stessa Internet, se usata in modo Dial-Up tramite una rete telefonica.

VM network (esempio: *uml-switch*, *VMWare's vmnet*[4]) Una *Virtual Machine network* è una rete simulata usata per far comunicare le macchine virtuali presenti su un unico host fisico. È di particolare importanza dato il crescente interesse nella virtualizzazione per ottimizzare i costi ed impiegare al meglio tutta la potenza di calcolo delle macchine fisiche.

NAT traversal (esempio: *Hamaci*[5], *Freenet*[6]) è una tecnica per consentire a due host protetti da *NAT gateway/firewall* differenti di comunicare direttamente tra loro, cosa impossibile altrimenti poiché ad entrambi gli host vengono assegnati indirizzi IP privati e non utilizzabili direttamente su internet.

Mobility supporto trasparente alla mobilità per macchine reali o virtuali. Questo risultato è stato per la maggior parte frutto del nostro lavoro su VDE 2, si veda 1.1.3 in merito.

Networking lab è un laboratorio dove costruire qualsiasi topologia di rete, senza necessità di una rete fisica. Il campo naturale di applicazione è l'insegnamento e la sperimentazione di protocolli di rete reali che

[3] http://www.akamai.com
[4] http://www.vmware.com
[5] http://www.hamachi.cc
[6] http://freenetproject.org

richiedono l'interazione tra molti host e tra sistemi operativi differenti. VDE viene attualmente utilizzato all'interno del tool *marionnet* [24] precisamente con questo scopo ed è stato utilizzato come tool di insegnamento anche in altri contesti illustrati da R. Davoli e M. Goldweber in [17].

1.1.2 La struttura

VDE è una collezione di applicazioni sviluppate per sistemi POSIX [21], interamente rilasciate sotto licenza GPL [14] e pertanto liberamente modificabili, studiabili e redistribuibili.

Come da tradizione UNIX, una delle filosofie alla base di VDE è quella della *modularità* e della *semplicità* dei componenti: i componenti specializzati possono essere riutilizzati come "mattoni" in altri progetti mentre componenti semplici sono meno suscettibili di bug ed il loro ruolo e funzionamento risulta chiaro.

Nel caso specifico di VDE i componenti sono implementati come applicazioni *standalone* che vengono collegate tra loro con meccanismi di IPC[7] quali *pipe* e *socket UNIX*. Di seguito vengono illustrate le applicazioni della suite VDE:

vde_switch costituisce la parte centrale di VDE e riproduce fedelmente uno switch Ethernet, instradando i pacchetti in ingresso sulle porte alle loro destinazioni secondo l'indirizzo MAC. Mantenendo l'analogia con il mondo reale, è possibile collegare più switch attraverso cavi (*VDE cable*) per formare reti più estese.

Lo switch supporta inoltre un protocollo di STP[8] per evitare *loop* all'interno di reti con collegamenti ridondati. Per aumentare maggiormente la flessibilità è stato implementato anche il supporto alle VLAN

[7] Inter Process Communication
[8] Spanning Tree Protocol

secondo specifica IEEE 802.1Q [20] che permette di separare logicamente sottoreti appartenenti allo stesso segmento fisico, esattamente come uno switch Ethernet reale di tipo *managed*.

vde_plug rappresenta un vero e proprio *connettore*: viene collegato ad una porta di `vde_switch` e legge/scrive i dati da/per la rete virtuale attraverso i propri canali di *standard input* e *standard output*. Questo lo rende del tutto simile ad un tool standard UNIX per l'analisi di flussi di byte[9].

dpipe è un'applicazione sviluppata per VDE ma di utilità generale. Nel mondo UNIX l'output di un processo può essere collegato all'input di un altro tramite pipe per formare filtri o comunque elaborare dei risultati.

Il concetto di pipe unidirezionale è stato ampliato tramite `dpipe` con l'idea di una *pipe bidirezionale*: l'output di un processo è accoppiato con l'input di un altro *e viceversa* l'output del secondo è connesso con l'input del primo.

Questo concetto in apparenza semplice, se unito alla moltitudine di comandi UNIX e applicazioni ad-hoc come `vde_plug`, risulta molto potente: per collegare due `vde_switch` locali è sufficiente

```
dpipe vde_plug switch1 = vde_plug switch2
```

e di fatto si è formato un *cavo virtuale* tra `switch1` e `switch2`.

In questo modo il *trasporto* del flusso dati è trasparente ai `vde_plug` e può essere cambiato a piacimento, ad esempio sostituito con canali sicuri di comunicazione attraverso internet quali SSH[10] per ottenere in modo semplice delle *Virtual Private Network*. L'esempio precedente può essere sintetizzato dal comando

```
dpipe vde_plug switch1 = ssh server vde_plug switch2
```

In questo caso `switch1` locale è connesso a `switch2` della macchina remota `server`.

[9] Ad esempio `sed`, `awk` o `grep`.

[10] http://openssh.com

vde_plug2tap è connesso a un `vde_switch` similmente a `vde_plug` ma l'input/output da/per la rete virtuale è rediretto su un'interfaccia di rete di tipo `tap`: in questo modo attraverso *bridging* di `tap` con un'interfaccia fisica è possibile affacciare uno switch virtuale su di una rete reale facendolo apparire in tutto e per tutto come un dispositivo fisico.

vde_tunctl è una utility che permette di creare e configurare device di tipo TUN/TAP. Grazie a questo tool un amministratore di sistema può preconfigurare un'interfaccia `tap` per essere utilizzata da un particolare utente che non dispone dei privilegi di amministratore. L'utente può utilizzare il device, ma non può cambiarne gli aspetti relativi all'interfaccia host.

vde_pcapplug è un'altra applicazione analoga a `vde_plug`, da un lato connessa ad un `vde_switch` e dall'altro collegata ad un'interfaccia Ethernet configurata in modalità promiscua. Tutti i pacchetti letti dall'interfaccia Ethernet vengono inviati nella rete virtuale tramite lo switch e viceversa tutti i pacchetti ricevuti dallo switch vengono iniettati nell'interfaccia Ethernet.

wirefilter è uno strumento per «*rendere un po' più reale il virtuale*», in altre parole per introdurre problemi tipici di reti fisiche quali rumore, limiti di banda, ritardi casuali ecc.

Da un punto di vista pratico agisce come un vero e proprio filtro da frapporre tra due `vde_plug` tramite `dpipe`:

```
dpipe vde_plug sw1 = wirefilter = vde_plug sw2
```

In questo modo il traffico tra `sw1` ed `sw2` potrà essere cambiato tramite le funzioni messe a disposizione da `wirefilter`, in entrambe le direzioni è possibile specificare:

- perdita/duplicazione di pacchetti in percentuale;
- ritardi di trasmissione in millisecondi;
- limiti di banda del canale e di velocità di trasmissione;

- limiti alla coda dei pacchetti e all'MTU;
- numero di bit danneggiati per megabyte;
- riordino dei pacchetti in transito.

vde_cryptcab [11] implementa un trasporto sicuro alternativo a SSH basato su UDP invece che TCP. La scelta di UDP è dettata dalla scarsa efficienza nell'incapsulare una connessione TCP (quella sul cavo virtuale) dentro un'altra (quella SSH, che trasporta Ethernet) specie in presenza di perdita di pacchetti. Quello che accade è che i meccanismi di *flow-control* agiscono in contemporanea ma con diversi *timeout* e significative perdite di performance, si vedano al riguardo [36] e [18].

Dal lato pratico viene dapprima scambiata una chiave privata tramite un canale sicuro già stabilito in precedenza[12] ed in seguito vengono connessi due `vde_plug` attraverso lo scambio di datagrammi UDP cifrati con il protocollo *Blowfish* [31] utilizzando la chiave privata. Questo approccio non comporta decrementi di sicurezza rispetto al trasporto SSH risolvendo però i problemi di performance sopraccitati.

vde_over_ns permette a due `vde_switch` remoti di comunicare mascherando lo scambio dei frame Ethernet all'interno di un flusso di finte query ad un server DNS. Questa metodologia di comunicazione, pur nella sua inefficienza, è in grado di superare alcune restrizioni tipicamente messe in atto dagli amministratori di reti ad accesso pubblico: il blocco di tutto il traffico verso l'esterno ad esclusione delle richieste DNS. La raggiungibilità di un server DNS esterno risulta fondamentale in quanto il client vde_over_ns si connette direttamente alla sua controparte server in ascolto sulla porta 53 UDP nell'host remoto.

slirpvde fornisce un'alternativa per connettere `vde_switch` alla rete reale senza l'ausilio di `tap` e soprattutto senza privilegi di amministratore.

[11] Da un'idea di Renzo Davoli e scritto da Daniele Lacamera.
[12] Ad esempio tramite SSH.

Il codice sorgente deriva per la maggior parte dal supporto di rete di QEMU [3] che a sua volta deriva da quello di Slirp [15].

Il suo funzionamento accomuna le caratteristiche dei *proxy* e del *Network Address Translation*: vengono registrate le connessioni per la rete esterna e il payload a livello 4 spedito in modo che appaia come generato direttamente da `slirpvde`. Alla ricezione di un pacchetto dalla stessa connessione questo viene ricreato sulla rete virtuale e spedito a `vde_switch`.

È anche presente la possibilità di registrare sulla rete virtuale un server DHCP, tutti gli indirizzi IP rilasciati da questo server sono riconosciuti da `slirpvde` e pertanto vengono mappati sulla rete esterna. Lo stesso vale per le query DNS che vengono risolte utilizzando il DNS della macchina ospite, fornendo pertanto la possibilità alle macchine della rete virtuale l'accesso di base alla esterna.

L'assenza di privilegi di amministratore comporta da un lato indubbi vantaggi per la sicurezza e la praticità di utilizzo, dall'altro l'impossibilità di generare certi tipi di traffico verso la rete esterna solitamente riservati a utenti con particolari privilegi quali ad esempio *ICMP echo request/reply*[13].

vde_l3 è un router *layer 3* che connette uno o più `vde_switch` tramite IP forwarding. Grazie a questo tool è possibile suddividere una rete Ethernet virtuale in più sottoreti in modo da controllare il proliferarsi di pacchetti di broadcast e multicast che possono degradare le performance di tutta la rete virtuale.

unixterm è lo strumento ispirato dalle *console seriali* utilizzato per il management dei componenti che prevedono amministrazione remota[14] con cui comunica tramite socket UNIX.

Sebbene anche questo *tool* sia semplice risulta tuttavia fondamentale poiché permette di accedere e modificare le funzionalità dei compo-

[13] In altre parole l'uso del comando `ping`.
[14] Ad esempio `vde_switch` e `wirefilter`.

nenti dinamicamente a *runtime*. Ad esempio è possibile monitorare lo stato delle porte di `vde_switch`, controllare la configurazione delle VLAN o variare il filtraggio applicato da `wirefilter` ai dati in transito.

vdeterm è uno strumento analogo a unixterm che offre una console leggermente più avanzata grazie all'utilizzo di alcune librerie che permettono di avere l'history dei comandi ed un elementare sistema di completamento.

unixcmd è un tool che consente di impartire un singolo comando sulla console di un'applicazione VDE. Questa utility è particolarmente indicata per includere il management all'interno di uno script poiché il nome del comando viene passato come argomento all'eseguibile, il risultato viene scritto sullo standard output e il valore di ritorno viene usato come valore di uscita di unixcmd.

vdetelweb costituisce un ulteriore strumento per l'amministrazione di `vde_switch`. A differenza di `unixterm` però, mette a disposizione un server `telnet` e un'*interfaccia web* per gestire lo switch in modo semplice attraverso una connessione TCP piuttosto che su socket UNIX.

vde_autolink fornisce un sistema di monitoraggio e ripristino delle connessioni di un `vde_switch`. Collegandosi al management dello switch stesso monitora lo stato di una connessione ed eventualmente tenta di ristabilirla in caso venga interrotta.

kvde_switch si tratta di uno switch sperimentale che non fa transitare i dati in userspace ma che li smista tra i vari client direttamente in kernel space appoggiandosi al meccanismo di inter-process communication IPN descritto in 1.2.2.

vdeq presente nel framework per ragioni di retrocompatibilità, implementa un wrapper per QEMU/KVM che ne consente la connessione alla

rete VDE. Il suo utilizzo è ad oggi deprecato in quanto QEMU/KVM ha ora il supporto nativo per VDE.

A corredo delle applicazioni sono state sviluppate alcune librerie che hanno permesso di riutilizzare parti di codice ed hanno facilitato l'inserimento del supporto VDE in software esterni al framework. Di seguito ne illustriamo le principali:

libvdeplug fornisce funzioni che permettono di connettersi come client ad una porta di `vde_switch`.

libvdeplug_dyn è un header che permette di caricare libvdeplug a runtime in modo dinamico. Un'applicazione, includendo al proprio interno questo header, non richiede la presenza di VDE a compile time, ma solo quando viene eseguita.

vdetaplib è una libreria che emula l'API utilizzata per governare un'interfaccia `tap` dirottando i frame all'interno di una rete VDE. Per poter essere utilizzata va caricata tramite la variabile di ambiente `LD_PRELOAD` prima dell'esecuzione dell'applicazione il cui supporto `tap` dev'essere redirezionato in VDE.

libvdemgmt consente un accesso programmatico alla console di management dei vari componenti del framework effettuando il parsing dell'output della console. Tramite questa libreria applicazioni esterne possono monitorare e configurare dinamicamente tutti i tool che dispongono di una console.

libvdesnmp fornisce un'interfaccia tra l'output della libreria di management ed il protocollo SNMP.

1.1.3 Il nostro contributo

La nostra partecipazione allo sviluppo di VDE è iniziata con i progetti alla base delle nostre tesi di laurea triennale e si è protratta sino ad oggi. Nel

corso di questo periodo abbiamo contribuito in vari modi: aggiungendo funzionalità, curando la distribuzione e il porting su varie piattaforme, risolvendo bug e fornendo aiuto e supporto alla crescente comunità. Riportiamo ora in sintesi i risultati più significativi da noi ottenuti.

Sul piano funzionale il nostro apporto si è dapprima concentrato sul sottosistema di controllo del framework. Abbiamo introdotto la possibilità di ricevere *notifiche asincrone* sulla console di management di `vde_switch`, facilitando il debugging di alcune feature ed il monitoring in tempo reale degli eventi che avvengono all'interno dello switch stesso.

Parallelamente alle notifiche abbiamo sviluppato la libreria `libvdemgmt` e parte del sistema di parsing ivi utilizzato.

Grazie alle due precedenti *feature* ci è stato possibile aggiungere un *agent SNMP* [16] per il monitoring del traffico sulle porte dello switch che garantisce un controllo unificato di `vde_switch` assieme ad altri apparati fisici ed anche un sistema per il *ripristino automatico delle connessioni* con *supporto alla mobilità* [4] che permette alla rete virtuale di riconfigurarsi in autonomia per rimanere operativa qualora le connessioni remote vengano interrotte per problemi alla rete fisica o per modifiche nell'ambiente circostante il dispositivo su cui è in esecuzione il nodo VDE.

Con l'introduzione in `vde_switch` del supporto a moduli dinamici per l'inspection e il filtering dei pacchetti abbiamo sviluppato un plugin che converte il traffico di rete nel formato della libreria libpcap [22] e lo salva su un file o lo redirige su una pipe, rendendo possibile il monitoraggio in tempo reale del traffico di rete con strumenti quali *Wireshark*[15].

Sempre utilizzando la libreria libpcap abbiamo creato `vde_pcapplug` che, utilizzando un'interfaccia Ethernet configurata in modalità promiscua, rende possibile connettere la rete VDE virtuale direttamente ad una rete reale senza l'ausilio di interfacce `tap` e di bridging.

Per semplificare l'utilizzo di VDE con alcune tra le più note tecnologie di virtualizzazione quali *User Mode Linux* [10] e *QEMU/KVM* abbiamo svi-

[15] http://www.wireshark.org

luppato delle *patch* per aggiungere a questi il *supporto nativo* a VDE. Siamo quindi entrati in contatto con le comunità opensource che sviluppano UML e QEMU/KVM ed a seguito di un dialogo costruttivo le patch da noi elaborate sono state incorporate nelle versioni ufficiali. Questo significa che gli utenti possono utilizzare VDE con UML e QEMU/KVM semplicemente installando il software senza ulteriori modifiche manuali.

Sempre al fine di espandere la visibilità e l'accettazione di VDE abbiamo implementato le modifiche necessarie al *porting* su FreeBSD, sistemato il supporto a Mac OS X e curato ulteriormente i pacchetti Debian GNU/Linux[16], con relativa gestione delle release e dei *bugfix* specifici di Debian.

Sul piano della manutenzione e del supporto abbiamo esercitato una costante attività di *debugging* dei malfunzionamenti del framework che sono stati scoperti grazie a diversi scenari di testing. Una buona parte dei bug è venuta alla luce grazie alle segnalazioni che la comunità di utenti ha effettuato tramite canali quali la pagina del progetto VDE sul portale SourceForge[17] o il *bug tracker* di Debian.

Il rapporto con gli utenti sia sotto forma di risoluzione dei bug che di aiuto nella configurazione per far fronte a diverse esigenze ci è stato molto utile per incrementare a vari livelli di dettaglio la nostra padronanza del framework e soprattutto ci ha portato a conoscere un ampio ventaglio di scenari ed esigenze reali nei quali viene impiegato VDE fornendoci quindi una preziosa collezione di *casi d'uso*.

1.1.4 Limitazioni

L'esperienza accumulata grazie al lavoro continuativo fatto su VDE 2 ci ha permesso di maturare una visione ad ampio spettro del progetto e della direzione da esso presa. Grazie a questa padronanza abbiamo individuato

[16] http://debian.org

[17] http://sourceforge.net

una serie di problematiche architetturali che ci prefiggiamo di risolvere con la nuova struttura del framework proposta in 2.1.

Estendere il framework con nuove funzionalità è laborioso poiché non vi sono molte possibilità di riutilizzare il codice già scritto. Per poter, ad esempio, usare libpcap per inviare e ricevere pacchetti utilizzando un'interfaccia di rete reale in modalità promiscua è stato necessario copiare la quasi totalità del sorgente di `vde_plug2tap` (a sua volta copia di `vde_plug`) ottenendo l'applicazione `vde_pcapplug`.

Un altro problema che abbiamo potuto riscontrare è che i componenti sono molto *accoppiati* tra loro anche per il fatto di essere sviluppati come applicazioni singole piuttosto che come libreria. Risulta in pratica piuttosto difficile estendere le funzionalità delle applicazioni in modo chiaro, con ovvie ricadute sulla manutenibilità del codice in questione e del progetto in generale. Data la mancanza di moduli ben definiti è difficile capire le interazioni tra le varie parti del programma per poterle testare in isolamento, in altre parole anche durante la correzione di un bug non è semplice stabilire che la modifica non introduca nuovi difetti.

Prendendo spunto dall'esempio di `vde_pcapplug` è possibile osservare che questo supporto potrebbe essere ancora più versatile se fosse possibile utilizzare `vde_pcapplug` direttamente all'interno di `vde_switch` o di QEMU invece che invocare separatamente `vde_pcapplug` e `vde_switch` ed in seguito collegarli. Sempre riguardo alla stessa problematica anche libvdeplug dipende da `vde_switch` senza il quale non è possibile collegarsi ad una rete remota o filtrare il traffico in entrata/uscita o utilizzare `slirpvde`. Questo aspetto di dipendenza diretta da `vde_switch` impatta negativamente in contesti nei quali un'applicazione di terze parti voglia includere il supporto a VDE senza avere reali necessità di connettersi a un `vde_switch` sulla macchina locale.

Il passaggio del flusso di dati attraverso processi diversi può risultare vantaggioso in alcuni scenari in cui sia necessario parallelizzare il carico computazionale complessivo effettuato su ogni pacchetto. Esistono però scenari in cui il carico computazionale passa in secondo piano rispetto al

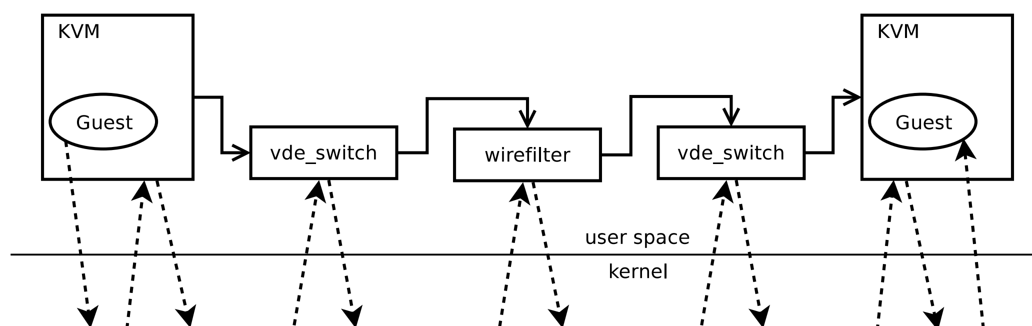


Figura 1.2: Diagramma dei context switch con wirefilter

throughput della rete. In questi casi ciò che penalizza le performance è il numero di *context switch* che un pacchetto deve affrontare per attraversare la catena di componenti virtuali. Si nota immediatamente dalla descrizione dei collegamenti fatta qui sopra come l'attuale architettura imponga un elevato numero di context switch ai pacchetti penalizzando notevolmente il throughput. Come esempio possiamo considerare due macchine virtuali KVM collegate tra loro in modo da simulare una perdita casuale di pacchetti tramite `wirefilter`. Come si può vedere in figura 1.2, in questo caso la catena per consegnare un pacchetto dal kernel di un sistema virtualizzato al kernel dell'altro sistema virtualizzato comprende due `vde_switch` e un `wirefilter` per un totale di cinque processi utente e dodici context switch.

Lo stesso `vde_switch` non offre una facile espansione delle proprie funzionalità: non è possibile ad esempio usare direttamente il protocollo di `vde_cryptcab` oppure collegare un numero arbitrario di interfacce tap senza ricorrere a `vde_plug2tap`. Nonostante il componente centrale del framework abbia una struttura modulare i suoi sottosistemi sono altamente accoppiati tra loro e risultano quindi in una scarsa flessibilità, causa dei problemi sopraccitati. Un'ulteriore prova di ciò sono le politiche di accesso ed autorizzazione allo switch, completamente delegate ai permessi UNIX sul filesystem e quindi legate ai socket UNIX. Questa delega rende poco pratico, se non impossibile, riuscire ad esprimere in modo semplice logiche di accesso che lavorino sulle risorse disponibili nello switch quali porte o

VLAN.

Sebbene in questi ultimi anni si sia lavorato molto sulla parte di controllo del framework vi si possono individuare ancora numerose lacune. Il management può essere fatto solo localmente tramite socket UNIX o da remoto usando complessi wrapper come `vdetelweb`, che esclude la possibilità di automatizzare il controllo da remoto. Non tutti i componenti offrono la possibilità di essere controllati e nessuno di questi usa un modello di testo strutturato ed autodocumentato, complicando inutilmente e limitando il potere delle librerie di management ad oggi presenti.

1.2 Virtual Square

Nonostante VDE possa essere considerato un progetto a sè stante si tratta in realtà della parte networking di un lavoro di ricerca più ampio sulla virtualità. Il progetto Virtual Square [8] si propone come una collezione di tecnologie per *unificare ed espandere* l'idea di virtualità su tanti fronti: dal sistema operativo alle reti. La figura di pagina 21 dà un'idea della complessità attualmente raggiunta dalle parti che compongono Virtual Square.

In questa sezione vengono illustrati i principali componenti del progetto e l'uso che ne può venir fatto.

1.2.1 View-OS

L'idea di View-OS è quella di *«un sistema operativo con vista»*[18] con cui è possibile rompere la *global view assumption* che forza tutti i processi in gestione al sistema operativo a condividere la stessa visione sulle risorse della macchina.

All'interno di View-OS ogni processo ha la propria visione sulla rete, il filesystem, i permessi, i dispositivi ed anche il tempo. Questo tipo di

[18]Vista: uno dei cinque sensi, non Windows Vista.

virtualizzazione è già possibile ma ad un costo molto elevato: virtualizzare completamente tutta l'architettura sottostante e far girare copie diverse dello stesso sistema operativo, un notevole spreco di risorse.

Attualmente View-OS è implementato a livello utente da UMview che utilizza il *tracing delle system call* per virtualizzare le risorse a disposizione, tecnica già impiegata con successo da UML. In questo senso UMview può essere considerato una *macchina virtuale parziale* poiché è in grado mescolare risorse virtuali e fisiche in modo trasparente per fornire una visione coerente ai processi.

Uno dei punti di forza di UMview è quello di poter girare su un kernel Linux non modificato e senza alcun privilegio di amministratore, proprio per facilitare la sperimentazione da parte degli utenti. È stato sviluppato anche un modulo kernel opzionale (KMview) che utilizza le stesse tecniche di UMview offrendo maggiori prestazioni, ma che risulta più invasivo poiché necessita di un kernel modificato.

Internamente UMview ha un'architettura modulare che permette di personalizzare ogni aspetto della virtualizzazione basata su system call. Ad esempio è possibile controllare la creazione di nuovi dispositivi a blocchi, la gestione dei percorsi dei file e la gestione della rete tramite i rispettivi moduli. Ogni modulo può venire caricato a tempo di esecuzione nel caso in cui sia necessario, alternativamente non è presente in memoria evitando di occupare risorse e di causare overhead.

1.2.2 IPN

IPN è un servizio che implementa un meccanismo di *inter-process communication* utilizzando le stesse interfacce e gli stessi protocolli che si trovano nelle comunicazioni di rete. I processi che utilizzano IPN si trovano connessi ad una rete che permette loro di effettuare comunicazioni *many to many*: i messaggi o i pacchetti inviati da un processo su una rete IPN possono essere contemporaneamente consegnati a più processi connessi alla stessa rete e quindi potenzialmente a tutti i processi.

All'interno di un servizio IPN è possibile definire diverse tipologie di protocolli per la consegna dei dati. Il protocollo di base implementa un meccanismo di broadcast: un pacchetto viene inviato a tutti i processi, mittente escluso. L'implementazione di altri protocolli è modulare per cui è molto semplice aggiungere nuovi sistemi di smistamento che possano ad esempio fare dispatching di frame Ethernet permettendo a più macchine virtuali di essere connesse tramite uno switch virtuale direttamente in kernel space.

Un'applicazione può accedere ai meccanismi di IPC offerti da IPN utilizzando la stessa API dei Berkeley socket opportunamente estesa dall'address family `AF_IPN`, oppure tramite un dispositivo a caratteri. Entrambe le interfacce vengono discusse in dettaglio in [6].

1.2.3 Possibili utilizzi

Uscire dalla logica della Global View Assumption permette di eseguire a livello utente alcune operazioni riservate all'amministratore. Tra le più interessanti ci sono quelle privilegiate non perché costituiscano un problema di sicurezza ma solamente perché influiscono sulla visione degli altri processi. Di seguito vengono illustrate alcune delle possibilità offerte da UMview e in generale dalla virtualizzazione a livello di system call[19]:

Mount privati ogni processo può disporre, dove più desidera all'interno della gerarchia di directory, di propri filesystem senza influire sul sistema sottostante: solo quel processo e i suoi figli vedranno i filesystem montati. Il modulo `umfuse` che permette tutto ciò è stato realizzato come un'interfaccia tra il progetto FUSE[20] ed UMview: è possibile utilizzare i moduli già pronti (ext2, iso9660, sshfs, encfs ecc.) ed è facile crearne di nuovi.

Installazione locale un utente normale potrebbe voler utilizzare versioni particolari di un software sul sistema senza per questo scomodare

[19] Si faccia riferimento anche a [9].
[20] `http://fuse.sf.net`

l'amministratore. Molti software offrono già la possibilità di essere installati per il singolo utente piuttosto che per il sistema, tuttavia questa capacità è dovuta al singolo software e non al sistema. Tramite il modulo `viewfs` è possibile montare una directory come *overlay* di un'altra ed effettuare *copy on write* su quest'ultima.

Ad esempio sovrapponendo una directory `/tmp/newroot` a `/` ed utilizzando la normale installazione di pacchetti della distribuzione è possibile installare un software che sarà disponibile solo nell'attuale sessione di UMview. I file modificati in `/` verranno in realtà scritti su `/tmp/newroot`.

Sandboxing un altro aspetto interessante è la sperimentazione e l'analisi di software di cui non si conosce (o non si può conoscere affatto) il funzionamento. Di questi processi è possibile fare il cosiddetto *sandboxing* ossia limitare le azioni che possono effettuare modificando le opportune system call ed eventualmente registrando un *log* delle attività per questioni di auditing e di sicurezza.

Rete privata il modulo `umnet` permette di virtualizzare lo stack di rete permettendo anche di averne più di uno utilizzabile dallo stesso processo. In questo modo è possibile avere una visione personalizzata sulla rete, ad esempio *un singolo processo* può essere collegato con un proprio indirizzo ad una rete VDE esattamente come se fosse una macchina reale o virtuale.

Con l'avvento di IPv6 non è difficile immaginare scenari dove *ogni processo abbia il proprio indirizzo IPv6*. Questo è già possibile poiché umnet utilizza LWIPv6: una libreria che fornisce uno stack di rete completo sia IPv4 che IPv6.

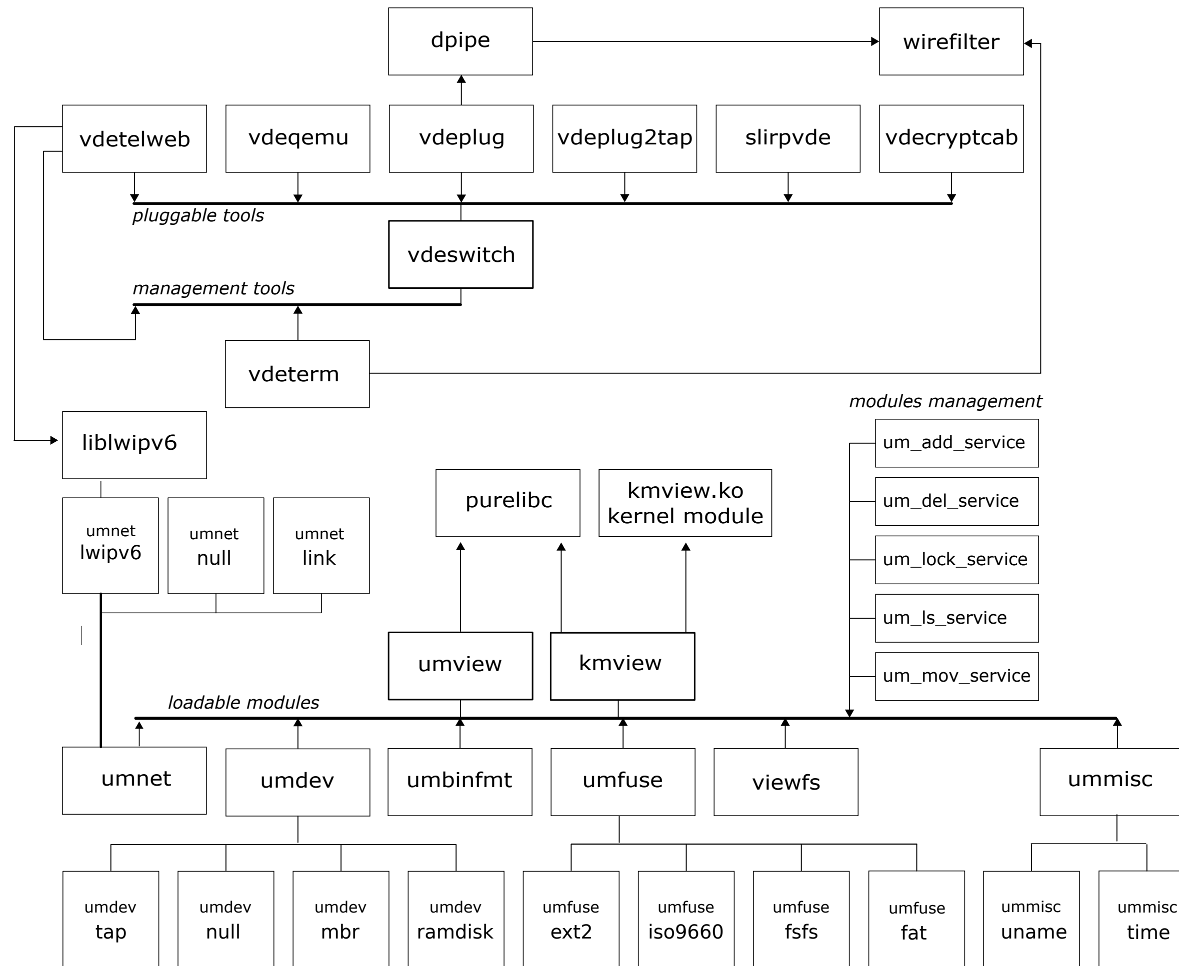


Figura 1.3: Diagramma dei componenti di Virtual Square

Capitolo 2

VDE 3

Il pubblico di utilizzatori del framework VDE è aumentato nel corso degli anni di pari passo con l'importanza della virtualizzazione. Il *feedback* della comunità è stato di grande aiuto nel trovare bug e aggiungere funzionalità, anche se questo ha richiesto di estendere il codice in modo poco naturale rispetto all'architettura originale.

VDE 3 nasce pertanto dall'esigenza di un framework per il networking virtuale con un'architettura estendibile pur garantendo compatibilità all'indietro con VDE 2. Durante il design siamo partiti dai problemi illustrati in 1.1.4 e abbiamo considerato come punti fondamentali la facilità di sviluppo, di estensione e di manutenibilità del codice in aggiunta alle caratteristiche proprie di VDE 2 quali le perfomance e l'interoperabilità con gli standard di rete.

La proposta di una *piattaforma programmabile* rispetto ad una suite di applicazioni non è casuale: ci prefiggiamo di attrarre un pubblico ancora più vasto rendendo allo stesso tempo semplici sia l'implementazione di nuove funzionalità che l'*embedding* di VDE dentro la propria applicazione.

2.1 Architettura

Il framework VDE 3 è incentrato sull'idea di *componenti* che interagiscono tra loro in modi ben definiti. Spetta all'utente scegliere che componenti utilizzare e come disporli all'interno di un nodo VDE 3 per poter adempiere alle funzionalità richieste.

Di seguito vengono illustrate in dettaglio le parti che compongono l'architettura e le loro interazioni.

2.1.1 Scelte architetturali

Ad ogni componente viene delegata una *funzionalità* ben definita all'interno di VDE che consente una chiara separazione delle responsabilità. Sono state inoltre rese esplicite le *relazioni* e le interazioni mediante un'*API interna* che ogni componente deve implementare. Queste assunzioni ci hanno permesso di ridurre al minimo l'accoppiamento tra i componenti rendendoli indipendenti l'uno dall'altro.

Dall'incapsulamento di funzionalità specifiche all'interno di componenti abbiamo derivato flessibilità e di estendibilità: modificare una funzionalità richiede solamente cambiamenti locali al componente che la implementa e risulta al contempo facile fornire VDE di nuove *feature* poiché basta aggiungere un nuovo componente che rispetti le interfacce stabilite.

La creazione di un'API interna ha semplificato il *testing*: un componente può essere esaminato singolarmente ed eventualmente connesso con altri componenti appositamente creati al fine di ricreare particolari interazioni[1] La creazione di componenti artificiali permette ad esempio di isolare gli scenari nel quale un bug si può verificare, cosa piuttosto difficile da fare con l'architettura di VDE 2 a causa dell'alto accoppiamento già discusso in 1.1.4.

Ad un'API interna, specifica dei componenti, abbiamo fatto corrispondere un'API esterna esposta alle applicazioni che intendono utilizzare

[1] Il cosiddetto *mock testing* [34].

VDE 3. In questo modo il framework appare come una libreria[2] grazie alla quale è possibile integrare tutte le funzionalità di VDE direttamente in un'applicazione, che può quindi essere equipaggiata con un nodo di rete virtuale personalizzato. L'implementazione di VDE come libreria non richiede altri processi esterni e rappresenta un netto passo in avanti, eliminando di fatto il requisito di VDE 2 di un `vde_switch` in esecuzione sulla macchina locale.

Il poter disporre di VDE come una libreria di sistema fornisce anche la possibilità di interfacciare `libvde` con linguaggi di *scripting ad alto livello* quali python o perl. Questi linguaggi, data la loro natura dinamica, sono utilizzati nello sviluppo di *prototipi software*, un campo di applicazione che riteniamo molto importante per chi voglia sperimentare le potenzialità messe a disposizione da VDE 3.

Per la gestione delle attività di rete abbiamo utilizzato un *modello a eventi* largamente ispirato a `libevent` [26], un paradigma assai noto in letteratura [23, 33] che permette la creazione di applicazioni di rete efficienti e scalabili ed ha reso VDE completamente asincrono. Questo ha semplificato di molto lo sviluppo e la chiarezza del codice, dando inoltre la possibilità a ciascun componente di monitorare con facilità l'attività sui propri file descriptor tramite la registrazione di funzioni di *callback* invocate al verificarsi dell'evento richiesto. Abbiamo inoltre scelto di non avere nessun meccanismo di gestione degli eventi all'interno di `libvde` ma ne abbiamo delegato l'implementazione all'applicazione esterna per favorire l'integrazione nel flusso di esecuzione del programma principale.

Il controllo remoto dei componenti ha ricevuto particolare attenzione: abbiamo utilizzato protocolli esistenti, integrati con linguaggi ad alto livello e con tipi di dato strutturati. Ci auguriamo che queste scelte incoraggino lo sviluppo di *console di management* personalizzate e l'integrazione della gestione di VDE all'interno di sistemi già esistenti. Per aumentare al massimo la flessibilità della gestione remota è stato dato pieno controllo ad ogni componente sulle funzionalità che può esportare all'esterno.

[2] D'ora in poi `libvde`.

Abbiamo reso infine molto più flessibile la parte di autorizzazione. Ad ogni connessione vengono associati degli *attributi* con i quali è possibile decidere se permettere o negare l'accesso ad una risorsa qualora venga richiesto. Particolari tipi di attributi sono le credenziali di accesso[3] scambiate durante la creazione di una nuova connessione oppure il nome dell'utente che sta accedendo alla risorsa. Questo apre molte possibilità come la differenziazione nell'utilizzo del sistema di management in modo che un utente con privilegi di amministratore possa impartire comandi che modificano lo stato dei componenti, lasciando ad un utente normale solamente la facoltà di controllare o monitorare il nodo senza alterarne lo stato.

2.1.2 Visione d'insieme

Le scelte architetturali discusse in precedenza ci hanno portato alla scrittura di una libreria event-based che consente di creare parti di una rete virtuale con nodi altamente personalizzabili. Presentiamo ora in breve i tratti essenziali della libreria di cui è possibile osservare una rappresentazione grafica in figura 2.1.

Un nodo VDE viene costruito interconnettendo alcuni componenti elementari all'interno di un contesto. Questi componenti sono implementati da moduli caricabili dinamicamente a runtime e possono quindi essere facilmente sviluppati e distribuiti da diversi autori.

Il *contesto* è l'entità preposta alla creazione di nuovi componenti e rappresenta anche l'ambiente di esecuzione di questi. All'interno di un contesto ogni *componente* viene identificato tramite il proprio nome che deve quindi essere univoco per ogni contesto.

Esistono tre tipologie (`kind`) di componenti i cui ruoli possono essere sintetizzati nel seguente modo:

transport si occupa della creazione di canali di comunicazione (connessioni) tra il contesto ed il mondo esterno;

[3] Tipicamente username e password.

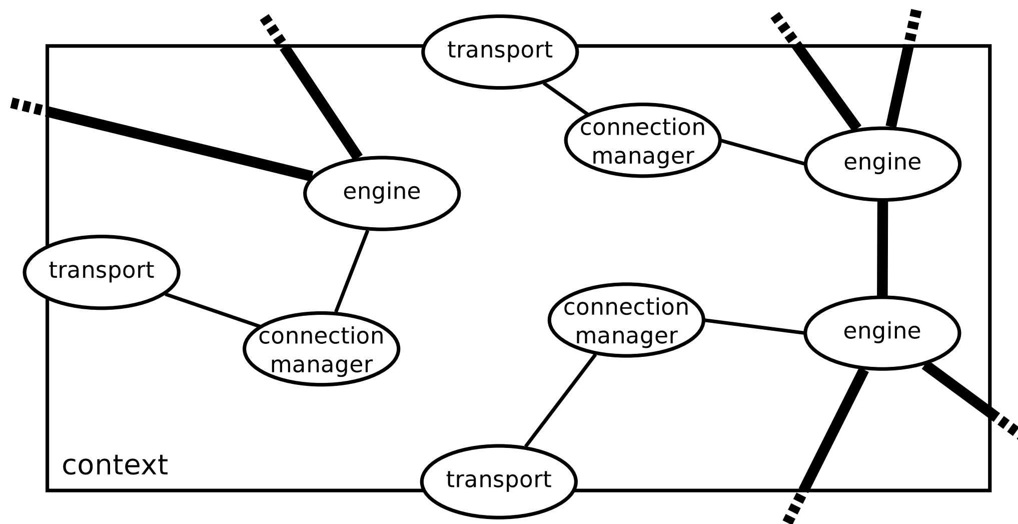


Figura 2.1: Schema di un contesto VDE 3 (le linee sottili rappresentano i legami logici, le linee in grassetto le connessioni)

engine processa i dati da/per le varie connessioni;

connection manager lega un transport ad un engine in modo che nuove connessioni originate dal transport possano essere associate all'engine.

Ogni *modulo* implementa una tipologia di componente. All'interno di un contesto un singolo modulo viene univocamente identificato tramite il proprio `kind` e la propria `family`, una seconda proprietà che permette di distinguere tra differenti implementazioni di uno stesso tipo di componente.

Una *connessione* è un oggetto astratto che viene usato da un `engine` per inviare e ricevere pacchetti di informazioni. Ad ogni connessione viene associato un *backend* che implementa i meccanismi di gestione dei pacchetti in modo del tutto trasparente all'interfaccia di utilizzo della connessione.

Un *pacchetto* è un contenitore strutturato dei dati scambiati tra gli `engine`. Ogni pacchetto al suo interno mantiene i dati come *payload* e vi associa un *header* che descrive alcune proprietà del payload come il tipo di dati e la dimensione.

I componenti possono esporre dinamicamente a runtime alcune loro funzionalità per poter essere *configurati e monitorati remotamente*. Vi sono due categorie di funzionalità remote:

comandi chiamate sincrone verso il componente usate per interrogazioni o modifiche dello stato di quest'ultimo;

segnali chiamate asincrone effettuate dal componente per informare del verificarsi di un evento al suo interno.

Le interfacce di comandi e segnali gestiscono esclusivamente tipi di dati serializzabili permettendo quindi l'implementazione di un sistema di *Remote Procedure Call* come specificato in [35]. È ad esempio possibile creare un `engine` dedicato al management del nodo VDE che invia e riceve dati di controllo scambiando pacchetti su una connessione.

2.1.3 Dettagli implementativi

Passiamo ora ad analizzare più nel dettaglio la libreria che abbiamo sviluppato descrivendo un semplice hub Ethernet in grado di accettare connessioni da VDE 2 riportato in listato 2.1. Per la documentazione formale delle funzioni descritte in seguito si faccia riferimento all'appendice A.

Contesto, moduli ed eventi

I primi passi da compiere sono l'allocazione e l'inizializzazione del contesto per il nodo VDE che stiamo creando, effettuati rispettivamente dalle funzioni *vde_context_new()* e *vde_context_init()*.

Un contesto è principalmente un contenitore di componenti, quindi in fase di allocazione una delle operazioni fondamentali è la creazione della struttura dati preposta ad ospitarli: una hash table in grado di mantenere un ordinamento negli elementi in base al tempo del loro inserimento.

```c
#include <vde3.h>
#include <event.h>

extern vde_event_handler libevent_eh;

int main(int argc, char **argv)
{
  vde_context *ctx;
  vde_component *transport, *engine, *cm;

  event_init();

  vde_context_new(&ctx);
  vde_context_init(ctx, &libevent_eh, NULL);

  vde_context_new_component(ctx, VDE_TRANSPORT, "vde2", "tr1",
                            &transport, "/tmp/vde3_test");

  vde_context_new_component(ctx, VDE_ENGINE, "hub", "e1", &engine);

  vde_context_new_component(ctx, VDE_CONNECTION_MANAGER, "default",
                            "cm1", &cm, transport, engine, 0);

  vde_conn_manager_listen(cm);

  event_dispatch();

  return 0;
}
```

Listato 2.1: Un hub di esempio creato con `libvde`

In questa hash le chiavi sono i nomi dei componenti e i valori sono i componenti stessi.

La fase di inizializzazione prevede la configurazione del contesto in base ai due argomenti passati a *vde_context_init()*: il `modules_path` e il `vde_event_handler`.

Il `modules_path` è un vettore NULL-terminato di percorsi sul filesystem in cui il contesto cerca e carica tutti i moduli VDE, che sono stati implementati come shared-object seguendo le linee guida in [12]. Ogni elemento del vettore può rappresentare un singolo file oppure una directory, che verrà controllata in modo non ricorsivo.

Per capire se uno shared-object contiene un modulo VDE il contesto cerca al suo interno una struttura `vde_module`, riportata in listato 2.2, identificata dal simbolo `vde_module_start`. Si può considerare `vde_module` come un indice, le informazioni ivi contenute riportano infatti il `kind` e la `family` del modulo, le funzioni usate per creare e gestire i componenti ed anche le funzioni specifiche di ogni `kind`.

Il `vde_event_handler`, la cui struttura è riportata in listato 2.3, fornisce al contesto le funzioni per la gestione degli eventi. L'interfaccia che ogni applicazione deve implementare prevede due funzioni per la creazione e la rimozione di eventi per la lettura/scrittura su un file descriptor e due funzioni per la creazione e la rimozione di timer.

Per creare l'hub Ethernet possiamo inizializzare il contesto usando il percorso di ricerca dei moduli predefinito dalla libreria e passando un'implementazione dell'event handler basata su libevent [27].

Componenti

Una volta che il contesto è stato inizializzato è possibile creare nuovi componenti al suo interno chiamando la funzione *vde_context_new_component()*. Nell'esempio usiamo tre componenti: un engine della family `hub`, un transport della family `vde2` e un connection manager della family `default` per collegare i due precedenti.

```
typedef int (*cm_listen)(vde_component *cm);

typedef int (*cm_connect)(vde_component *cm, vde_request *local,
                          vde_request *remote,
                          vde_connect_success_cb success_cb,
                          vde_connect_error_cb error_cb,
                          void *arg);

typedef int (*eng_new_conn)(vde_component *engine,
                            vde_connection *conn,
                            vde_request *req);

typedef int (*tr_listen)(vde_component *transport);

typedef int (*tr_connect)(vde_component *transport,
                          vde_connection *conn);

typedef struct {
  vde_component_kind kind;
  char* family;
  component_ops *cops;
  cm_connect cm_connect;
  cm_listen cm_listen;
  eng_new_conn eng_new_conn;
  tr_connect tr_connect;
  tr_listen tr_listen;
  void *dlhandle;
} vde_module;
```

Listato 2.2: La struttura vde_module e le funzioni specifiche

```
typedef struct {
  void *(*event_add)(int fd, short events,
                     const struct timeval *timeout,
                     event_cb cb, void *arg);

  void (*event_del)(void *ev);

  void *(*timeout_add)(const struct timeval *timeout, short events,
                       event_cb cb, void *arg);

  void (*timeout_del)(void *tout);
} vde_event_handler;
```

Listato 2.3: La struttura vde_event_handler

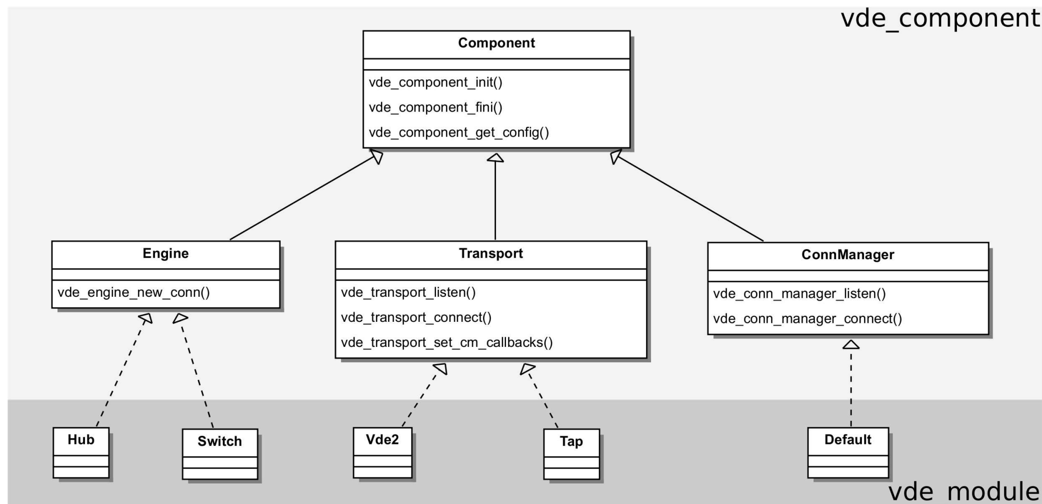


Figura 2.2: Rappresentazione della gerarchia dei componenti

Prima di analizzare l'API interna che ogni tipo di componente deve implementare soffermiamoci un attimo sui ruoli e le relazioni di queste parti della libreria. Abbiamo già detto che esistono tre tipologie di componenti suddivise in `kind` e che un modulo fornisce una `family` in riferimento ad uno dei tre `kind`. Volendo fare un parallelo con i linguaggi di programmazione orientati agli oggetti possiamo ricondurre il *componente* ad una *classe base astratta*, il *`kind`* ad una *sottoclasse astratta* del componente e la *`family`* ad una *sottoclasse* che implementa uno dei tre `kind`. Una rappresentazione di questo parallelo è raffigurata in 2.2.

La struttura dati opaca `vde_component` fornisce tutte le operazioni necessarie per il management di comandi e segnali all'interno di un componente e definisce le interfacce dei vari `kind`. Continuando il parallelo possiamo considerare `vde_component` come fusione della classe base astratta e delle sottoclassi astratte definite qui sopra; conseguentemente possiamo vedere il `vde_module` come la sottoclasse corrispondente alla `family` che implementa le interfacce dei `kind` ed eventualmente le estende.

In aggiunta all'API specifica di ogni `kind` un modulo deve fornire anche una serie di operazioni comuni a tutti i componenti ma intrinsecamente legate all'implementazione della `family`. Fanno parte di questa categoria le

funzioni per l'inizializzazione e la finalizzazione di un componente, paragonabili al costruttore ed al distruttore di una classe e quindi naturalmente dipendenti dalla classe stessa.

Anche le funzioni che consentono di salvare e ripristinare lo stato di un componente devono essere implementate da tutti i moduli. Lo stato deve essere rappresentato da una struttura dati serializzabile in modo che il contesto lo possa salvare o caricare da un file di configurazione.

Transport

I componenti di tipo `transport` sono fondamentalmente dei generatori di connessioni che seguono il ben noto approccio client-server in cui normalmente il transport client risiede in un `vde_context` differente rispetto a quello del transport server. A seguito di una riuscita fase di setup di un canale comunicativo tra i due transport vengono create due connessioni, una per ogni estremo del canale.

Analogamente ai Berkeley socket, l'API che un transport deve implementare include le funzioni *vde_transport_listen()* e *vde_transport_connect()* che servono ad un connection manager per impostare lato server l'attesa dei client e lato client per tentare di stabilire un canale di comunicazione verso un server.

L'utilizzo pratico di queste funzioni è però molto diverso rispetto alle loro controparti nei Berkeley socket a causa della natura asincrona del framework: la *vde_transport_connect()* al ritorno non restituisce una connessione e non esiste l'equivalente della chiamata *accept()*. Il transport processa in modo indipendente queste richieste e utilizzando una serie di callback ne comunica il risultato finale al connection manager che lo gestisce. Il connection manager prima di impartire una listen o una connect deve quindi configurare le proprie callback tramite l'ultima funzione prevista dall'API che il transport deve implementare: *vde_transport_set_cm_callbacks()*.

Engine

Gli `engine` sono il cuore di un nodo VDE 3: sono i componenti che processano i pacchetti da e per le varie connessioni. Un engine può generare un pacchetto, può esserne la destinazione o può inoltrarlo tra una connessione e l'altra eventualmente modificandolo.

Dal punto di vista architetturale un engine deve essere in grado di prendersi carico di una connessione passatagli dal connection manager, deve quindi implementare l'interfaccia *vde_engine_new_conn()*. Deve inoltre essere in grado di restituire una serie di politiche di accesso utilizzate dal connection manager per stabilire se accettare o meno una connessione.

Connection Manager

I componenti di tipo `connection manager` ricoprono un ruolo prettamente gestionale. È tramite essi che l'applicazione istruisce i transport client e server ed è grazie ad essi che le nuove connessioni vengono aggiunte all'engine, passando un eventuale controllo delle autorizzazioni.

L'API del connection manager prevede che vengano implementate le funzioni *vde_conn_manager_listen()* e *vde_conn_manager_connect()* usate dall'applicazione e che vengano forniti i callback chiamati dal transport come descritto in precedenza.

Riprendendo il nostro hub di esempio possiamo vedere come dopo la creazione dei componenti venga chiamata la *vde_conn_manager_listen()* che permette al transport vde2 sottostante di mettersi in ascolto e ricevere nuove connessioni.

Connessioni

Vediamo ora cosa succede quando proviamo a connettere al nostro hub un'applicazione del framework VDE 2, `vde_plug` ad esempio.

Il setup di un canale di comunicazione di tipo VDE 2 richiede che venga effettuata una negoziazione di vari parametri tra il client e il server. Questa fase è intrinsecamente legata alle peculiarità della tecnologia di comunicazione e quindi viene curata dal transport in modo trasparente al resto dell'architettura. Se un canale viene creato con successo il transport si occupa di creare una nuova connessione.

Come già accennato in precedenza, quando il transport inizializza una nuova connessione tramite la funzione *vde_connection_init()* ne configura anche il backend, impostando all'interno della connessione una serie di puntatori alle funzioni che sono in grado di comunicare utilizzando il canale appena stabilito.

Volendo continuare il paragone con l'approccio orientato agli oggetti iniziato in precedenza possiamo considerare la struttura `vde_connection` come una *classe astratta* che definisce un'interfaccia di utilizzo indipendente dal canale di comunicazione e il *backend* come l'implementazione di questa interfaccia.

Una connessione è un oggetto in cui vengono inviati pacchetti e da cui vengono ricevuti in modo asincrono. Per inviare pacchetti un componente usa la funzione *vde_connection_write()*, mentre per riceverli imposta nella connessione la propria *read_cb()* callback utilizzando la funzione *vde_connection_set_callbacks()*. Opzionalmente un componente può impostare anche una *write_cb()* callback che viene invocata dalla connessione per notificare l'avvenuto invio di una serie di pacchetti.

L'astrazione fornita dalla `vde_connection` permette anche di effettuare *connessioni locali* tra due engine all'interno dello stesso contesto. Per convincersi di ciò è sufficiente immaginare un backend che unisce due connessioni registrate su due engine locali: quando il primo engine invoca la *vde_connection_write()* per la connessione registrata presso di sè la funzione di backend che la implementa consegna il pacchetto al secondo engine chiamando la `read_cb` che questi ha registrato nell'altra connessione.

Continuando con l'esempio possiamo vedere in figura 2.3 che quando il transport ha finito il setup e creato la nuova connessione la passa al con-

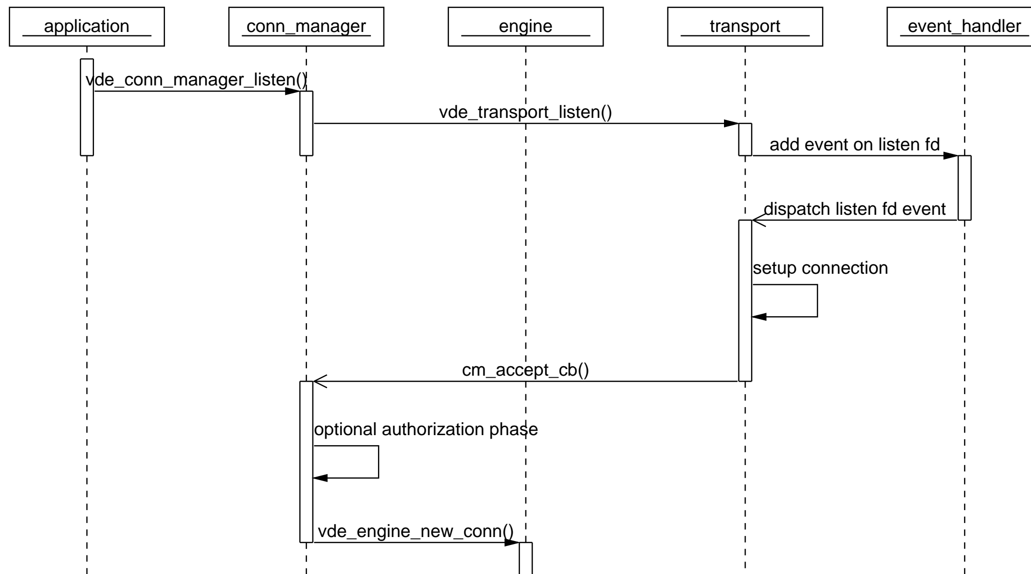


Figura 2.3: Diagramma di sequenza per una nuova connessione

nection manager che lo gestisce invocandone la callback *cm_accept_cb()*. A questo punto la connessione può passare un'opzionale fase di autorizzazione (descritta in seguito) ed essere quindi registrata all'engine chiamando la funzione *vde_engine_new_conn()*.

Una connessione si può chiudere sia dietro richiesta esplicita dell'engine che la sta usando che come conseguenza di un errore fatale nel backend. Nel primo caso l'engine si occupa di chiamare le funzioni *vde_conn_fini()* e *vde_conn_delete()* per chiudere la connessione, nel secondo caso è la connessione ad informare l'engine tramite una callback che gli permette di deregistrare correttamente la connessione.

Pacchetti

Un `vde_pkt` è l'unità di informazione più piccola che un engine possa processare. Ad un pacchetto corrisponde un'area di memoria logicamente contigua suddivisa tra uno *header* della rete VDE 3 contenente alcuni metadati e un *payload* contenente i dati veri e propri. I campi dello header sono:

version indica la versione del formato del pacchetto;

type indica il tipo di dati contenuti nel payload, permette ad esempio di sapere se un pacchetto contiene frame Ethernet, pacchetti IP o informazioni di controllo;

pkt_len indica la dimensione in byte del payload.

Sapere il tipo di payload può essere utile per effettuare controlli ma soprattutto permette a due engine di sfruttare il canale di comunicazione della connessione per scambiarsi informazioni di controllo *in-band* rispetto al flusso dati.

Come abbiamo detto prima e come si può vedere in figura 2.4 quando una connessione ha un `vde_pkt` pronto per essere processato chiama la *read_cb()* dell'engine passando un puntatore al pacchetto; viceversa quando un engine vuole inviare un pacchetto in una connessione chiama la *vde_connection_write()*, passando sempre un puntatore al pacchetto. In entrambi i casi la gestione dell'area di memoria occupata dal pacchetto è responsabilità del chiamante, quindi se il chiamato vuole che il pacchetto venga preservato anche dopo il ritorno della funzione deve farne una copia. Questa politica permette di localizzare all'interno di un singolo oggetto l'allocazione e la deallocazione dei pacchetti semplificando le relazioni tra le diverse parti del codice e permettendo di introdurre ottimizzazioni quali lo sfruttamento dello stack al posto dell'allocazione dinamica nello heap per la ricezione di pacchetti.

Sempre a livello di gestione della memoria un engine mentre processa un pacchetto può necessitare di uno spazio di lavoro più grande del payload originale, ad esempio può dover aggiungere ad un frame Ethernet il tag 802.11Q oppure può dover costruire un frame Ethernet partendo da un pacchetto IP. In questi casi, anziché ricopiare il pacchetto in una nuova area di memoria, l'engine può richiedere alla connessione di preallocare uno spazio di `head` ed uno di `tail` prima e dopo il payload in modo da poter lavorare *in-place*.

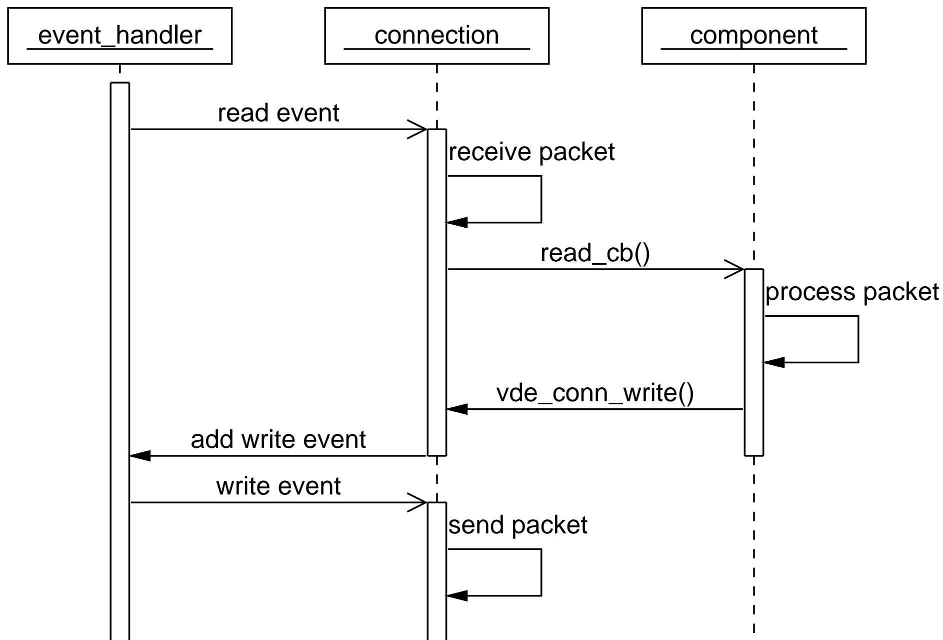


Figura 2.4: Diagramma di sequenza per lo scambio di `vde_pkt`

È possibile che alcuni transport fungano da ponte tra una rete VDE 3 ed un altro tipo di rete che non si scambia `vde_pkt`. Rientrano in questa categoria i transport `vde2` o `tap` che connettono un nodo VDE 3 a una rete basata su frame Ethernet standard. In questi casi è compito del backend della connessione incapsulare/decapsulare i dati all'interno di un `vde_pkt` in fase di ricezione/invio.

Comandi, segnali e serializzazione

A livello funzionale un `vde_command` è assimilabile ad una funzione esposta dal componente nella propria API, mentre un `vde_signal` è assimilabile ad un punto interno al componente di chiamata a callback.

Le differenze fondamentali con funzioni e callback sono dovute al fatto che comandi e segnali devono poter essere *utilizzati ed ispezionati da remoto*, perciò devono trattare esclusivamente tipi di dati serializzabili e devono essere inseriti in una struttura che permetta all'utente remoto di capire a runtime quali funzionalità siano presenti e quale sia la loro sintassi.

Le strutture che contengono rispettivamente comandi e segnali non vengono determinate a tempo di compilazione ma bensì vengono costruite dinamicamente. In questo modo un componente può modificare la lista delle funzionalità remote a seguito di cambiamenti nel proprio stato come l'attivazione di alcune feature o il caricamento di plugin.

Un componente registra un comando tramite la funzione di libreria *vde_component_command_add()* e similmente lo può deregistrare con *vde_component_command_del()*. Il nome del comando deve essere univoco all'interno del componente. Per usare un comando il sottosistema di interfaccia remota deve anzitutto recuperarlo all'interno del componente con *vde_component_command_get()* e quindi invocarne la funzione associata.

Analogamente a quanto avviene per i comandi un componente comunica la facoltà di emanare un determinato tipo di notifiche registrando un segnale tramite la *vde_component_signal_add()* e lo può in seguito deregistrare chiamando *vde_component_signal_del()*. Anche in questo caso il nome del segnale deve essere univoco all'interno del componente. Per ricevere un segnale il sottosistema di interfaccia remota deve registrare una funzione di callback per quel segnale usando la *vde_component_signal_attach()*, per smettere di ricevere quel segnale può passare le stesse informazioni di callback a *vde_component_signal_detach()*. Un componente può chiamare tutti i callback associati ad un segnale invocando *vde_component_signal_raise()*.

Per maggiori dettagli sul sottosistema di interfaccia remota si veda l'implementazione dell'engine di controllo in 2.2.1 e il suo protocollo di comunicazione in 2.4.2.

Autorizzazione

Parlando di connessioni abbiamo detto che il connection manager, quando prende in gestione una connessione, può effettuarne l'autorizzazione prima di registrarla nell'engine.

Come si può vedere nel diagramma in figura 2.5 durante questa fase il connection manager del nodo VDE client invia una *richiesta* al no-

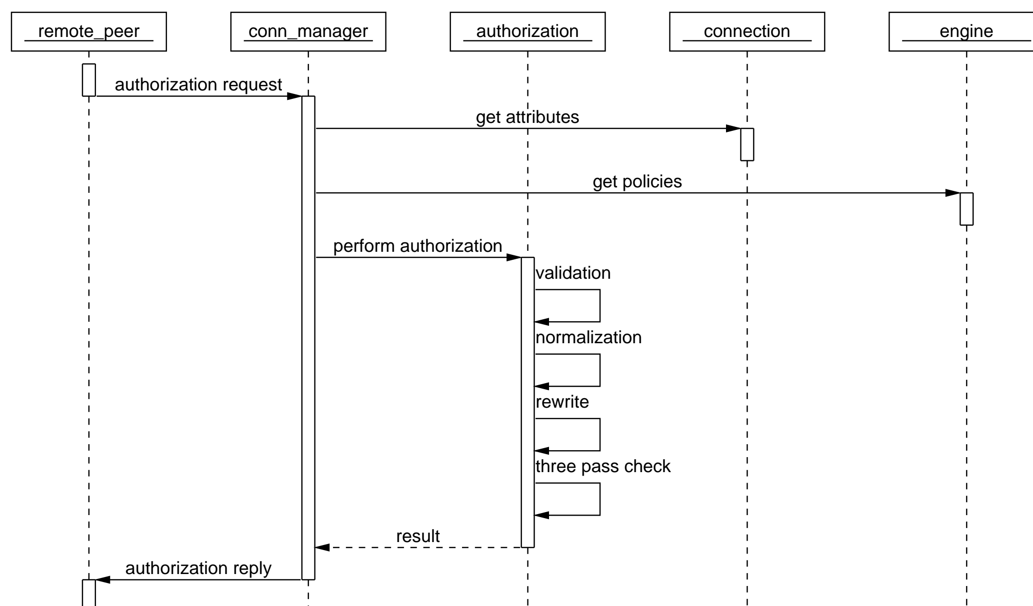


Figura 2.5: Diagramma di sequenza della fase di autorizzazione

do server, ovvero un oggetto contenente una serie di elementi di tipo `chiave : valore` che rappresentano i parametri desiderati con cui la connessione dev'essere registrata nell'engine. Queste informazioni possono ad esempio permettere ad un client di chiedere di essere collegato ad una determinata porta di uno switch Ethernet oppure di essere inserito in una particolare VLAN.

Una volta che il connection manager del nodo server riceve la richiesta dal client recupera gli *attributi* dalla connessione da cui ha ricevuto la richiesta, le *politiche di accesso* dall'engine a cui questa dev'essere registrata e le passa al sottosistema che implementa il meccanismo di autorizzazione.

Analogamente alle richieste gli attributi di una connessione sono sempre una serie di elementi `chiave : valore` che rappresentano alcune proprietà, come ad esempio l'utente autenticatosi per quella connessione. È compito del transport popolare gli attributi durante il setup della connessione.

Le politiche di accesso di un engine hanno una struttura più complessa che permette di esprimere in modo dichiarativo dei concetti quali *"gli*

utenti del gruppo G possono connettersi solamente alla VLAN X" e *"alla VLAN X possono accedere solamente gli utenti del gruppo G"*. Queste due frasi possono sembrare a prima vista molto simili, ma in realtà esprimono politiche molto diverse tra loro: nel primo caso vogliamo limitare l'accesso di una categoria di soggetti ad un solo tipo di risorsa, mentre nel secondo caso vogliamo che una risorsa sia utilizzata esclusivamente da una categoria di soggetti, senza però impedire che questi ultimi possano usare anche altre risorse.

Il meccanismo di autorizzazione è composto da quattro macro operazioni: validazione, normalizzazione, riscrittura e controllo. Anzitutto si esegue la *validazione* per controllare che la richiesta non contenga parametri non contemplati dalle politiche dell'engine. Successivamente la richiesta viene *normalizzata*, ovvero vengono aggiunti al suo interno tutti i parametri di accesso all'engine inizializzati ad un valore predefinito. Si passa quindi alla fase di *riscrittura* durante la quale ogni parametro di accesso può essere riscritto in base agli attributi della connessione, questo permette ad esempio di forzare gli utenti del gruppo G a connettersi alla VLAN X. Si esegue infine il *controllo* dei valori di ogni parametro sempre rispetto agli attributi della connessione per stabilire se una richiesta può passare o meno, questo permette di rispettare altre politiche come impedire ad utenti del gruppo G di usare VLAN diverse da X oppure impedire ad utenti che non fanno parte del gruppo G di usare la VLAN X.

Il controllo finale dei valori implementa un sistema di Discretionary Access Control basato su Access Control List utilizzando una logica a tre passate ispirata al ben noto sistema implementato nel modulo `mod_access` del webserver Apache[4]. Il risultato finale dipende da tre direttive: allow, deny e order. In *allow* si inserisce una lista di elementi `chiave : valore`, se uno degli elementi trova corrispondenza negli attributi della connessione allora la connessione può essere autorizzata. Analogamente anche in *deny* si inserisce una lista di elementi `chiave : valore`, in questo caso però se uno degli elementi trova corrispondenza negli attributi della connessione allora la connessione può essere rifiutata. La direttiva *order* infine

[4] http://httpd.apache.org/docs/2.0/mod/mod_access.html

stabilisce l'ordine di valutazione delle due direttive precedenti in uno dei seguenti modi:

allow, deny Prima vengono esaminati tutti gli elementi nella direttiva allow; almeno uno deve avere corrispondenza, in caso contrario la richiesta viene rifiutata. Successivamente vengono esaminati tutti gli elementi della direttiva deny, se viene trovata una corrispondenza la richiesta viene rifiutata. Infine se non viene trovata nessuna corrispondenza in entrambe le direttive la richiesta viene rifiutata.

deny, allow Prima vengono esaminati tutti gli elementi della direttiva deny; se viene trovata corrispondenza la richiesta viene rifiutata a meno che non venga trovata una corrispondenza anche nella direttiva allow. Infine se non viene trovata nessuna corrispondenza in entrambe le direttive la richiesta viene accettata.

Alla fine del processo di autorizzazione il connection manager del nodo server invia una risposta al connection manager del nodo client per informarlo del risultato. Se l'autorizzazione è stata ottenuta entrambi i connection manager registrano la connessione al proprio engine, in caso contrario la chiudono.

2.2 Confronto con VDE 2

Nel corso della progettazione della nuova architettura abbiamo costantemente validato le nostre bozze rispetto a VDE 2 sul piano concettuale. Lo scopo del confronto è sia controllare che tutte le feature del vecchio framework siano presenti nella nuova libreria ma anche garantire che vi sia un buon grado di *retrocompatibilità*.

La possibilità di reimplementare i tool di VDE 2 utilizzando `libvde` è fondamentale per garantire continuità nel passaggio al nuovo framework, poiché permette agli utenti di continuare ad utilizzare le vecchie configurazioni riferendosi alla knowledge base attualmente esistente per VDE 2.

Applicazione VDE 2		Transport VDE 3
vde_plug	→	vde2
vde_cryptcab	→	ssl_udp
vde_over_ns	→	dns
vde_plug2tap	→	tap
vde_pcapplug	→	pcap

Tabella 2.1: Corrispondenze tra VDE 2 e VDE 3 (transport)

In aggiunta le reimplementazioni forniscono anche un ottimo ventaglio di esempi di utilizzo di libvde.

2.2.1 Comparazione funzionale

Vediamo ora di catturare le peculiarità essenziali di ogni vecchio applicativo e di ricondurle con una breve descrizione al nuovo framework.

Le varie tipologie di *plug* o di *cavo* veicolano il traffico di rete e trovano quindi la loro controparte nei *transport*. Come si può notare in tabella 2.1 scrivendo i transport vde2 (che usa i socket unix come VDE 2), ssl_udp, dns, pcap e tap è possibile veicolare il traffico nello stesso modo rispettivamente a vde_plug, vde_cryptcab, vde_over_ns, vde_pcapplug e vde_plug2tap. Aggiungendo un transport stdio è inoltre possibile riutilizzare la pipe bidirezionale dpipe per creare cavi basati su ssh o netcat.

Facendo riferimento alla tabella 2.2 vediamo come i sistemi che processano pacchetti possano essere resi tramite opportuni *engine*: a vde_switch in modalità hub o in modalità switch corrispondono rispettivamente un engine hub e un engine switch; wirefilter si riconduce ad un engine container di filtri; vde_l3 ad un engine che permette di passare da pacchetti layer 2 a pacchetti layer 3 e viceversa unita ad un engine router per layer 3; slirpvde ad un engine slirp che da un lato accetta connessioni VDE e

Applicazione VDE 2		Engine VDE 3
vde_switch	→	hub switch
wirefilter	→	filter_container
vde_l3	→	l2l3 l3router
slirpvde	→	slirp
kvde_switch	→	kvde_ctl

Tabella 2.2: Corrispondenze tra VDE 2 e VDE 3 (engine)

Controllo VDE 2		Controllo VDE 3
unixterm vdeterm vdecmd vdetelweb	→	linguaggio di alto livello control engine

Tabella 2.3: Corrispondenze tra VDE 2 e VDE 3 (controllo)

dall'altro apre normali socket di rete e infine kvde_switch ad un engine che si limita a controllare lo switch in kernel space.

Come risulta evidente dalla tabella 2.3 tutti i vecchi sistemi di management possono essere sostituiti con brevi script redatti in un qualsiasi linguaggio di programmazione ad alto livello che fornisca le funzionalità necessarie per processare l'output strutturato che deriva dalla serializzazione degli oggetti gestiti dal nuovo sottosistema di controllo. Per riutilizzare al massimo l'architettura di libvde questi script possono dialogare con i componenti di un nodo connettendosi tramite un transport ad un *engine di controllo*. Questo engine da un lato usa vde_connection per scambiare vde_packet contenenti il testo strutturato, mentre dall'altro si occupa di fare dispatching agli opportuni componenti dei comandi e dei segnali derivati dalla deserializzazione delle stringhe ricevute.

Alcuni applicativi del vecchio framework permettono all'avvio di caricare un file di *configurazione* contenente una serie di comandi che impostano

lo stato dell'applicativo stesso. `libvde` permette di configurare lo stato di ogni componente in modo *dichiarativo* tramite strutture dati serializzabili che possono essere facilmente scritte in un file di configurazione.

2.2.2 Reimplementazione con `libvde`

Passiamo ora ad analizzare il livello di retrocompatibilità che è possibile ottenere reimplementando a scopo di esempio due delle parti maggiormente usate di VDE 2: `vde_switch` e `libvdeplug`.

La riscrittura di `vde_switch` è fondamentale anzitutto perché si tratta del componente centrale del vecchio framework ma anche perché racchiude al suo interno un ampio set di funzionalità. Per emularlo possiamo configurare un contesto con un transport `vde2` in modalità server, un transport `tap` e alternativamente un engine hub o switch. Al fine di mantenere piena compatibilità con la parte di controllo si può considerare l'implementazione di un engine di controllo che anziché inviare pacchetti contenenti i dati serializzati li riformatta secondo lo stile sintattico utilizzato in VDE 2. Per ottenere la console da riga di comando o tramite il socket di management possiamo collegare al suddetto engine di controllo un transport `stdio` equivalente a quello discusso in precedenza oppure un transport `unix` che legge e scrive su un socket UNIX di tipo stream.

Il mantenimento dell'interfaccia di `libvdeplug` è anch'esso importante poiché permette alle applicazioni che si connettono a VDE 2 di continuare a connettersi anche alla versione successiva. Abbiamo considerato due approcci: il primo consiste nel conservare il codice di `libvdeplug` invariato ed includerlo nel nuovo framework, trattandosi di una libreria piccola e autocontenuta.

Il secondo approccio invece prevede di costruire `libvdeplug` utilizzando `libvde` e fornisce quindi un esempio di come sia possibile connubiare un approccio event-based (asincrono) con un'API sincrona. Dovendo mantenere la compatibilità è necessario mascherare un'istanza di VDE 3

all'interno di una connessione aperta da `libvdeplug`, pertanto si inseriscono il contesto e l'event-handler all'interno di `VDECONN`. In questo modo durante l'apertura di una nuova connessione la libreria di compatibilità si occupa di creare e gestire gli eventi in modo trasparente all'applicazione e una volta effettuato il collegamento lascia il controllo dei file descriptor al chiamante.

2.3 Analisi delle prestazioni

VDE 3 è flessibile e versatile, ma è anche desiderabile che un nodo VDE sia reattivo e consumi poche risorse. Durante lo sviluppo del nuovo framework abbiamo costruito un ambiente di testing che ci ha permesso di effettuare una serie di esperimenti controllati e ripetibili da cui abbiamo raccolto dati per effettuare paragoni con altre tecnologie e indirizzare l'ottimizzazione.

Come recenti studi hanno ulteriormente confermato [37], non basandosi su misurazioni certe, ma solo su intuizioni personali e su una conoscenza parziale o non aggiornata dei sistemi utilizzati è altamente probabile introdurre ottimizzazioni che complicano inutilmente il progetto senza portare vantaggi significativi o in alcuni casi inserire soluzioni che hanno effetti negativi.

2.3.1 Ambiente di testing

Nello scegliere le metodologie di testing ci siamo indirizzati verso soluzioni che potessero mettere alla prova la struttura di un contesto VDE 3 e conseguentemente anche l'architettura. Ad esempio abbiamo deciso non misurare la trasmissione dati tra due nodi remoti, poiché dipendente dall'implementazione di un transport. Abbiamo inoltre cercato di avere un sistema che ci permettesse la comparazione con le tecniche che ad oggi rappresentano lo stato dell'arte per la creazione di reti tra più macchine virtuali

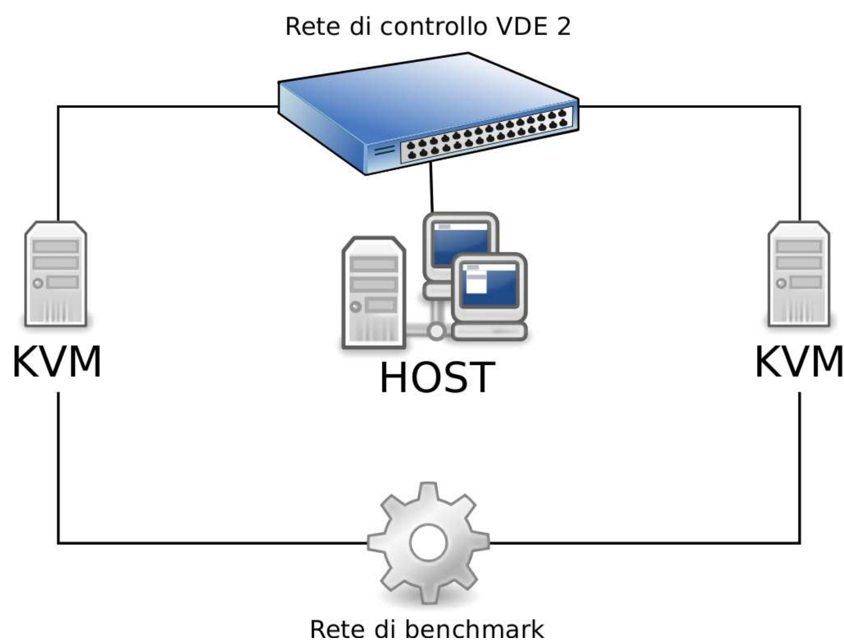


Figura 2.6: Schematizzazione dell'ambiente di testing

che condividono un singolo host fisico, settore in cui le performance sono estremamente cruciali.

L'ambiente creato, rappresentato in figura 2.6 è composto da due macchine virtuali KVM ospitanti due istanze Debian GNU/Linux e connesse a due reti virtuali: una rete di controllo connessa anche con l'host fisico tramite cui le macchine vengono comandate e monitorate e una rete di benchmark a cui sono state connesse solo le due macchine virtuali usata per gli esperimenti. La rete di benchmark è stata configurata utilizzando le seguenti tecnologie:

vde3_hub il piccolo hub di prova descritto in 2.1 costruito inizializzando un engine hub e un transport vde2 in un contesto `libvde`;

vde3_hub2hub un apparato di rete creato con `libvde` in cui il contesto viene inizializzato con due transport vde2 collegati a due engine hub, a loro volta connessi tramite una local connection. In questo caso ogni macchina virtuale viene connessa ad un transport diverso;

vde2_switch `vde_switch` di VDE 2;

vde2_hub `vde_switch` di VDE 2 eseguito in modalità hub;

vde2_wf una rete creata connettendo due `vde_switch` tra loro attraverso un `wirefilter` in cui non è stato impostato nessun filtro. Anche in questo caso ogni macchina virtuale viene connessa ad uno switch diverso;

kvde uno switch in kernel space costruito utilizzando IPN;

bridge due interfacce tap connesse tra loro utilizzando il Linux Bridge;

virtio due interfacce tap connesse tra loro utilizzando il Linux Bridge come nel caso precedente ma con in più l'utilizzo del sistema di paravirtualizzazione virtio[5] presente in KVM;

I test sono stati eseguiti su un Apple MacBook con un processore Intel Core 2 Duo T7500 a 2.2 GHz e 4 GB di RAM DDR2 a 667 MHz equipaggiato con Debian GNU/Linux unstable, kernel 2.6.33-rc3 ed eglibc 2.10.2-5. I sistemi guest montano Debian GNU/Linux 5.0 con kernel 2.6.30.8. Sono stati utilizzati VDE 2 da svn revisione 401 e KVM userspace versione 0.11.1. Il NIC model utilizzato in KVM è sempre stato e1000, tranne ovviamente per virtio, ed al fine di ottenere una situazione che rispecchi maggiormente scenari reali ogni macchina virtuale è stata accoppiata ad un core differente della CPU.

La generazione e la misurazione dei dati è stata effettuata utilizzando Iperf[6] 2.0.4, netperf[7] 2.4.4 e dstat[8] 0.7.0.

Per rendere gli esperimenti facilmente ripetibili abbiamo creato una serie di script, riportati in appendice B, che hanno automatizzato il setup dell'ambiente e l'esecuzione dei test.

[5] http://www.linux-kvm.org/page/Virtio
[6] http://sourceforge.net/projects/iperf/
[7] http://www.netperf.org/
[8] http://dag.wieers.com/home-made/dstat/

2.3.2 Misurazioni effettuate

I test che abbiamo effettuato sono stati utili per misurare su varie dimensioni le performance. In particolar modo ci siamo concentrati sulle seguenti sperimentazioni:

bandwidth per vedere la capacità effettivamente fruibile dalle applicazioni utenti della rete abbiamo riprodotto un flusso TCP da una macchina virtuale all'altra mutando di volta in volta la grandezza dei dati[9] inviati e abbiamo misurato il numero di bit trasmessi al secondo;

latency per stabilire quale tecnologia sia più responsiva abbiamo riprodotto un flusso di datagrammi UDP aventi la dimensione minima al fine di ottenere la minor latenza possibile sulla rete e abbiamo misurato il numero di scambi al secondo;

resources per determinare quale tecnologia fa un uso più efficiente delle risorse abbiamo riprodotto un flusso UDP fissando il numero di bit trasmessi al secondo[10] ad una soglia raggiungibile da tutte le tipologie di rete senza che vi sia perdita di pacchetti e abbiamo misurato i context switch e il tempo di CPU a livello utente e di sistema.

Per aumentare la confidenza nei dati da analizzare ogni esperimento è stato ripetuto più volte calcolando media e deviazione standard sui risultati. Abbiamo notato che dopo poche iterazioni i valori raggiungevano livelli di confidenza accettabili, quindi abbiamo fissato il numero di ripetizioni a cinque.

Come si è potuto osservare per misurare il bandwidth abbiamo utilizzato TCP anziché UDP. La scelta di un protocollo più complesso, potenzialmente dipendente da più variabili, è stata dettata dalla necessità di misurare il flusso sui dati effettivamente trasmessi senza dover considerare anche i pacchetti persi dagli stack di rete.

[9]Sono stati usati blocchi da: 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16834, 32768 e 65536 byte.

[10]Il bandwidth è stato limitato a 70,000,000 bit/secondo.

Naturalmente i risultati ottenuti, seppur significativi, derivano da una condizione di traffico artificiale che non riproduce fedelmente uno scenario di utilizzo reale. Questo è particolarmente significativo nella misurazione della responsività poiché sarebbe necessario confrontare l'andamento di questi valori con l'aumentare del numero di macchine virtuali connesse ad una stessa rete.

2.3.3 Analisi dei risultati

Vediamo ora di trarre alcune considerazioni comparando le varie tecnologie testate con l'ausilio visivo di alcuni grafici che rappresentano i rapporti tra le seguenti grandezze: numero di richieste al secondo, bandwidth con blocksize massimo, andamento del bandwidth rispetto al blocksize, numero di context switch, utilizzo percentuale di CPU utente e CPU sistema. Leggendo i grafici riguardanti il consumo delle risorse di sistema è necessario tener presente che nei primi 10 - 12 secondi vi è un transiente dovuto all'inizio dei test.

vde2_hub - vde3_hub

Compariamo anzitutto il componente più semplice che è possibile ottenere con VDE 2 e la sua controparte implementata in VDE 3 per dare maggior risalto alle differenze architetturali.

Possiamo notare (figura 2.7) come su blocchi di piccole dimensioni non vi sia differenza di *bandwidth*, ma quando il carico aumenta notevolmente vde3_hub riesce a smistare più traffico. Il numero di *richieste al secondo* (figura 2.8a) è superiore in vde3_hub, probabilmente conseguenza di una maggior responsività del modello ad eventi.

Non vi è infine differenza sostanziale nell'*utilizzo delle risorse* (figura 2.9), ma si può notare l'effetto del *tight loop* dell'event handler di vde3_hub: il sistema spende meno tempo in userspace ed il controllo passa più spesso al kernel causando un numero leggermente superiore di context switch.

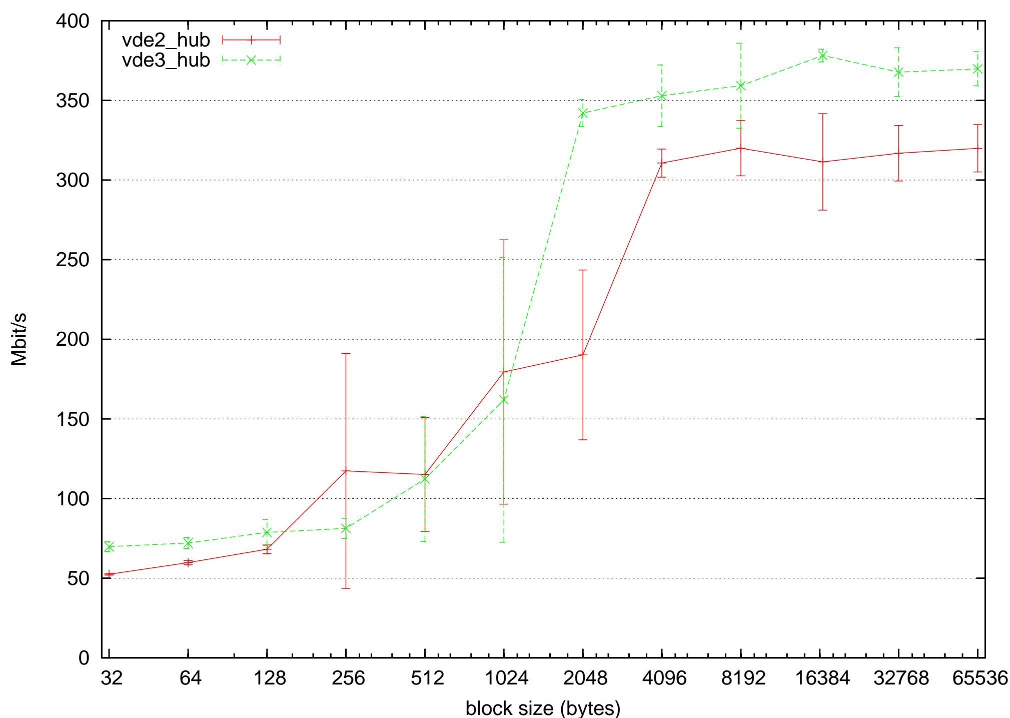
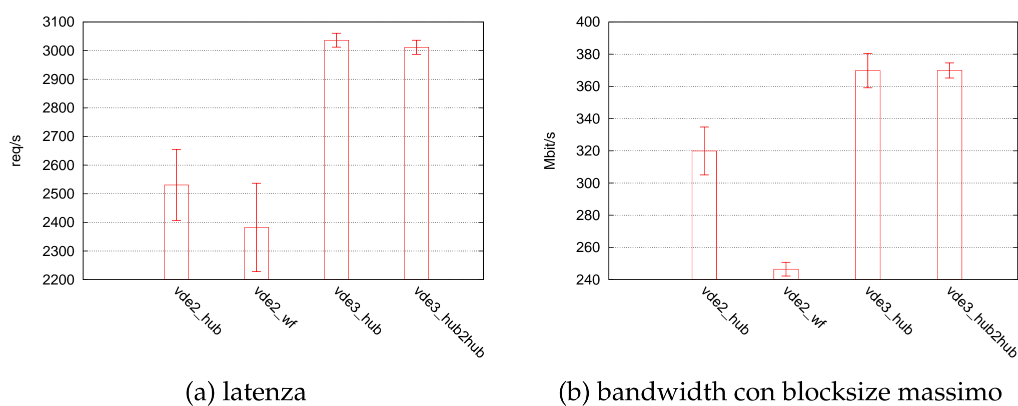
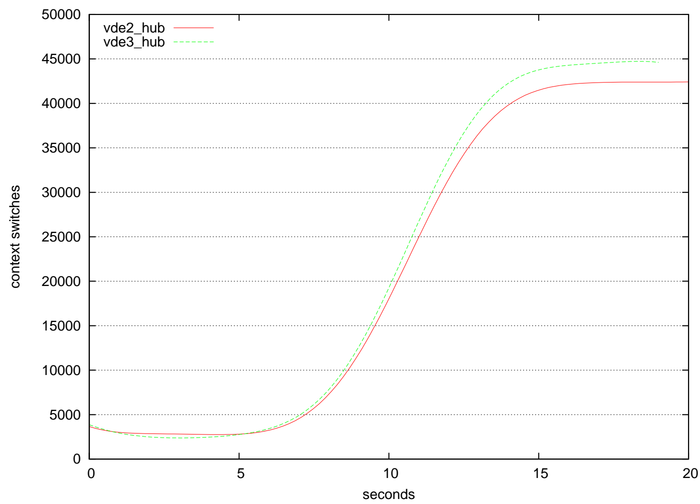


Figura 2.7: Utilizzo di bandwidth di vde2_hub e vde3_hub

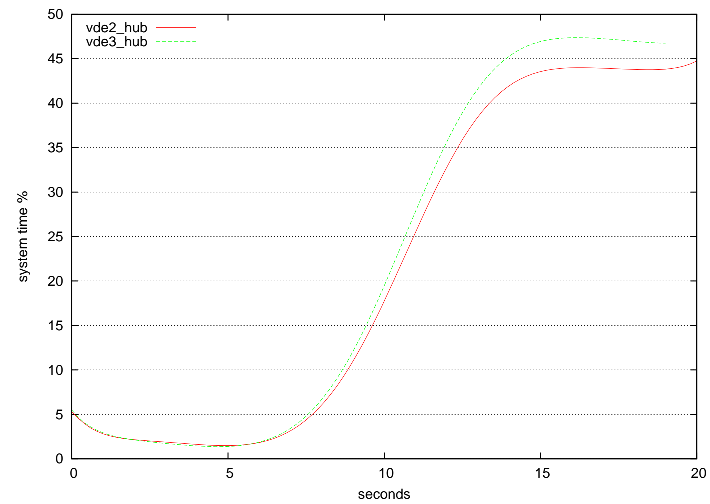


(a) latenza (b) bandwidth con blocksize massimo

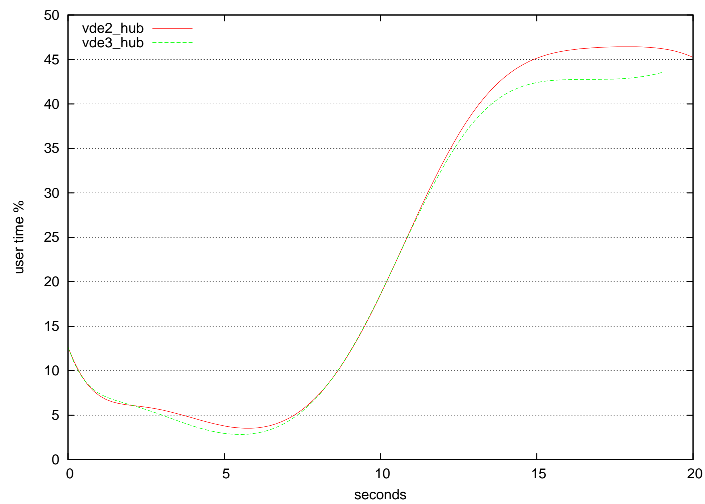
Figura 2.8: Bandwidth e latenza delle reti VDE 2 e VDE 3



(a) numero di context switch



(b) tempo di CPU sistema



(c) tempo di CPU utente

Figura 2.9: Utilizzo delle risorse di vde2_hub e vde3_hub

vde2_wf - vde3_hub2hub

Confrontiamo ora l'approccio a più processi con l'approccio a più componenti in un singolo processo paragonando vde2_wf con vde3_hub2hub. Entrambe le configurazioni sono state approntate con lo scopo di inserire un filtro (inattivo) tra due macchine virtuali. Abbiamo utilizzato vde3_hub2hub anziché un nodo con filtro poiché un engine di filtering non è ancora stata implementata e quindi abbiamo creato un'applicazione computazionalmente equivalente costituita da un engine di smistamento dei pacchetti collegato localmente ad un altro engine che in questa configurazione possiamo ritenere funzionalmente comparabile ad un engine di filtering.

Osservando i risultati sul *bandwidth* (figura 2.10) possiamo notare, tramite una comparazione aggiuntiva con vde2_hub e vde3_hub, come l'aggiunta di un componente non impatti in modo sostanziale su VDE 3. Le performance di VDE 2 invece calano enormemente saturando molto prima la capacità disponibile. Possiamo anche notare che vde2_wf, prima di saturare, ottiene risultati migliori rispetto allo stesso vde2_hub quasi sicuramente a causa della parallelizzazione nello smistamento dei dati dovuta all'aver più processi in gioco.

La struttura a più processi impatta negativamente anche sul numero di *richieste al secondo* (figura 2.8a): la responsività della rete VDE 3 continua ad essere superiore e si nota inoltre come l'incremento della latenza tra vde3_hub e vde3_hub2hub sia sensibilmente inferiore se paragonato all'incremento tra vde2_hub e vde3_wf.

Dal punto di vista dell'*utilizzo delle risorse* possiamo notare (figura 2.11) come il tempo di CPU utente sia simile tra vde2_wf e vde3_hub2hub, ma si osserva subito un numero molto elevato di context switch effettuati da vde2_wf a causa dei tanti processi in gioco. Il tempo di CPU sistema infine è sensibilmente più elevato in vde2_wf, probabile conseguenza dei numerosi context switch.

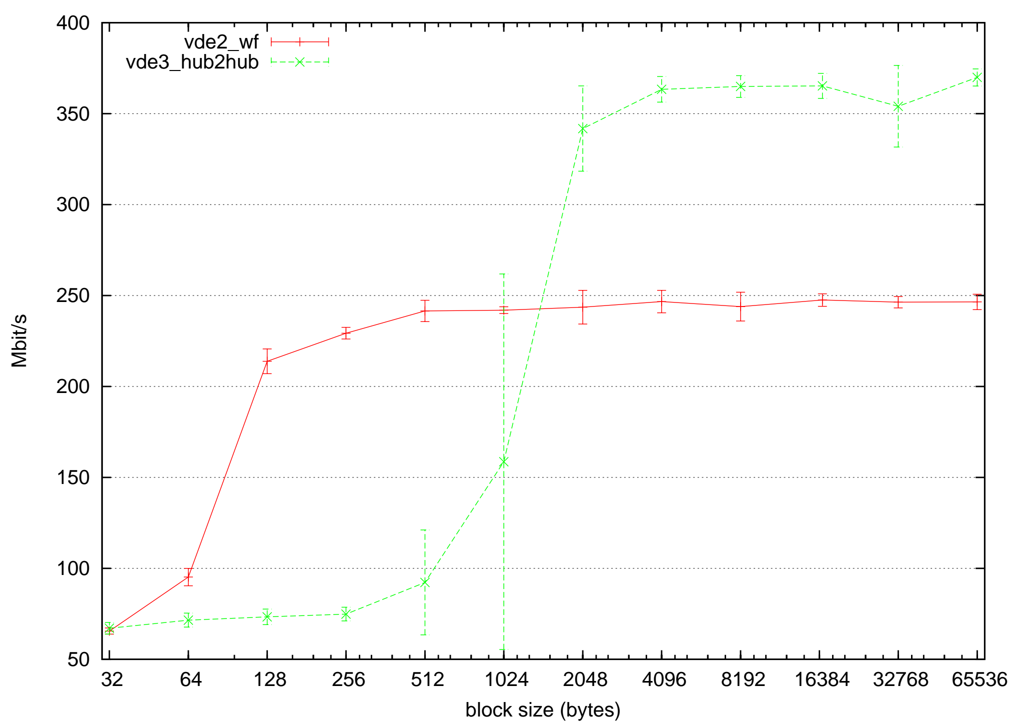
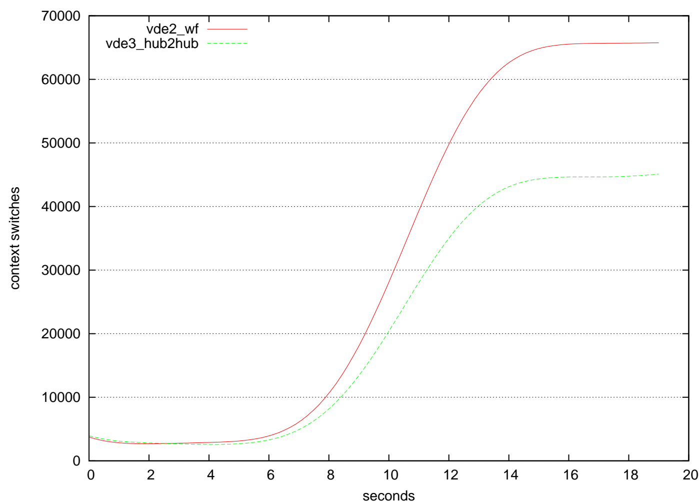
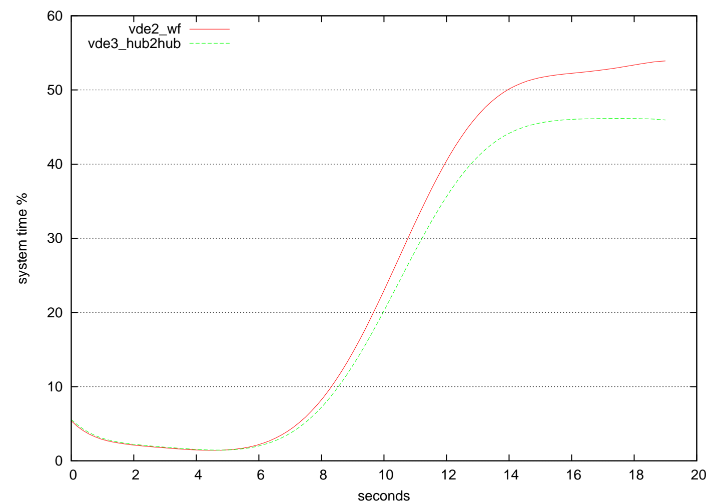


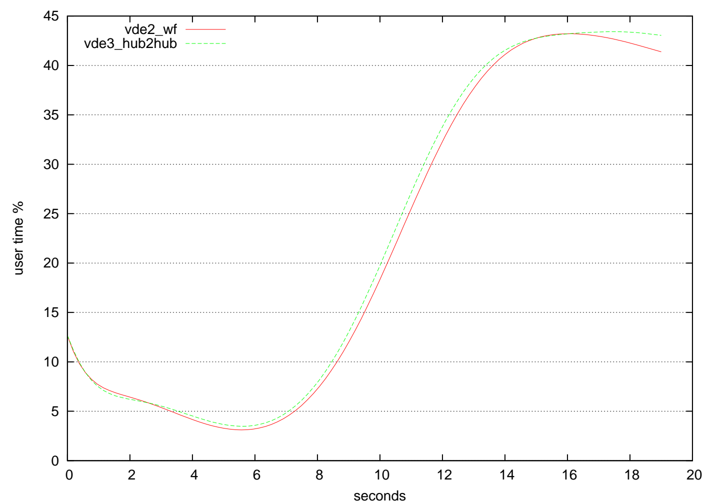
Figura 2.10: Utilizzo di bandwidth di vde2_wf e vde3_hub2hub



(a) numero di context switch



(b) tempo di CPU sistema



(c) tempo di CPU utente

Figura 2.11: Utilizzo delle risorse di vde2_wf e vde3_hub2hub

vde2_switch - vde3_hub - kvde

Dopo aver effettuato comparazioni con valenza prettamente architetturale passiamo ora a confrontare vde3_hub con vde2_switch e kvde, rispettivamente il tool di punta della suite VDE 2 e lo switch sperimentale in kernel space.

Osservando i risultati sul *bandwidth* (figura 2.12) si nota come i valori raggiunti da kvde restino molto bassi, probabilmente a perché il sistema resta a lungo bloccato in kernel space dalla rete IPN. Si vede anche che con blocchi piccoli kvde abbia performance superiori, probabilmente a causa del minor numero di context switch che un pacchetto deve affrontare.

Sul piano delle *richieste al secondo* (figura 2.13a) osserviamo che kvde riesce ad avere una latenza inferiore, probabilmente sempre a causa dei ridotti context switch e che vde3_hub si posiziona circa a metà tra vde2_switch e kvde.

Come nel confronto tra vde2_hub e vde3_hub non notiamo differenze apprezzabili tra vde2_switch e vde3_hub nell'*utilizzo delle risorse* (figura 2.14), mentre nel caso di kvde osserviamo l'impatto di una soluzione in kernel space nel numero ridotto di context switch e nel maggior impiego di CPU sistema.

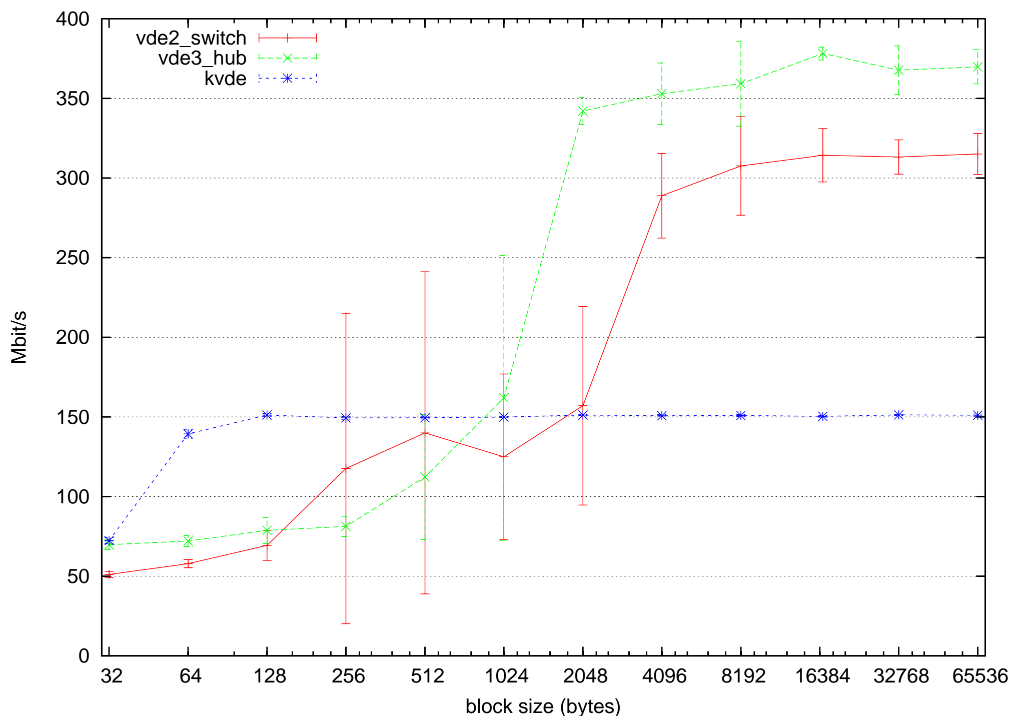
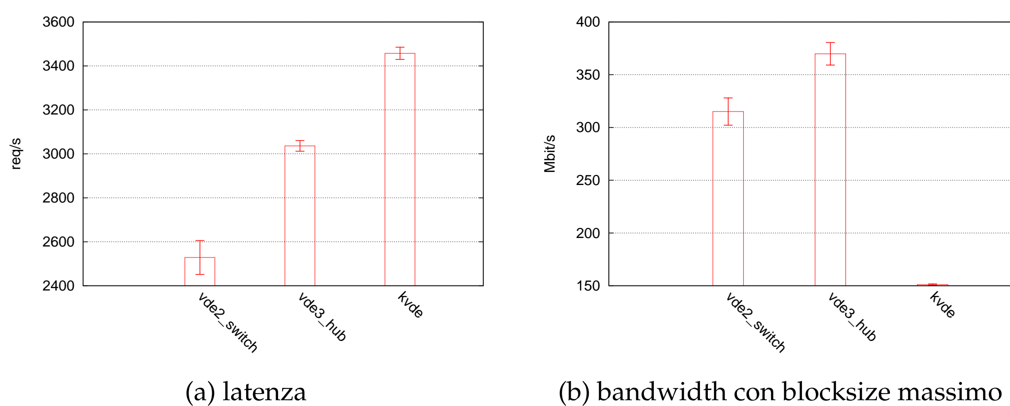


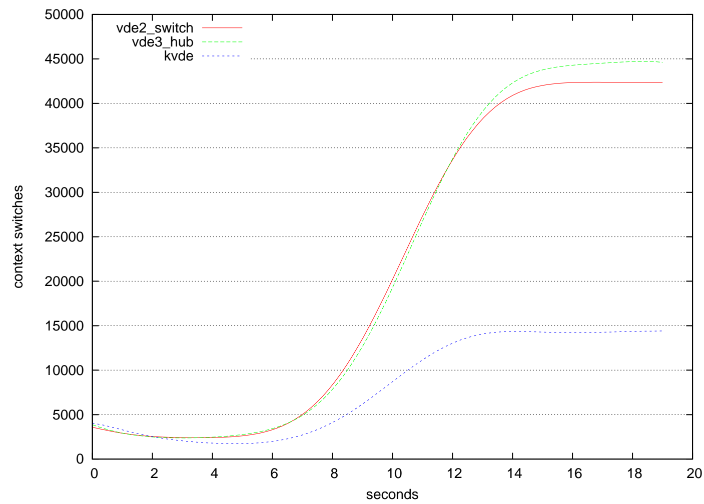
Figura 2.12: Utilizzo di bandwidth di vde2_switch, vde3_hub e kvde



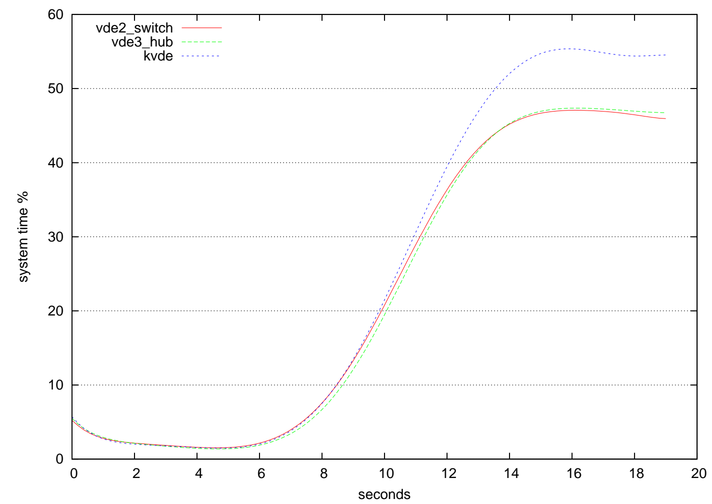
(a) latenza

(b) bandwidth con blocksize massimo

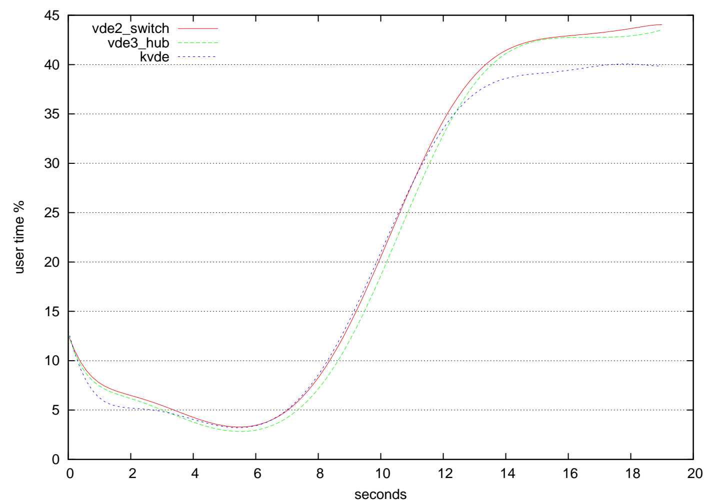
Figura 2.13: Bandwidth e latenza di vde2_switch, vde3_hub e kvde



(a) numero di context switch



(b) tempo di CPU sistema



(c) tempo di CPU utente

Figura 2.14: Utilizzo delle risorse di vde2_switch, vde3_hub e kvde

vde3_hub - kvde - bridge

Vediamo ora come si relazionano vde3_hub e kvde con il sistema utilizzato nel kernel Linux per connettere reti virtuali e reali.

Come prevedibile il *bandwidth* (figura 2.15) raggiungibile con una soluzione basata sullo stack di rete nativo del kernel linux è di molto superiore rispetto a vde3_hub e soprattutto a kvde.

Il bridge riesce però a soddisfare meno *richieste al secondo* (figura 2.16a) sia rispetto a kvde che rispetto a vde3_hub. Al momento non sono chiare le motivazioni di questo comportamento che possono andare da peculiarità intrinseche allo stack di rete di Linux a problemi nell'implementazione del backend tap di KVM.

Sul piano dell'*utilizzo delle risorse* (figura 2.17) notiamo come il numero di context switch del brige sia paragonabile a kvde e di molto inferiore rispetto a vde3_hub, si osserva anche che bridge utilizza molto meno la CPU sistema sia rispetto a kvde che a vde3_hub probabilmente a causa di uno stack estremamente ottimizzato.

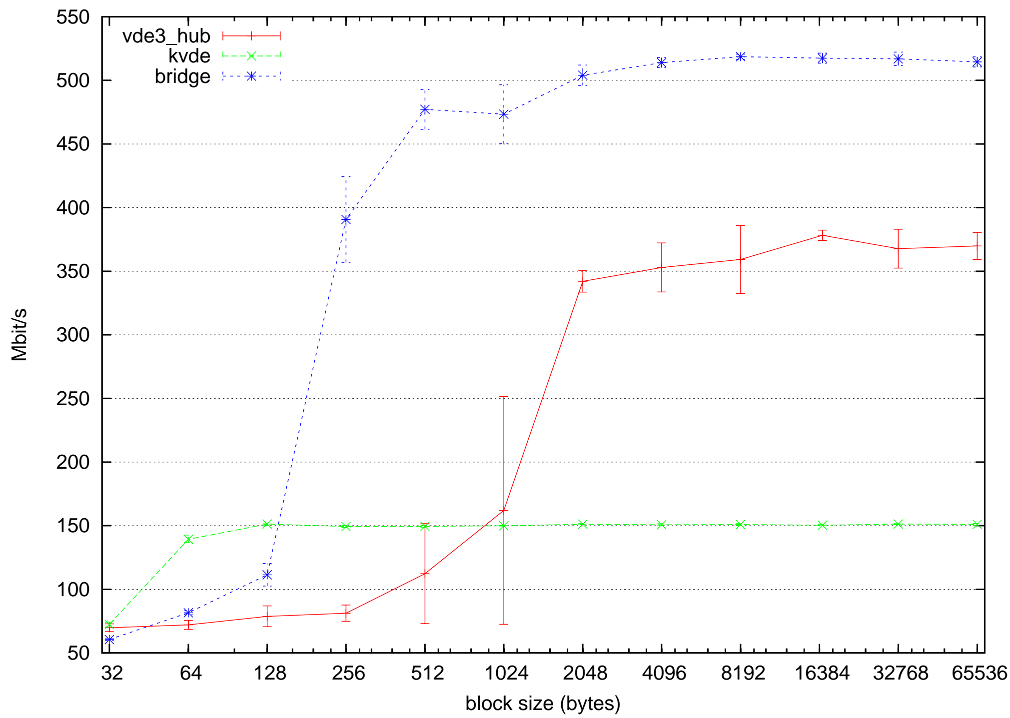
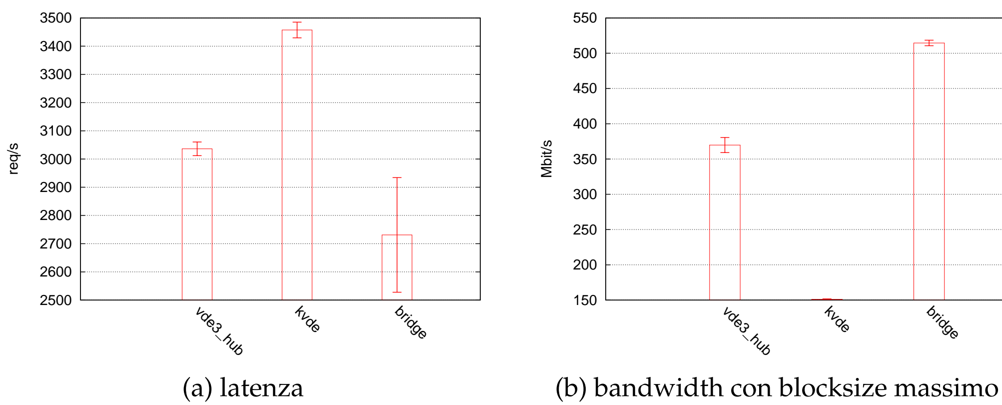


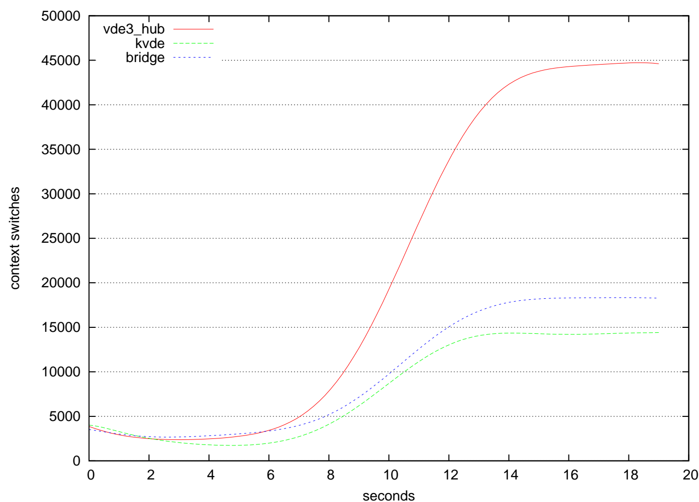
Figura 2.15: Utilizzo di bandwidth di vde3_hub, kvde e bridge



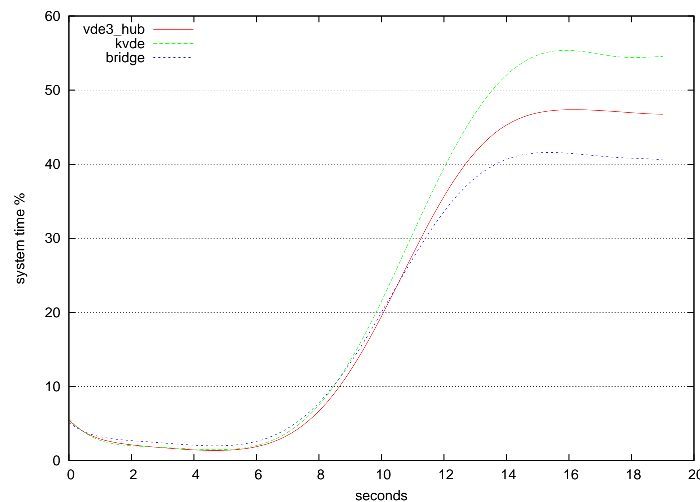
(a) latenza

(b) bandwidth con blocksize massimo

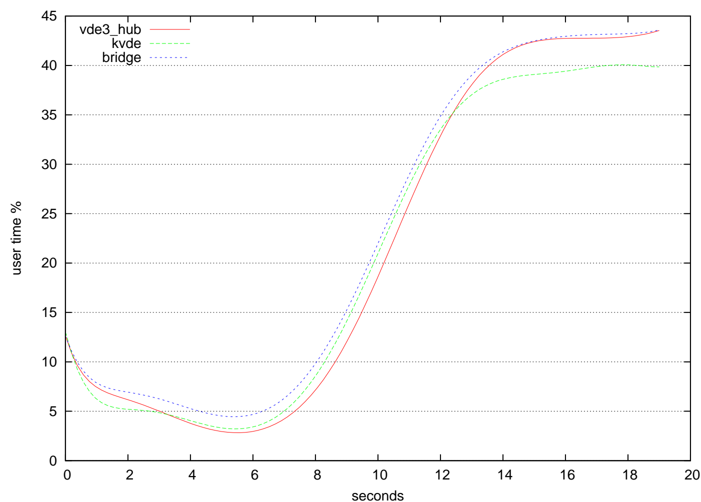
Figura 2.16: Bandwidth e latenza di vde3_hub, kvde e bridge



(a) numero di context switch



(b) tempo di CPU sistema



(c) tempo di CPU utente

Figura 2.17: Utilizzo delle risorse di vde3_hub, kvde e bridge

kvde - bridge - virtio

Compariamo infine tre soluzioni residenti kernel-based tenendo presente che kvde e bridge sono sistemi generici, mentre virtio è una soluzione ad-hoc per le macchine virtuali.

Virtio, in quando soluzione specializzata, riesce ad totalizzare valori molto alti nel *bandwidth* (figura 2.18), superiori a bridge per un fattore di circa 4.3.

Osservando il numero di *richieste al secondo* (figura 2.19a) si nota una valore estremamente basso in virtio. Indagando su questo risultato abbiamo scoperto trattarsi di un problema noto dovuto all'inserimento di un sistema di "TX mitigation" nel backend virtio di KVM per il quale sono già state proposte alcune soluzioni[11].

Analizzando come ultima cosa l'*utilizzo delle risorse* (figura 2.20) si nota che virtio ha un utilizzo molto inferiore della CPU sistema.

[11]Si veda in proposito la soluzione proposta da Michael S. Tsirkin sulla Linux Kernel Mailing List il 17 Agosto 2009 nella discussione "AlacrityVM numbers updated for 31-rc4" (http://lkml.indiana.edu/hypermail/linux/kernel/0908.2/00458.html).

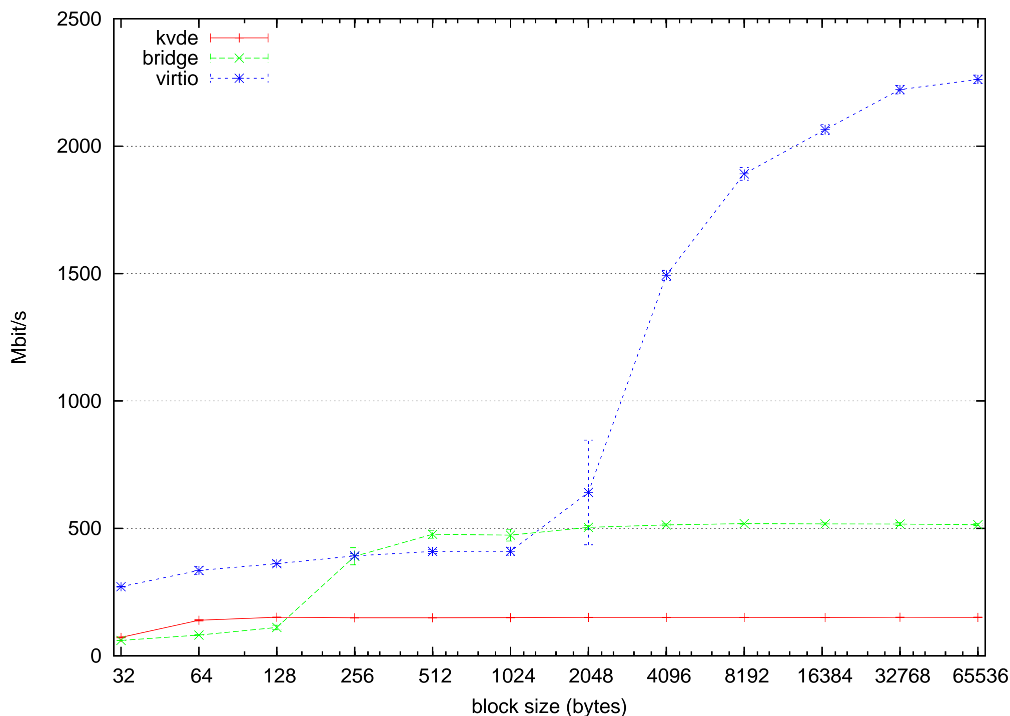
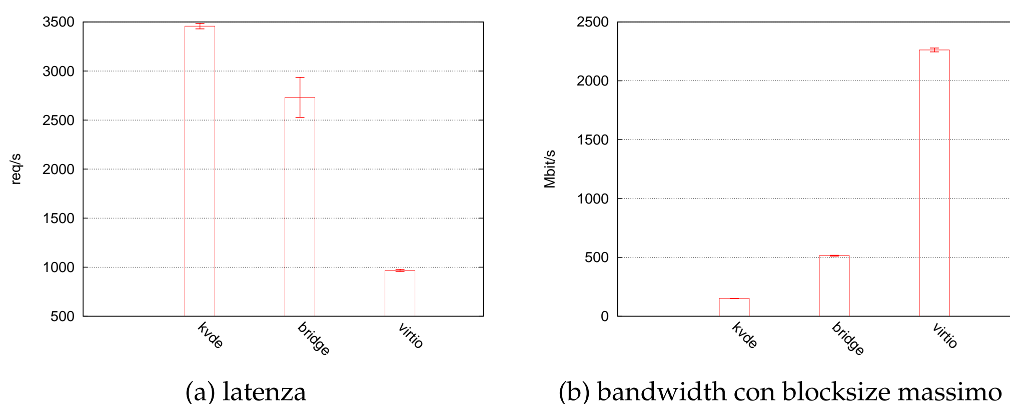


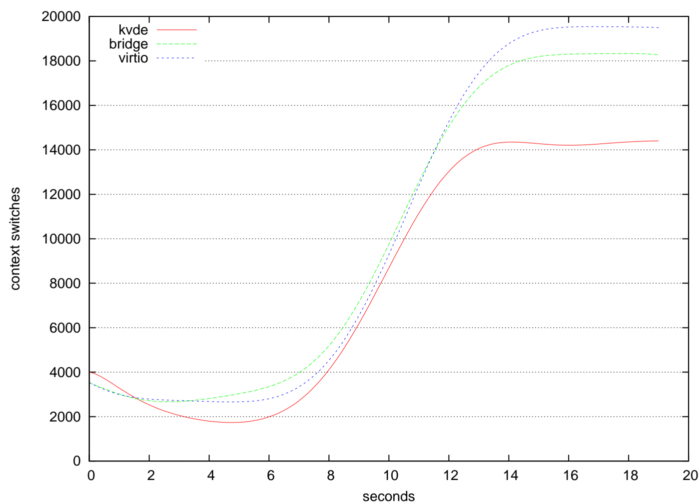
Figura 2.18: Utilizzo di bandwidth di kvde, bridge e virtio



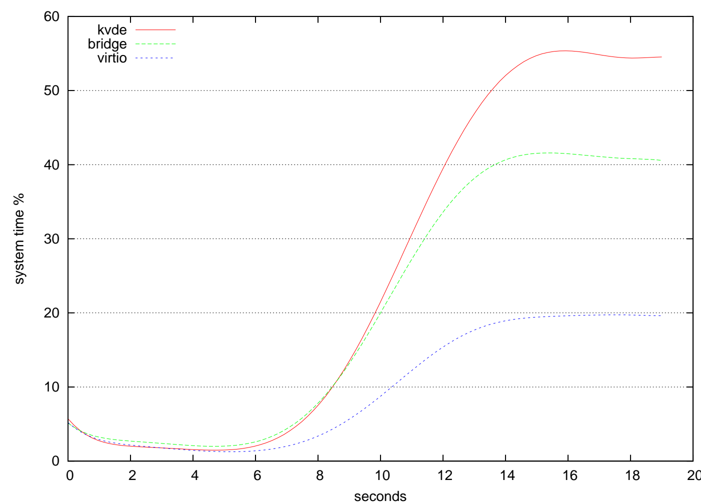
(a) latenza

(b) bandwidth con blocksize massimo

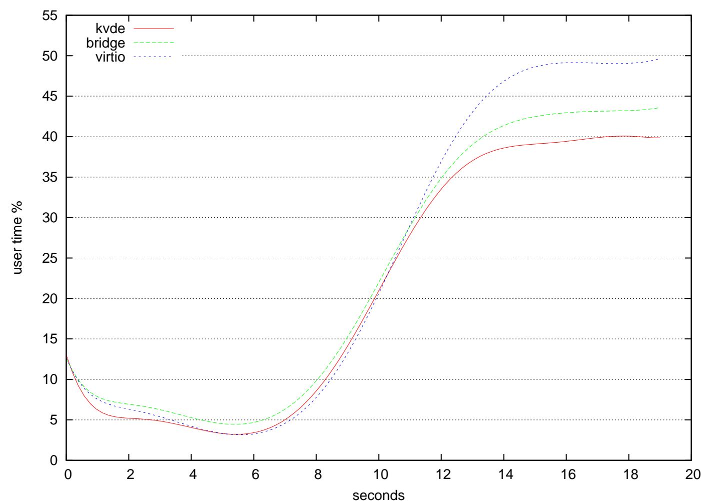
Figura 2.19: Bandwidth e latenza di kvde, bridge e virtio



(a) numero di context switch



(b) tempo di CPU sistema



(c) tempo di CPU utente

Figura 2.20: Utilizzo delle risorse di kvde, bridge e virtio

2.4 Ingegnerizzazione

Qualunque software che non sia triviale richiede l'utilizzo *sistematico* di *processi* per la corretta gestione dello sviluppo e della manutenzione. Nella sezione che segue vengono illustrati i princìpi che abbiamo adottato durante lo sviluppo di VDE 3, molti dei quali sono ripresi anche in [30].

2.4.1 Modello di sviluppo

Nella fase di progettazione abbiamo redatto una *specifica dei requisiti*, ossia una descrizione in linguaggio naturale di come il software avrebbe agito ad alto livello e quali fossero le sue caratteristiche. Un grande aiuto in questo senso ci è stato dato dall'aver stilato dei *casi d'uso*, in altre parole dei casi concreti di come un utente possa interagire con il software.

Una volta esplicitati i requisiti abbiamo ideato un'*architettura* (2.1) e ne abbiamo eseguito una prima *validazione* attraverso l'implementazione di un prototipo in un linguaggio ad alto livello[12]. Questo ci ha permesso dei tempi di sviluppo molto rapidi e refactoring del codice molto semplice, a fronte di performance ridotte.

Attraverso l'esperienza e le correzioni che abbiamo maturato sviluppando il prototipo ci è stato possibile implementare una versione di VDE 3 nel linguaggio C con tutte le caratteristiche desiderate quali performance, tipi espliciti, gestione esplicita della memoria ecc.

Anche la versione in C non si può considerare definitiva senza ulteriori validazioni, dopo un periodo di testing abbiamo condiviso il codice con gli attuali sviluppatori di VDE per raccogliere idee e commenti ed apportare le modifiche necessarie.

[12] python

2.4.2 Protocolli e formati

L'amministrazione remota gioca un ruolo molto importante nell'accettazione e nella facilità d'uso di VDE, è per questo che abbiamo scelto il *protocollo* e il *formato* da utilizzare con particolare attenzione.

I dati in uscita sulla console vengono *serializzati* nel formato scelto poi trasportati fino a destinazione e *deserializzati* dal formato nei tipi di dato nativi del linguaggio che sta leggendo dalla console, è importante quindi che il formato in questione sia *ampiamente diffuso*, supporti *tipi di dato strutturati* e di *facile serializzazione/deserializzazione*[13].

Per il protocollo la scelta è ricaduta naturalmente su quelli di *RPC* data la presenza di comandi e segnali esposti dai componenti illustrati in 2.1.2; anche in questo caso il protocollo deve essere standardizzato ed ampiamente diffuso.

Il formato che più si avvicina a soddisfare tutti i requisiti è JSON[14]: un formato testuale basato su liste ed array associativi con performance ragionevoli e supporto in un'ampia gamma di linguaggi. La grammatica di JSON è studiata per essere *human-readable*, in altre parole la sintassi risulta comprensibile anche *a occhio nudo* senza ulteriori filtri. Per dare un'idea di quanto questo formato sia semplice e conciso viene proposto di seguito qualche esempio delle due strutture dati fondamentali:

liste possono contenere qualunque tipo di dato valido, anche altre liste. La lista è delimitata da parentesi quadre e gli elementi separati da virgole, ad esempio `[1, "due", [42, 3, 14]]`

array associativi anche chiamati *oggetti* sono delle mappe tra una chiave di tipo stringa e un valore di tipo qualunque, particolarmente utili per rappresentare in modo semplice gerarchie, ad esempio `{"uno": 1, "due": 2, {"foo": "bar"}}`.

[13] Facile è sinonimo di efficiente in questo caso.

[14] JavaScript Object Notation

In aggiunta a queste due strutture gli altri tipi di dato contemplati da JSON sono: stringhe (`"foo", "bar"`), numeri (`42, -3.14, 1e6`), booleani (`true, false`) e il valore `null`. La specifica e la grammatica completa sono state presentate come standard nel RFC 4627 [5].

Per quanto riguarda il protocollo di RPC la scelta è ricaduta abbastanza naturalmente su JSON-RPC[15]. Tra le caratteristiche interessanti c'è soprattutto la *semplicità* rispetto ad altri protocolli come XML-RPC, la specifica completamente aperta e l'utilizzo di JSON come formato del protocollo. JSON-RPC è composto da richieste e risposte, o più propriamente da *chiamate a metodi* e *risultati* insieme a *notifiche asincrone* ossia risultati non legati ad alcuna chiamata a metodo.

Durante l'evoluzione del protocollo ne sono state proposte diverse versioni, la prima e più diffusa è la versione 1.0 ed è quella che abbiamo scelto di implementare in VDE 3.

Il flusso normale di messaggi consiste di un oggetto JSON con il nome del metodo da invocare (`method`) e una lista di parametri (`params`). Similmente la risposta è formata da un risultato (`result`) ed un eventuale errore (`error`). Le risposte sono associate alle rispettive chiamate tramite un identificativo intero (`id`), il quale se settato a `null` indica una risposta non associata a nessun metodo ossia una *notifica*.

Illustriamo ora alcune chiamate di JSON-RPC. Le frecce `-->` e `<--` si riferiscono alla direzione del messaggio dal punto di vista del server, rispettivamente in entrata e in uscita. In ordine, negli esempi si trovano una normale chiamata al metodo `echo` e una risposta di successo, una chiamata a un metodo non esistente `invalid` con risposta di errore ed una notifica asincrona del server verso il client all'arrivo di una nuova connessione.

```
--> {"method": "echo", "params": ["Hello"], "id": 1}
<-- {"result": "Hello", "error": null, "id": 1}
```

Chiamata a metodo con successo

[15] http://json-rpc.org

```
--> {"method": "invalid", "params": ["Hello"], "id": 2}
<-- {"result": none, "error": "Unknown method name", "id": 2}
```

Chiamata a metodo con errore

```
<-- {"method": "new_client", "params": ["localhost", "127.0.0.1"],
      id: null}
```

Notifica asincrona

Rispetto ad altri sistemi di RPC quello che abbiamo utilizzato include la specifica del protocollo e non direttamente il supporto in linguaggi di programmazione per la creazione automatica di *stub* ossia funzioni del linguaggio che automaticamente invocano i metodi remoti e restituiscono i risultati. Questo non è stato ritenuto un problema visto il supporto di cui gode il formato JSON tra i linguaggi di programmazione e la semplice implementazione del protocollo. In altre parole non risulta difficile implementare una libreria ad alto livello sia per JSON-RPC sia per l'interazione specifica con VDE 3.

2.4.3 Generazione automatica del codice

I messaggi scambiati sulla console di VDE 3 vengono deserializzati da JSON nel formato interno di VDE 3 ed in seguito passati al componente al quale sono destinati. In particolare per le chiamate a metodo questo implica la *validazione* del numero e dei tipi di parametri passati al metodo per evitare che vengano effettuate chiamate invalide. Questo processo risulta tedioso da implementare manualmente e difficile da mantenere qualora i parametri del metodo cambino, pertanto abbiamo deciso di *autogenerare* la validazione dei parametri ai metodi partendo da una descrizione dei metodi stessi.

All'autore di un componente che voglia esporre un metodo sulla console di management è richiesto di corredare il metodo con delle informazioni quali la descrizione della funzionalità del metodo, il tipo e il nome dei parametri accettati in ingresso e la descrizione del singolo parametro.

```
{
  "fun": "engine_switch_hash_add",
  "name": "hash_add",
  "description": "Add an entry to the hash table",
  "parameters": [
    { "type": "string",
      "name": "mac",
      "description": "The MAC address" },
    { "type": "string",
      "name": "ip",
      "description": "The IP address" }
  ]
}
```

Listato 2.4: Descrizione JSON di un comando esportato

Da questa descrizione vengono autogenerati il controllo dei parametri e la descrizione del metodo come autodocumentazione da esportare sulla console. Il codice autogenerato, una volta validati e deserializzati correttamente tutti i parametri, chiama la funzione vera e propria mentre in caso di validazione errata si occupa di rispondere in modo appropriato; in questo modo è garantito che le funzioni esposte verranno chiamate solo a seguito di messaggi sulla console che rispettano le specifiche.

Più in dettaglio, la descrizione dei metodi è un oggetto JSON con la lista dei metodi esposti, ognuno dei quali contiene: il nome della funzione C da chiamare (`fun`), il nome del metodo esposto (`name`), la sua descrizione (`description`) e la lista dei parametri (`parameters`). Ognuno dei parametri contiene: il tipo (`type`), il nome del parametro (`name`) e la descrizione (`description`).

Riportiamo in listato 2.4 un ipotetico comando di uno switch Ethernet per aggiungere manualmente un'associazione tra indirizzo MAC e IP.

Attualmente le descrizioni dei metodi sono in file `.json` esterni per ragioni di praticità, tuttavia uno sviluppo interessante potrebbe essere quello di includere le descrizioni dei metodi nei commenti della funzione da esportare. Una volta *decorate* le funzioni sarà possibile estrarre da queste la descrizione JSON, così come viene fatto attualmente per la

documentazione dell'API in sezione 2.4.5.

2.4.4 Testing

Durante la progettazione e lo sviluppo di VDE 3 abbiamo dato molta importanza al *testing del codice*. A basso livello (*unit testing*) vengono testate le singole unità come funzioni o moduli mentre ad alto livello (*integration test*) tutto il sistema è testato dal punto di vista dell'utilizzatore per garantire consistenza e integrità, sia nelle fasi di sviluppo attivo che in quelle di manutenzione.

Unit testing

Il testing delle singole unità di codice costituisce una delle tecniche moderne di sviluppo di software, le funzioni vengono testate singolarmente per il comportamento che dovrebbero avere, possibilmente guardando solamente la documentazione della funzione e testando che gli input e gli output rispettino le specifiche (*black box testing*). In altre parole i test che *esercitano* una funzione possono essere considerati un *contratto* che la funzione stessa deve rispettare per essere funzionante.

Un altro approccio al testing delle funzioni è quello di considerare l'implementazione della funzione, in questo caso si parla di *white box testing* e consiste nello stabilire gli input e gli output a partire da come questi vengono utilizzati dalla funzione. Ad esempio è possibile testare tutti i percorsi di errore fornendo opportuni input che li attiveranno ed essere sicuri che siano testati, il cosiddetto *path testing* [28].

I benefici di questo lavoro a volte dispendioso sono molteplici: il codice può essere riscritto con la sicurezza di non aver introdotto nuovi difetti se tutti i test hanno successo, un bug in una funzione può essere isolato e testato attraverso un singolo *test case* ed avere la certezza di non reintrodurlo in versioni successive, i test rappresentano una sorta di documentazione attuale su come il modulo o la funzione dovrebbero essere usati (ossia la

loro API), infine lo unit testing rende anche più facile i test ad alto livello (di integrazione) che compongono le singole unità.

Praticamente per ogni linguaggio di programmazione esistono dei framework di unit testing, per il linguaggio C uno dei più diffusi è check [25] ed è quello che abbiamo utilizzato per VDE 3. L'API di check rispecchia quella del cosiddetto *xUnit* ossia un insieme di framework di testing cross-linguaggio che adottano le convenzioni dell'originale SUnit per il linguaggio Smalltalk [2]. Ogni test è rappresentato da un *test case* e ad ognuno di essi viene associata una *test fixture* ossia delle azioni di `setup`/`teardown` da eseguire prima/dopo il test per verificare pre-condizioni e post-condizioni.

L'unione di tutti i test di unità rappresenta una *test suite* da poter far girare automaticamente e periodicamente per assicurarsi l'assenza di difetti (*continuous testing*). Una delle metriche utilizzate per valutare una test suite è quella del *test coverage* ossia quanta parte del codice esercitano i test; sebbene un coverage vicino al 100% sia auspicabile non è sempre possibile in quanto alcune parti di codice non sono facilmente testabili in isolamento. Per il test coverage abbiamo utilizzato il tool `gcov` messo a disposizione da GCC[16] semplicemente ricompilando il codice con opzioni appropriate, facendo girare la test suite ed analizzando i risultati tramite `lcov`.

Analisi statica e dinamica

Durante il testing del codice ci siamo avvalsi anche di altri strumenti per l'analisi sia dinamica che statica del linguaggio C.

Il framework free software più diffuso per l'analisi dinamica è valgrind [29] che include al suo interno parecchi tool quali un *memory checker* per monitorare gli accessi in memoria[17] o un *heap profiler* per controllare le allocazioni fatte dal programma. Abbiamo integrato valgrind anche all'interno della test suite: in questo modo è possibile controllare che durante i test

[16] `http://gcc.gnu.org`

[17] Compresi i leak.

non si verifichino accessi invalidi o leak diminuendo ancora la possibilità di introdurre regressioni nel codice.

Un altro aspetto che abbiamo considerato è quello dell'analisi statica del codice, questa volta con un tool ben più giovane ma non per questo meno promettente di valgrind: l'analizzatore statico di codice per clang[18]. clang è un frontend C facente parte di LLVM[19]: un framework per la creazione di ottimizzazioni e compilatori. Tra i test effettuati da questo analizzatore statico ci sono: variabili assegnate ma mai lette, dereferenziazione di puntatori nulli e rilevazione di *codice morto*. Ai controlli statici di clang vengono anche affiancati quelli statici di gcc in fase di compilazione attraverso le opzioni per i *warning* quali `-Wall` e `-Wextra`.

Regression tests

Per garantire un corretto funzionamento durante le fasi di sviluppo e mantenimento del codice è stato considerato anche il *system testing* ossia dei test ad alto livello che mirano a validare gli aspetti di VDE 3 come sistema unico.

Tra gli aspetti più importanti del system testing c'è il *regression testing* in altre parole il controllare di non avere introdotto regressioni durante lo sviluppo rispetto alla versione precedente. Fa parte del test di regressione ad esempio l'abilità di riuscire a compilare il codice su più piattaforme e configurazioni senza errori, questo per garantire la riproducibilità della compilazione non solo nell'ambiente nel quale il software è stato sviluppato.

Uno dei tool più diffusi per automatizzare il processo di regression testing è buildbot[20] ed è quello che abbiamo pensato per VDE 3. In particolare è possibile ricompilare il codice e soprattutto far girare la test suite in molteplici ambienti sia in seguito a una singola modifica da parte di uno

[18] http://clang-analyzer.llvm.org
[19] http://llvm.org
[20] http://buildbot.net

sviluppatore sia periodicamente ad esempio ogni notte. Consideriamo il building automatico uno strumento indispensabile per scovare errori in fretta ma soprattutto *automaticamente*, è altrettanto indispensabile avere delle notifiche sullo stato del build ad esempio attraverso e-mail o chat istantanea (IRC) e ridurre al minimo il numero di falsi positivi.

Un altro aspetto del regression testing che abbiamo considerato è quello dell'*integration testing* ossia utilizzare VDE 3 dal punto di vista di un utente facendo una serie di operazioni e controllando che queste vadano a buon fine, ancora una volta la parola chiave è l'automatizzazione del testing: si possono svolgere molteplici test di funzionalità anche complesse in modo *unattended*[21] e registrando i risultati.

Un esempio particolarmente interessante di integration test si ha in presenza di un *tutorial* per utenti alle prime armi: i comandi del tutorial vengono eseguiti periodicamente e si confronta l'output con quello atteso, in questo modo è garantito che il tutorial stia funzionando come ci si aspetta. Un altro esempio è il feedback da parte degli utenti nella riproducibilità di un bug in VDE 3: una volta che il bug è stato *isolato* lo si può aggiungere agli integration tests e verificare che sia stato effettivamente sistemato anche in versioni future. Infine, si può pensare a test automatici per apportare modifiche casuali all'input del programma e verificare che non ci siano comportamenti anomali o addirittura crash (*fuzzying*), questo aspetto diventa di particolare importanza quando il programma manipola dati in diretto controllo di utenti anche remoti come nel caso di VDE 3 per anticipare problemi di sicurezza.

In generale l'idea del regression testing è quello di tracciare l'evoluzione di una *metrica* che caratterizza il software durante lo sviluppo dello stesso; i campi di applicazione da investigare sono sicuramente quelli del tracciamento delle performance facendo dei *micro-benchmark* piuttosto che il *profiling* per identificare i punti di maggiore criticità a seconda del tipo di input piuttosto che la percentuale di codice coperto dalla test suite.

[21] Ossia senza intervento umano.

2.4.5 Documentazione

Per facilitare lo sviluppo e la manutenzione di VDE 3 la documentazione gioca un ruolo molto importante, abbiamo deciso quindi di arricchire il codice di commenti significativi e di *decorare* le funzioni ed i file con una sintassi particolare che permetta di estrapolare automaticamente la documentazione dell'API.

Uno tra i tool più diffusi nel mondo del free software è certamente doxygen[22] ed è quello che abbiamo utilizzato per VDE 3. L'uso base di doxygen è piuttosto semplice e si rifà a strumenti quali JavaDoc: prima di ogni funzione viene introdotto un commento (`/**`) che segnala l'inizio della documentazione per la funzione. All'interno del commento vengono utilizzate delle parole chiave per marcare una breve descrizione della funzione, la descrizione di tutti i suoi argomenti ed il valore di ritorno se presente. I file sorgente vengono in seguito processati con doxygen e viene generata la documentazione da pubblicare in svariati formati come HTML, LaTeX ed RTF. Inoltre, durante la generazione viene controllata l'integrità della documentazione rispetto alla funzione: vengono ad esempio segnalati parametri mancanti per evitare di generare documentazione obsoleta.

L'esempio in listato 2.5 è il prototipo della funzione `vde_context_new` così come appare nel file header `vde3.h`, in questo modo risulta piuttosto semplice mantenere aggiornata la documentazione di riferimento per l'API. Lo sforzo di documentare le funzioni è ampiamente ripagato dai vantaggi: il codice risulta molto più chiaro durante la lettura diretta anche da parte di nuovi sviluppatori. Inoltre il codice diventa *navigabile* ad esempio attraverso le pagine HTML generate da doxygen: è possibile seguire i collegamenti ipertestuali per le funzioni e saltare alla definizione della stessa, di sicuro facilitando la comprensione di come le varie parti del codice vengano utilizzate.

[22] `http://www.doxygen.org`

```
/**
 * @brief Alloc a new VDE 3 context
 * @param ctx reference to new context pointer
 * @return zero on success, -1 on error
 */
int vde_context_new(vde_context **ctx);
```

Listato 2.5: Esempio di documentazione nel codice

La documentazione per gli sviluppatori deve essere affiancata anche da quella per gli utilizzatori finali, specialmente sotto forma di *tutorial* per dimostrare l'uso di VDE 3. In questo caso la scelta è ricaduta su formati di *lightweight markup* di testo come quelli utilizzati per i *wiki* che permettono di leggere e scrivere la documentazione sia nel formato originale[23] che in un formato renderizzato, come ad esempio HTML. Un vantaggio particolarmente interessante, citato in 2.4.4, è quello di poter segnare nella documentazione quali sono le parti di codice che rappresentano degli esempi. In seguito questi esempi vengono automaticamente estratti e compilati, in altre parole è facile garantire che il codice di esempio sia sempre funzionante rispetto agli ultimi sviluppi del codice, certamente un vantaggio per i nuovi utenti.

Il formato di markup che abbiamo scelto è reStructuredText[24], la sintassi è molto intuitiva e non troppo diversa da come si formatterebbe un documento di testo senza rispettare alcuna convenzione. Per riferimento il listato 2.6 riporta un testo decorato con reStructuredText (in breve RST).

2.4.6 Community

Durante lo sviluppo di VDE 3 abbiamo anche pensato a come modificare i processi attuali dal punto di vista della comunità di utenti, sia quella esistente che quella futura. Molti dei temi proposti qui sono anche ripresi in [13].

[23] Il classico file `README`.

[24] http://docutils.sourceforge.net/rst.html

```
Sezione                 * elenco puntato
=======                   - elenco puntato annidato

Sottosezione            1. elenco numerato
------------            2. elenco numerato

*testo corsivo*         ``testo letterale``
**testo grassetto**     http://collegamento.ipertestuale
```

Listato 2.6: Esempio di markup RST

Il passo più importante è sicuramente quello di utilizzare un tool di *version control distribuito* come git[25] che permetta a chiunque voglia farlo di copiare il codice sorgente completo della sua storia e sviluppare le proprie modifiche. In questo modo si passa dal modello tradizionale di un *repository centralizzato* che solo gli sviluppatori autorizzati possono modificare ad uno di *repository peer to peer* dove gli sviluppatori sono liberi di prendere e pubblicare modifiche tra loro. Di fatto si accelera lo sviluppo di nuove feature anche molto complesse ed è più facile attrarre nuovi contributori che vogliano sperimentare il codice.

Il passaggio a un modello di sviluppo distribuito implica molto spesso lo scambio di *patch* via email, solitamente nella mailing list degli sviluppatori del progetto per fare *code reviews*: un contributore propone una modifica al software, suddivisa in piccole patch incrementali e la manda in mailing list affinché ogni altro sviluppatore possa commentarla. Una volta che le modifiche siano ritenute soddisfacenti vengono incluse nel repository *canonico* che rappresenta lo sviluppo corrente del progetto. Per rendere ancora più effettivo il processo di review è fondamentale che tutte le discussioni siano archiviate e pubblicate su web: in questo modo si ha la storia completa di come il progetto si è evoluto dal punto di vista della comunità di sviluppo, mentre la storia del codice è separata e tenuta nel repository citato poco fa.

Avendo a che fare con contributori esterni è molto importante avere documentate delle chiare *linee guida* da seguire prima dell'invio di modifiche.

[25] http://git-scm.com

Ad esempio mantenere un consistente *stile di programmazione* per tutto il progetto risulta molto importante per la facile lettura e manutenzione del codice. Ancora una volta è possibile assicurarsi che queste policy vengano rispettate il più possibile automatizzando i controlli che vengono effettuati su ogni modifica.

2.5 Nuovi componenti

Attraverso la flessibilità che deriva dall'architettura di VDE 3 è possibile implementare facilmente nuovi componenti per aggiungere funzionalità. In questa sezione vengono esplorate alcune di queste applicazioni che riteniamo particolarmente interessanti per dare un'idea delle potenzialità di VDE 3.

2.5.1 Demone di controllo per STP

Lo *Spanning Tree Protocol* [19] viene utilizzato su reti Ethernet per evitare loop all'interno della stessa LAN in presenza di collegamenti ridondati. Tramite lo scambio di messaggi (*BPDU*) viene eletto uno switch *root* e da questo creato un albero di copertura di tutti gli altri switch, le porte che non fanno parte dell'albero vengono disattivate e riattivate solo in caso di necessità di tollerare guasti.

In VDE 2 la logica per gestire STP è stata inclusa direttamente in `vde_switch`, sebbene questo sia pratico poiché non sono richieste ulteriori applicazioni, implica anche che le policy e l'implementazione del protocollo sono incluse nello switch. Un amministratore di rete potrebbe voler separare le *policy* della gestione dello spanning tree oppure provare altre versioni ed implementazioni del protocollo STP senza per questo modificare lo switch, per delegare questo compito in modo semplice è necessario utilizzare un demone che interagisca con lo switch. Lo stesso approccio proposto qui è già stato implementato per il bridge del kernel Linux con un demone che implementa RSTP ed impartisce comandi attraverso *netlink*.

Esternalizzare STP risulta piuttosto semplice in un engine VDE 3 con funzioni di switch Ethernet: il demone deve poter essere in grado di ricevere e spedire i propri BPDU sulla rete oltre che poter controllare lo stato delle porte dello switch. Tutti questi compiti possono essere svolti tramite l'esposizione di comandi e segnali sulla console di management, senza ulteriori metodi di comunicazione tra switch e demone STP.

2.5.2 Anonimato e offuscamento del traffico

La possibilità di creare overlay network anonime consente di soddisfare numerose esigenze quali ad esempio la garanzia di *privacy* per i cittadini, la *sicurezza* per il mondo industriale, la *resistenza all'analisi del traffico* per comunicazioni governative e la *raggiungibilità* di persone in territori in cui vigono restrizioni sulle libertà di comunicazione.

Tramite `libvde` è possibile implementare ed integrare nella rete virtuale molteplici meccanismi che offrono diversi livelli di anonimato e offuscamento. Ad esempio è possibile proteggersi da semplici sistemi di fingerprinting del traffico combinando un contenitore di filtri con un transport che invia dati crittografati e attivando due filtri per inserire ritardo casuale nella trasmissione e aggiungere dati di padding all'interno del payload.

Per soddisfare requisiti di anonimato di più alto livello inoltre è possibile implementare transport o engine specifici che permettano di utilizzare sistemi preesistenti ed ampiamente diffusi come Tor [11], tinc [32] o Sabbia [1].

2.5.3 Logging eventi asincroni

Poter tracciare in un log di sistema i vari eventi asincroni occorsi in un contesto permette di svolgere attività di *auditing* sul nodo VDE, requisito fondamentale per la sicurezza di ogni sistema.

A tal scopo è possibile implementare un semplice engine in grado di registrarsi ai segnali esposti dagli altri componenti e che all'arrivo di una notifica riformatta e redirige a *vde_log()* i dati ricevuti.

2.5.4 Integrazione emulatore Cisco

VDE viene utilizzato anche in contesti didattici per poter ad esempio creare un *laboratorio di rete virtuale* con cui si possono sperimentare in modo simulato vari tipi di apparati di rete.

Un progetto particolarmente interessante è quello del *simulatore di router Cisco*[26]. Anche in questo caso adattare il simulatore a VDE 3 richiede uno sforzo minimo: è possibile includere un engine ed i transport necessari direttamente nel simulatore tramite `libvde`.

Un altro sviluppo interessante potrebbe essere quello di portare il simulatore all'architettura di VDE, di fatto trasformandolo in un engine con appositi transport a seconda dei protocolli supportati.

2.5.5 Integrazione con libvirt

La gestione degli ambienti virtuali e in particolare delle macchine virtuali sta acquistando sempre più importanza per poter consolidare con successo tante macchine logiche dentro una fisica.

libvirt[27] è una libreria in C per rendere semplice la manipolazione e il collegamento di macchine virtuali (domini) sullo stesso host (nodo): attualmente supporta come *backend* alcuni tra i software più noti come XEN, KVM/QEMU, UML e quelli di nuova generazione come i *linux container* (LXC).

Il supporto di rete di libvirt prevede una modalità di NAT nel quale le macchine virtuali sono in rete tra di loro e vengono mascherate con

[26] http://www.ipflow.utc.fr/index.php/Cisco_7200_Simulator
[27] http://libvirt.org

l'indirizzo IP della macchina host qualora vogliano utilizzare la rete fisica di quest'ultima. In aggiunta al NAT esiste anche la possibilità di dare accesso ai guest direttamente alla rete dell'host creando un bridge tra le due interfacce, quella fisica e quella virtuale.

L'integrazione di VDE con libvirt aprirebbe la possibilità di gestire in modo ancora più flessibile il networking tra le macchine virtuali e soprattutto senza privilegi di amministratore sul sistema host. Grazie alla struttura modulare di libvirt è sufficiente implementare un nuovo backend per la rete che gestisca il setup e teardown degli switch VDE: sia attraverso l'invocazione di applicazioni esterne e il management su console come nel caso di VDE 2 sia attraverso l'embedding di VDE all'interno del demone di gestione `virtd` e dunque senza dipendenze esterne se non `libvde`.

2.5.6 Interfacciamento con linguaggi ad alto livello

La creazione di una libreria come `libvde` permette di *esporre le funzionalità* di VDE 3 all'interno di linguaggi di programmazione ad alto livello.

Tramite un tool automatico [28] viene generato un *wrapping* della libreria utilizzando l'API C o C++ del linguaggio desiderato. Una volta compilato il wrapping è possibile chiamare le funzioni di `libvde` dal linguaggio in oggetto *e viceversa*.

L'utilizzo di linguaggi di scripting come python o perl permette di sperimentare in modo semplice e veloce le funzionalità di `libvde`: ad esempio in un contesto didattico, per l'integrazione di VDE all'interno di sistemi già esistenti o per la creazione di prototipi di reti VDE.

[28] SWIG — `http://swig.org`

Conclusioni

Con il nostro lavoro riteniamo di aver valorizzato alcune caratteristiche di VDE quali la facilità e la flessibilità con cui un utente può lavorare e sperimentare su reti virtuali in ambienti eterogenei. Il proliferare delle tecnologie di virtualizzazione delle reti ne testimonia sicuramente l'importanza, ma traccia anche un panorama estremamente frammentato. Con il nuovo VDE speriamo di ottenere un punto di contatto e di interoperazione tra tutti questi sistemi.

È proprio la possibilità di integrare diverse tecnologie che è alla base dell'architettura di VDE 3. L'approccio modulare facilita l'estensione del framework con nuove funzionalità e permette quindi di supportare molteplici sistemi di virtualizzazione.

Tramite l'esperienza maturata durante lo sviluppo di VDE 2 ci è stato possibile individuare i limiti e le necessità del processo di sviluppo. Grazie a ciò siamo riusciti a impiegare nuovi strumenti che rendono più agevole sia la manutenzione del software che lo studio del codice da zero.

Ci auguriamo che l'architettura modulare e il processo di sviluppo documentato e supportato da sistemi di test favoriscano la crescita del progetto, attraendo numerosi contributi esterni ed incrementandone quindi la visibilità.

Sviluppi futuri

Introdurre una nuova architettura in un progetto largamente diffuso come VDE è un processo che richiede tempi molto lunghi. Con questa tesi abbiamo voluto tracciare le basi per un lavoro molto più ampio. Ad oggi abbiamo presentato l'architettura alla comunità di sviluppatori e questa è stata accettata con numerosi commenti e feedback positivi, ci auguriamo quindi di poter entrare presto nella fase di riscrittura del framework e dell'introduzione di nuovi componenti secondo le indicazioni tracciate in 2.2.1 e 2.5.

Restano tuttavia irrisolti alcuni problemi che possono potenzialmente avere un impatto sulle interfacce proposte, ma che abbiamo deciso di non affrontare durante il design e l'implementazione poiché riguardano dei miglioramenti prestazionali non contemplati in VDE 2 e non ancora studiati in modo approfondito ma che non dovrebbero ripercuotersi in modo invasivo sull'architettura.

Come prima cosa è necessario capire se e come introdurre parallelismo tramite *multithreading* all'interno del framework. Attraverso un lavoro di profiling occorre stabilire dove sono ad oggi presenti i *bottleneck*, in seguito bisogna capire se questi bottleneck possono essere risolti tramite multithreading per decidere infine il livello di granularità dei thread rispetto a `libvde`: interni ad un componente, interni al contesto o gestiti dall'applicazione in modo da avere un contesto in ogni thread.

Un'altra possibile ottimizzazione riguarda l'introduzione di sistemi di *caching* interni alla libreria per la memoria utilizzata dai pacchetti in modo da evitare chiamate a sistema per l'allocazione e la deallocazione. Occorre anzitutto studiare in modo esaustivo l'allocatore presente in GNU libc e le alternative, bisogna quindi capire l'impatto del multithreading nell'utilizzo di questi sistemi ed il loro comportamento su piattaforme diverse da GNU/Linux. Infine è necessario il profiling all'interno di diversi scenari per stabilire quale sia il loro impatto in `libvde`.

Sempre relativamente ai pacchetti si può investigare l'introduzione di un *reference counter* per un singolo pacchetto o la modifica della *politica di gestione della memoria* descritta in 2.1.3 in modo che durante lo scambio di un pacchetto il ricevente venga informato dal mittente nel caso in cui debba effettuare o meno la gestione della memoria del pacchetto stesso.

Infine per incrementare la flessibilità del framework vorremmo introdurre, grazie ad una serie di modifiche al contesto, la possibilità di scrivere una semplice applicazione che crei *componenti VDE la cui struttura sia decisa interamente tramite un file di configurazione*. In questo modo si possono istanziare nodi VDE altamente personalizzati senza dover ricorrere ad un linguaggio di programmazione ma semplicemente modificando un file di testo.

Appendice A

Header rilevanti

Listato A.1: vde3.h

```c
#ifndef __VDE3_H__
#define __VDE3_H__

#include <stdarg.h>
#include <syslog.h>
#include <sys/types.h>

#define VDE_EV_READ       0x02
#define VDE_EV_WRITE      0x04
#define VDE_EV_PERSIST    0x10
#define VDE_EV_TIMEOUT    0x01

/**
 * @brief The callback to be called on events.
 *
 * The events argument must contain the event(s) which are available on fd,
 * plus VDE_EV_TIMEOUT if the callback is being called due to a timeout.
 */
typedef void (*event_cb)(int fd, short events, void *arg);

/**
 * @brief This is the event handler the application must supply.
 *
 * It contains several functions pointers for handling events from file
 * descriptors and timeouts.  It is modeled after libevent and thus shares some
 * API details.
 */
typedef struct {
  /**
   * @brief Function to add a new event
```

```
 *
 * This function is called by vde whenever there's interest on events for a
 * file descriptor.
 *
 * @param fd The interested fd
 * @param events The events to monitor for this event
 * @param timeout The timeout, can be NULL
 * @param cb The function to call as a callback for this event
 * @param arg The argument to pass to the callback
 *
 * @return a token representing the event, can be anything
 * application-specific, vde uses this to refer to the event e.g. in calls to
 * event_del, NULL on error.
 *
 * Events is a mask combining one or more of
 *   VDE_EV_READ to monitor read-availability
 *   VDE_EV_WRITE to monitor write-availability
 *   VDE_EV_PERSIST to keep calling the callback even after an event has
 *                  occured
 *
 * If timeout is not NULL and no events occur within timeout then the callback
 * is called, if timeout is NULL then the callback is called only if events of
 * the specified type occur on fd.
 *
 */
void *(*event_add)(int fd, short events, const struct timeval *timeout,
                   event_cb cb, void *arg);

/**
 * @brief Function to delete an event
 *
 * @param ev The event to delete, as returned by event_add
 *
 * This function is called by vde to delete an event for which the callback
 * has not been called yet and/or to explicitly delete events with the
 * VDE_EV_PERSIST flag.
 */
void (*event_del)(void *ev);

/**
 * @brief Function to add a new timeout
 *
 * @param timeout The timeout, cannot be NULL
 * @param events The events for this timeout
 * @param cb The function to call as a callback when the timeout expires
 * @param arg The argument to pass to the callback
 *
 * @return a token representing the timeout, likewise for event this is an
 * opaque object and defined by the application. NULL on error.
 *
```

```
   * This function is called whenever the callback must be called after the time
   * represented by timeout, if VDE_EV_PERSIST is present in events then the
   * callback is called repeatedly every timeout until timeout_del is called.
   *
   */
  void *(*timeout_add)(const struct timeval *timeout, short events, event_cb cb,
                    void *arg);

  /**
   * @brief Function to delete a timeout
   *
   * @param tout The timeout to delete, as returned by timeout_add
   *
   * This function is called by vde to delete a timeout for which the callback
   * has not been called yet and/or to explicitly delete timeouts with the
   * VDE_EV_PERSIST flag.
   */
  void (*timeout_del)(void *tout);
} vde_event_handler;


/**
 * @brief This enum represents the possible component kinds
 */
typedef enum {
  VDE_ENGINE,
  VDE_TRANSPORT,
  VDE_CONNECTION_MANAGER
} vde_component_kind;


/**
 * @brief VDE 3 Component
 */
typedef struct vde_component vde_component;


/**
 * @brief The callback called by connection manager on successful connect
 *
 * @param cm The calling connection manager
 * @param arg The callback private data
 */
typedef void (*vde_connect_success_cb)(vde_component *cm, void *arg);

/**
 * @brief The callback called by connection manager on unsuccessful connect
 *
 * @param cm The calling connection manager
 * @param arg The callback private data
```

```
 */
typedef void (*vde_connect_error_cb)(vde_component *cm, void *arg);

/**
 * @brief Put the underlying transport in listen mode
 *
 * @param cm The connection manager to use
 *
 * @return zero on success, -1 on error (and errno is set appropriately)
 */
int vde_conn_manager_listen(vde_component *cm);

/**
 * @brief Initiate a new connection using the underlying transport
 *
 * @param cm The connection manager for the connection
 * @param local The request for the local system
 * @param remote The request for the remote system
 * @param success_cb A callback to be called on successful connect, can be
 * NULL
 * @param error_cb A callback to be called on error during connect, can be
 * NULL
 * @param arg A pointer to private data, passed to callbacks
 *
 * @return zero on success, -1 on error (and errno is set appropriately)
 */
int vde_conn_manager_connect(vde_component *cm, vde_request *local,
                             vde_request *remote,
                             vde_connect_success_cb success_cb,
                             vde_connect_error_cb error_cb, void *arg);


/**
 * @brief VDE 3 Context
 */
typedef struct vde_context vde_context;

/**
 * @brief Alloc a new VDE 3 context
 *
 * @param ctx reference to new context pointer
 *
 * @return zero on success, -1 on error (and errno is set appropriately)
 */
int vde_context_new(vde_context **ctx);

/**
 * @brief Initialize VDE 3 context
 *
 * @param ctx The context to initialize
```

```
 * @param handler An implementation of vde_event_handler to use
 * @param modules_path A NULL-terminated array of paths to load modules from
 *
 * @return zero on success, -1 on error (and errno is set appropriately)
 */
int vde_context_init(vde_context *ctx, vde_event_handler *handler,
                     char **modules_path);

/**
 * @brief Stop and reset a VDE 3 context
 *
 * @param ctx The context to fini
 */
void vde_context_fini(vde_context *ctx);

/**
 * @brief Deallocate a VDE 3 context
 *
 * @param ctx The context to delete
 */
void vde_context_delete(vde_context *ctx);

/**
 * @brief Alloc a new VDE 3 component
 *
 * @param ctx The context where to allocate this component
 * @param kind The component kind  (transport, engine, ...)
 * @param family The component family (unix, data, ...)
 * @param name The component unique name (NULL for auto generation)
 * @param component The reference to new component pointer
 * @param ... The component-specific arguments
 *
 * @return zero on success, -1 on error (and errno is set appropriately)
 */
int vde_context_new_component(vde_context *ctx, vde_component_kind kind,
                              const char *family, const char *name,
                              vde_component **component, ...);

/**
 * @brief Lookup a component by name
 *
 * @param ctx The context where to lookup
 * @param name The component name
 *
 * @return the component, NULL if not found
 */
vde_component* vde_context_get_component(vde_context *ctx, const char *name);

/**
 * @brief Remove a component from a given context
```

```
 *
 * @param ctx The context where to remove from
 * @param component The component to remove
 *
 * @return zero on success, -1 on error (and errno is set appropriately)
 */
int vde_context_component_del(vde_context *ctx, vde_component *component);

/**
 * @brief Save current configuration in a file
 *
 * @param ctx The context to save the configuration from
 * @param file The file to save the configuration to
 *
 * @return zero on success, -1 on error (and errno is set appropriately)
 */
int vde_context_config_save(vde_context *ctx, const char* file);

/**
 * @brief Load configuration from a file
 *
 * @param ctx The context to load the configuration into
 * @param file The file to read the configuration from
 *
 * @return zero on success, -1 on error (and errno is set appropriately)
 */
int vde_context_config_load(vde_context *ctx, const char* file);


#define VDE3_LOG_ERROR   LOG_ERR
#define VDE3_LOG_WARNING LOG_WARNING
#define VDE3_LOG_NOTICE  LOG_NOTICE
#define VDE3_LOG_INFO    LOG_INFO
#define VDE3_LOG_DEBUG   LOG_DEBUG

/**
 * @brief Log handler function prototype
 *
 * @param priority The message priority: VDE3_LOG_ERROR, ..., VDE3_LOG_DEBUG
 * @param format The format string
 * @param arg Arguments for the format string
 */
typedef void (*vde_log_handler)(int priority, const char *format, va_list arg);

/**
 * @brief Set handler in the logging system
 *
 * @param handler The function responsible to print logs, if NULL stderr will be
 *                used
 */
```

```
void vde_log_set_handler(vde_log_handler handler);

/**
 * @brief Log a message using va_list
 *
 * @param priority Logging priority
 * @param format Message format
 * @param arg Variable arguments list
 */
void vvde_log(int priority, const char *format, va_list arg);

/**
 * @brief Log a message
 *
 * @param priority Logging priority
 * @param format Message format
 */
void vde_log(int priority, const char *format, ...);

#define vde_error(fmt, ...) vde_log(VDE3_LOG_ERROR, fmt, ##__VA_ARGS__)
#define vde_warning(fmt, ...) vde_log(VDE3_LOG_WARNING, fmt, ##__VA_ARGS__)
#define vde_notice(fmt, ...) vde_log(VDE3_LOG_NOTICE, fmt, ##__VA_ARGS__)
#define vde_info(fmt, ...) vde_log(VDE3_LOG_INFO, fmt, ##__VA_ARGS__)
#ifdef VDE3_DEBUG
#define vde_debug(fmt, ...) vde_log(VDE3_LOG_DEBUG, fmt, ##__VA_ARGS__)
#else
#define vde_debug(fmt, ...)
#endif

#endif /* __VDE3_H__ */
```

Listato A.2: module.h

```
#ifndef __VDE3_MODULE_H__
#define __VDE3_MODULE_H__

#include <vde3.h>

#include <vde3/connection.h>

#include <stdlib.h>

/**
 * @brief This symbol will be searched when loading a module
 */
#ifndef VDE_MODULE_START
#define VDE_MODULE_START vde_module_start
#define VDE_MODULE_START_S "vde_module_start"
#endif
```

```
/**
 * @brief Function called on a connection manager to set it in listen mode.
 * Connection managers must implement it.
 *
 * @param cm The Connection Manager
 *
 * @return zero on success, -1 on error (and errno is set appropriately)
 */
typedef int (*cm_listen)(vde_component *cm);

/**
 * @brief Function called on a connection manager to connect it. Connection
 * managers must implement it.
 *
 * @param cm The Connection Manager
 * @param local Connection parameters for local engine
 * @param remote Connection parameters for remote engine
 * @param success_cb The callback to be called upon successfull connect
 * @param error_cb The callback to be called when an error occurs
 * @param arg Callbacks private data
 *
 * @return zero on success, -1 on error (and errno is set appropriately)
 */
typedef int (*cm_connect)(vde_component *cm, vde_request *local,
                          vde_request *remote,
                          vde_connect_success_cb success_cb,
                          vde_connect_error_cb error_cb,
                          void *arg);

/**
 * @brief Function called on an engine to add a new connection. Engines must
 * implement it.
 *
 * @param engine The engine to add the connection to
 * @param conn The connection
 * @param req The connection parameters
 *
 * @return zero on success, -1 on error (and errno is set appropriately)
 */
typedef int (*eng_new_conn)(vde_component *engine, vde_connection *conn,
                            vde_request *req);

/**
 * @brief Function called on a transport to set it in listen mode. Transports
 * must implement it.
 *
 * @param transport The transport
 *
 * @return zero on success, -1 on error (and errno is set appropriately)
 */
```

```c
typedef int (*tr_listen)(vde_component *transport);

/**
 * @brief Function called on a transport to connect it. Transports must
 * implement it.
 *
 * @param transport The transport
 * @param conn The connection to connect
 *
 * @return zero on success, -1 on error (and errno is set appropriately)
 */
typedef int (*tr_connect)(vde_component *transport, vde_connection *conn);



/**
 * @brief Common component operations.
 */
typedef struct {
  int (*init)(vde_component*, va_list);
  //!< called when a context init a new component
  void (*fini)(vde_component*);
  //!< called when a context closes a component
  char* get_configuration; //!< called to get a serializable config
  char* set_configuration; //!< called to set a serializable config
  char* get_policy; //!< called to get a serializable policy
  char* set_policy; //!< called to set a serializable policy
} component_ops;

/**
 * @brief A vde module.
 *
 * It is dynamically loaded and is used to implement a component.
 */
typedef struct {
  vde_component_kind kind;
  char* family;
  component_ops *cops;
  cm_connect cm_connect;
  cm_listen cm_listen;
  eng_new_conn eng_new_conn;
  tr_connect tr_connect;
  tr_listen tr_listen;
  void *dlhandle;
} vde_module;

static inline vde_component_kind vde_module_get_kind(vde_module *module)
{
  return module->kind;
}
```

```
static inline const char *vde_module_get_family(vde_module *module)
{
  return module->family;
}

static inline component_ops *vde_module_get_component_ops(vde_module *module)
{
  return module->cops;
}

static inline cm_connect vde_module_get_cm_connect(vde_module *module)
{
  return module->cm_connect;
}

static inline cm_listen vde_module_get_cm_listen(vde_module *module)
{
  return module->cm_listen;
}

static inline eng_new_conn vde_module_get_eng_new_conn(vde_module *module)
{
  return module->eng_new_conn;
}

static inline tr_connect vde_module_get_tr_connect(vde_module *module)
{
  return module->tr_connect;
}

static inline tr_listen vde_module_get_tr_listen(vde_module *module)
{
  return module->tr_listen;
}

/**
 * @brief Load vde modules found in path
 *
 * @param ctx The context to load modules into
 * @param path The search path of modules, if NULL use default path
 *
 * @return 0 if every valid module has been successfully loaded,
 *         -1 on error (and errno set appropriately)
 */
int vde_modules_load(vde_context *ctx, char **path);

#endif /* __VDE3_MODULE_H__ */
```

Listato A.3: context.h

```
#ifndef __VDE3_CONTEXT_H__
#define __VDE3_CONTEXT_H__

#include <vde3/module.h>

/**
 * @brief A vde context
 */
struct vde_context {
  int initialized;
  vde_event_handler event_handler;
  vde_hash *components;
  vde_list *modules;
};

/**
 * @brief Register a vde_module inside the context
 *
 * @param ctx The context to register the module into
 * @param module The module to register
 *
 * @return zero on success, otherwise an error code
 */
int vde_context_register_module(vde_context *ctx, vde_module *module);

static inline void *vde_context_event_add(vde_context *ctx, int fd,
                                          short events,
                                          const struct timeval *timeout,
                                          event_cb cb, void *arg)
{
  vde_assert(ctx != NULL);
  vde_assert(ctx->initialized == 1);

  return ctx->event_handler.event_add(fd, events, timeout, cb, arg);
}

static inline void vde_context_event_del(vde_context *ctx, void *event)
{
  vde_assert(ctx != NULL);
  vde_assert(ctx->initialized == 1);
  vde_assert(event != NULL);

  ctx->event_handler.event_del(event);
}

static inline void *vde_context_timeout_add(vde_context *ctx, short events,
                                            const struct timeval *timeout,
                                            event_cb cb, void *arg)
```

```
{
  vde_assert(ctx != NULL);
  vde_assert(ctx->initialized == 1);

  return ctx->event_handler.timeout_add(timeout, events, cb, arg);
}

static inline void vde_context_timeout_del(vde_context *ctx, void *timeout)
{
  vde_assert(ctx != NULL);
  vde_assert(ctx->initialized == 1);
  vde_assert(timeout != NULL);

  ctx->event_handler.timeout_del(timeout);
}

#endif /* __VDE3_CONTEXT_H__ */
```

Listato A.4: component.h

```
#ifndef __VDE3_COMPONENT_H__
#define __VDE3_COMPONENT_H__

#include <vde3/command.h>
#include <vde3/common.h>
#include <vde3/connection.h>
#include <vde3/module.h>
#include <vde3/signal.h>

/**
 * @brief Alloc a new VDE 3 component
 *
 * @param component reference to new component pointer
 *
 * @return zero on success, -1 on error (and errno is set appropriately)
 */
int vde_component_new(vde_component **component);

/**
 * @brief Init a VDE 3 component
 *
 * @param component The component to init
 * @param qname The quark representing component name
 * @param module The module which will implement the component functionalities
 * @param ctx The context in which the component will run
 * @param args Parameters for initialization of module properties
 *
 * @return zero on success, -1 on error (and errno is set appropriately)
 */
int vde_component_init(vde_component *component, vde_quark qname,
```

```
                        vde_module *module, vde_context *ctx, va_list args);

/**
 * @brief Stop and reset a VDE 3 component
 *
 * @param component The component to fini
 */
void vde_component_fini(vde_component *component);

/**
 * @brief Deallocate a VDE 3 component
 *
 * @param component The component to delete
 */
void vde_component_delete(vde_component *component);

/**
 * @brief Increase reference counter
 *
 * @param component The component
 * @param count The pointer where to store reference counter value (might be
 * NULL)
 */
void vde_component_get(vde_component *component, int *count);

/**
 * @brief Decrease reference counter
 *
 * @param component The component
 * @param count The pointer where to store reference counter value (might be
 * NULL)
 */
void vde_component_put(vde_component *component, int *count);

/**
 * @brief Decrease reference counter if there's only one reference
 *
 * @param component The component
 * @param count The pointer where to store reference counter value (might be
 * NULL)
 *
 * @return zero if there was only one reference, 1 otherwise
 */
int vde_component_put_if_last(vde_component *component, int *count);

/**
 * @brief Get component private data
 *
 * @param component The component to get private data from
 *
```

```
 * @return The private data
 */
void *vde_component_get_priv(vde_component *component);

/**
 * @brief Set component private data
 *
 * @param component The component to set private data to
 * @param priv The private data
 */
void vde_component_set_priv(vde_component *component, void *priv);

/**
 * @brief Retrieve the context of a component
 *
 * @param component The component to get context from
 *
 * @return The context
 */
vde_context *vde_component_get_context(vde_component *component);

/**
 * @brief Retrieve the component kind
 *
 * @param component The component to get the kind from
 *
 * @return The component kind
 */
vde_component_kind vde_component_get_kind(vde_component *component);

/**
 * @brief Return component name
 *
 * @param component The component
 *
 * @return The quark of component name
 */
vde_quark vde_component_get_qname(vde_component *component);

/**
 * @brief Return component name
 *
 * @param component The component
 *
 * @return The string with component name
 */
const char *vde_component_get_name(vde_component *component);

/**
 * @brief vde_component utility to register commands
```

```
 *
 * @param component The component to add commands to
 * @param commands The NULL-terminated array of commands
 *
 * @return zero on success, -1 on error (and errno is set appropriately)
 */
int vde_component_commands_register(vde_component *component,
                                    vde_command *commands);

/**
 * @brief vde_component utility to deregister commands
 *
 * @param component The component to remove commands from
 * @param commands The NULL-terminated array of commands
 *
 * @return zero on success, -1 on error (and errno is set appropriately)
 */
int vde_component_commands_deregister(vde_component *component,
                                      vde_command *commands);

/**
 * @brief vde_component utility to add a command
 *
 * @param component The component to add the command to
 * @param command The command to add
 *
 * @return zero on success, -1 on error (and errno is set appropriately)
 */
int vde_component_command_add(vde_component *component,
                              vde_command *command);

/**
 * @brief vde_component utility to remove a command
 *
 * @param component The component to remove the command from
 * @param command The command to remove
 *
 * @return zero on success, -1 on error (and errno is set appropriately)
 */
int vde_component_command_del(vde_component *component,
                              vde_command *command);

/**
 * @brief Lookup for a command in a component
 *
 * @param component The component to look into
 * @param name The name of the command
 *
 * @return a vde command, NULL if not found
 */
```

```
vde_command *vde_component_command_get(vde_component *component,
                                       const char *name);

/**
 * @brief List all commands of a component
 *
 * @param component The component
 *
 * @return A null terminated array of commands
 */
vde_command **vde_component_commands_list(vde_component *component);

/**
 * @brief vde_component utility to register signals
 *
 * @param component The component to add signals to
 * @param signals The NULL-terminated array of signals
 *
 * @return zero on success, -1 on error (and errno is set appropriately)
 */
int vde_component_signals_register(vde_component *component,
                                   vde_signal *signals);

/**
 * @brief vde_component utility to deregister signals
 *
 * @param component The component to remove signals from
 * @param signals The NULL-terminated array of signals
 *
 * @return zero on success, -1 on error (and errno is set appropriately)
 */
int vde_component_signals_deregister(vde_component *component,
                                     vde_signal *signals);

/**
 * @brief vde_component utility to add a signal
 *
 * @param component The component to add the signal to
 * @param signal The signal to add
 *
 * @return zero on success, -1 on error (and errno is set appropriately)
 */
int vde_component_signal_add(vde_component *component,
                             vde_signal *signal);

/**
 * @brief vde_component utility to remove a signal
 *
 * @param component The component to remove the signal from
 * @param signal The signal to remove
```

```
 *
 * @return zero on success, -1 on error (and errno is set appropriately)
 */
int vde_component_signal_del(vde_component *component,
                             vde_signal *signal);

/**
 * @brief Lookup for a signal in a component
 *
 * @param component The component to look into
 * @param name The name of the signal
 *
 * @return a vde signal, NULL if not found
 */
vde_signal *vde_component_signal_get(vde_component *component,
                                     const char *name);

/**
 * @brief List all signals of a component
 *
 * @param component The component
 *
 * @return A null terminated array of signals
 */
vde_signal **vde_component_signals_list(vde_component *component);

/**
 * @brief Attach a callback to a signal
 *
 * @param component The component to start receiving signals from
 * @param signal The signal name
 * @param cb The callback function
 * @param destroy_cb The callback destroy function
 * @param data Callback private data
 *
 * @return zero on success, -1 on error (and errno is set appropriately)
 */
int vde_component_signal_attach(vde_component *component, const char *signal,
                                vde_signal_cb cb,
                                vde_signal_destroy_cb destroy_cb,
                                void *data);

/**
 * @brief Detach a callback from a signal
 *
 * @param component The component to stop receiving signals from
 * @param signal The signal name
 * @param cb The callback function
 * @param destroy_cb The callback destroy function
 * @param data Callback private data
```

```
 *
 * @return zero on success, -1 on error (and errno is set appropriately)
 */
int vde_component_signal_detach(vde_component *component, const char *signal,
                                vde_signal_cb cb,
                                vde_signal_destroy_cb destroy_cb,
                                void *data);

/**
 * @brief Raise a signal from this component
 *
 * @param component The component to raise signal from
 * @param signal The signal name
 * @param info The information attached to the signal
 */
void vde_component_signal_raise(vde_component *component, const char *signal,
                                vde_sobj *info);

#endif /* __VDE3_COMPONENT_H__ */
```

Listato A.5: transport.h

```
#ifndef __VDE3_TRANSPORT_H__
#define __VDE3_TRANSPORT_H__

#include <vde3/component.h>

/**
 * @brief Prototype of the callback called by the transport to notify a new
 * connection has been accepted.
 *
 * @param conn The new connection
 * @param arg Callback private data
 */
typedef void (*cm_accept_cb)(vde_connection *conn, void *arg);

/**
 * @brief Prototype of the callback called by the transport to notify the
 * success of a previous connect request.
 *
 * @param conn The connection successfully connected
 * @param arg Callback private data
 */
typedef void (*cm_connect_cb)(vde_connection *conn, void *arg);

/**
 * @brief Prototype of the callback called by the transport to notify an error
 * has occurred during listen or connect.
 *
 * @param conn If not NULL it represents the connection which has caused the
```

```
 * error
 * @param arg Callback private data
 */
typedef void (*cm_error_cb)(vde_connection *conn, int tr_errno,
                            void *arg);

/**
 * @brief Put the transport in listen state
 *
 * @param transport The transport
 *
 * @return zero on success, -1 on error (and errno is set appropriately)
 */
int vde_transport_listen(vde_component *transport);

/**
 * @brief Connect using the given transport
 *
 * @param transport The transport
 * @param conn The connection to use
 *
 * @return zero on success, -1 on error (and errno is set appropriately)
 */
int vde_transport_connect(vde_component *transport, vde_connection *conn);

/**
 * @brief Fill the connection manager callbacks of a transport
 *
 * @param transport The transport to set callbacks to
 * @param connect_cb Connect callback
 * @param accept_cb Accept callback
 * @param error_cb Error callback
 * @param arg Callbacks private data
 */
void vde_transport_set_cm_callbacks(vde_component *transport,
                                    cm_connect_cb connect_cb,
                                    cm_accept_cb accept_cb,
                                    cm_error_cb error_cb, void *arg);

/**
 * @brief Function called by transport implementation to tell the attached
 * connection manager its previous connect() succeeded.
 *
 * @param transport The calling transport
 * @param conn The new connection
 */
void vde_transport_call_cm_connect_cb(vde_component *transport,
                                      vde_connection *conn);

/**
```

```
 * @brief Function called by transport implementation to tell the attached
 * connection manager a new connection has been accepted.
 *
 * @param transport The calling transport
 * @param conn The new connection
 */
void vde_transport_call_cm_accept_cb(vde_component *transport,
                                     vde_connection *conn);

/**
 * @brief Function called by transport implementation to tell the attached
 * connection manager an error occurred in the transport.
 *
 * @param transport The calling transport
 * @param conn The connection which was being created when error occurred (can
 * be NULL)
 * @param tr_errno The errno value
 */
void vde_transport_call_cm_error_cb(vde_component *transport,
                                    vde_connection *conn, int tr_errno);

#endif /* __VDE3_TRANSPORT_H__ */
```

Listato A.6: engine.h

```
#ifndef __VDE3_ENGINE_H__
#define __VDE3_ENGINE_H__

#include <vde3/component.h>

/**
 * @brief Attach a connection to an engine
 *
 * @param engine The engine to attach the connection to
 * @param conn The connection to attach
 * @param req The request to be passed
 *
 * @return zero on success, -1 on error (and errno is set appropriately)
 */
int vde_engine_new_connection(vde_component *engine, vde_connection *conn,
                              vde_request *req);

#endif /* __VDE3_ENGINE_H__ */
```

Listato A.7: connection.h

```
#ifndef __VDE3_CONNECTION_H__
#define __VDE3_CONNECTION_H__

#include <sys/time.h>
```

```c
#include <limits.h>

#include <vde3/attributes.h>
#include <vde3/packet.h>
#include <vde3/common.h>


/**
 * @brief An error code passed to conn_error_cb
 */
typedef enum {
  CONN_OK, //!< Connection OK
  CONN_READ_CLOSED, //!< fatal error occurred during read
  CONN_READ_DELAY, //!< non-fatal error occurred during read
  CONN_WRITE_CLOSED,  //!< fatal error occurred during write
  CONN_WRITE_DELAY, //!< non-fatal error occurred during write
} vde_conn_error;

/**
 * @brief A VDE 3 connection
 */
typedef struct vde_connection vde_connection;

/*
 * Functions to be implemented by a connection backend/transport
 *
 */

/**
 * @brief Backend implementation for writing a packet
 *
 * @param conn The connection to send the packet to
 * @param pkt The packet to send, if the backend doesn't send the packet
 * immediately it must reserve its own copy of the packet.
 *
 * @return zero on success, an error code otherwise
 */
typedef int (*conn_be_write)(vde_connection *conn, vde_pkt *pkt);

/**
 * @brief Backend implementation for closing a connection, when called the
 * backend must free all its resources for this connection.
 *
 * @param The connection to close
 */
typedef void (*conn_be_close)(vde_connection *conn);


/*
 * Functions set by a component which uses the connection.
```

```
 *
 * These functions return 0 on success, -1 on error (and errno is set
 * appropriately). The following errno values have special meanings:
 * - EAGAIN: requeue the packet
 * - EPIPE: close the connection
 *
 */

/**
 * @brief Callback called when a connection has a packet ready to serve, after
 * this callback returns the pkt will be free()d
 *
 * @param conn The connection with the packet ready
 * @param pkt The new packet
 * @param arg The argument which has previously been set by connection user
 *
 * @return zero on success, -1 on error (and errno is set appropriately)
 */
typedef int (*conn_read_cb)(vde_connection *conn, vde_pkt *pkt, void *arg);

/**
 * @brief (Optional) Callback called when a packet has been sent by the
 * connection, after this callback returns the pkt will be free()d.
 *
 * @param conn The connection which has sent the packet
 * @param pkt The sent packet
 * @param arg The argument which has previously been set by connection user
 *
 * @return zero on success, -1 on error (and errno is set appropriately)
 */
typedef int (*conn_write_cb)(vde_connection *conn, vde_pkt *pkt, void *arg);

/**
 * @brief Callback called when an error occur.
 *
 * @param conn The connection generating the error
 * @param pkt The packet which was being sent when the error occurred, can be
 * NULL.
 * @param err The error type
 * @param arg The argument which has previously been set by connection user
 *
 * @return zero on success, -1 on error (and errno is set appropriately)
 */
typedef int (*conn_error_cb)(vde_connection *conn, vde_pkt *pkt,
                             vde_conn_error err, void *arg);


/**
 * @brief A vde connection.
 */
```

```
struct vde_connection {
  vde_attributes *attributes;
  vde_context *context;
  unsigned int max_pload;
  unsigned int pkt_head_sz;
  unsigned int pkt_tail_sz;
  unsigned int send_maxtries;
  struct timeval send_maxtimeout;
  conn_be_write be_write;
  conn_be_close be_close;
  void *be_priv;
  conn_read_cb read_cb;
  conn_write_cb write_cb;
  conn_error_cb error_cb;
  void *cb_priv;
};


/**
 * @brief Alloc a new VDE 3 connection
 *
 * @param conn reference to new connection pointer
 *
 * @return zero on success, -1 on error (and errno is set appropriately)
 */
int vde_connection_new(vde_connection **conn);

/**
 * @brief Initialize a new connection
 *
 * @param conn The connection to initialize
 * @param ctx The context in which the connection will run
 * @param payload_size The maximum payload size a connection implementation can
 * handle, 0 for unlimited.
 * @param be_write The backend implementation for writing packets
 * @param be_close The backend implementation for closing a connection
 * @param be_priv Backend private data
 *
 * @return zero on success, an error code otherwise
 */
int vde_connection_init(vde_connection *conn, vde_context *ctx,
                        unsigned int payload_size, conn_be_write be_write,
                        conn_be_close be_close, void *be_priv);

/**
 * @brief Finalize a VDE 3 connection
 *
 * @param conn The connection to finalize
 */
void vde_connection_fini(vde_connection *conn);
```

```
/**
 * @brief Deallocate a VDE 3 connection
 *
 * @param conn The connection to free
 */
void vde_connection_delete(vde_connection *conn);

/**
 * @brief Function used by connection user to send a packet
 *
 * @param conn The connection to send the packet into
 * @param pkt The packet to send
 *
 * @return zero on success, an error code otherwise
 */
static inline int vde_connection_write(vde_connection *conn, vde_pkt *pkt)
{
  vde_assert(conn != NULL);

  return conn->be_write(conn, pkt);
}

/**
 * @brief Function called by connection backend to tell the connection user a
 * new packet is available.
 *
 * @param conn The connection whom backend has a new packet available
 * @param pkt The new packet. Connection users will duplicate the pkt if they
 * need it after this callback will return, so it can be free()d afterwards
 *
 * @return zero on success, -1 on error (and errno is set appropriately)
 */
static inline int vde_connection_call_read(vde_connection *conn, vde_pkt *pkt)
{
  vde_assert(conn != NULL);
  vde_assert(conn->read_cb != NULL);

  return conn->read_cb(conn, pkt, conn->cb_priv);
}

/**
 * @brief Function called by connection backend to tell the connection user a
 * packet has been successfully sent.
 *
 * @param conn The connection whom backend has a sent the packet
 * @param pkt The sent packet. Connection users will duplicate the pkt if they
 * need it after this callback will return, so it can be free()d afterwards
 *
 * @return zero on success, -1 on error (and errno is set appropriately)
```

```
 */
static inline int vde_connection_call_write(vde_connection *conn, vde_pkt *pkt)
{
  vde_assert(conn != NULL);

  if (conn->write_cb != NULL) {
    return conn->write_cb(conn, pkt, conn->cb_priv);
  }
  return 0;
}

/**
 * @brief Function called by connection backend to tell the connection user an
 * error occurred.
 *
 * @param conn The connection whom backend is having an error
 * @param pkt The packet which was being sent when the error occurred, can be
 * NULL
 *
 * @return zero on success, -1 on error (and errno is set appropriately)
 */
static inline int vde_connection_call_error(vde_connection *conn, vde_pkt *pkt,
                                            vde_conn_error err)
{
  vde_assert(conn != NULL);
  vde_assert(conn->error_cb != NULL);

  return conn->error_cb(conn, pkt, err, conn->cb_priv);
}

/**
 * @brief Set user's callbacks in a connection
 *
 * @param conn The connection to set callbacks to
 * @param read_cb Function called when a new packet is available
 * @param write_cb Function called when a packet has been sent (can be NULL)
 * @param error_cb Function called when an error occurs
 * @param cb_priv User's private data
 */
void vde_connection_set_callbacks(vde_connection *conn,
                                  conn_read_cb read_cb,
                                  conn_write_cb write_cb,
                                  conn_error_cb error_cb,
                                  void *cb_priv);

/**
 * @brief Get connection context
 *
 * @param conn The connection
 *
```

```
 * @return The context in which the connection is running
 */
static inline vde_context *vde_connection_get_context(vde_connection *conn)
{
  vde_assert(conn != NULL);

  return conn->context;
}

/**
 * @brief Get maximum payload size a connection can handle
 *
 * @param conn The connection
 *
 * @return The maximum payload size, if 0 no limit is set.
 */
unsigned int vde_connection_max_payload(vde_connection *conn);

/**
 * @brief Get connection backend private data
 *
 * @param conn The connection to receive private data from
 *
 * @return The private data
 */
static inline void *vde_connection_get_priv(vde_connection *conn)
{
  vde_assert(conn != NULL);

  return conn->be_priv;
}

/**
 * @brief Called by the component using the connection to set some properties
 * about the packet it will receive.
 *
 * @param conn The connection to set properties to
 * @param head_sz The amount of free space reserved before packet payload
 * @param tail_sz The amount of free space reserved after packet payload
 */
void vde_connection_set_pkt_properties(vde_connection *conn,
                                       unsigned int head_sz,
                                       unsigned int tail_sz);

/**
 * @brief Get the amount of free space to be reserved before packet payload
 *
 * @param conn The connection to get headsize from
 *
 * @return The amount of space
```

```
 */
static inline unsigned int vde_connection_get_pkt_headsize(vde_connection *conn)
{
  vde_assert(conn != NULL);

  return conn->pkt_head_sz;
}

/**
 * @brief Get the amount of free space to be reserved after packet payload
 *
 * @param conn The connection to get tailsize from
 *
 * @return The amount of space
 */
static inline
unsigned int vde_connection_get_pkt_tailsize(vde_connection *conn)
{
  vde_assert(conn != NULL);

  return conn->pkt_tail_sz;
}

/**
 * @brief Called by the component using the connection to set some packet
 * sending options. The connection will try to write the packet for AT MOST
 * max_timeout x max_tries time; after that if the send is failed conn_error_cb
 * is called to inform a packet has been dropped.
 *
 * @param conn The connection to set packet sending options to
 * @param max_tries The maximum number of attempts for sending a packet
 * @param max_timeout The maximum amount of time between two packet send
 * attempts
 */
void vde_connection_set_send_properties(vde_connection *conn,
                                        unsigned int max_tries,
                                        struct timeval *max_timeout);

/**
 * @brief Get the maximum number of tries a send should be performed
 *
 * @param conn The connection to get the maximum number of tries from
 *
 * @return The maximum number of tries
 */
static inline
unsigned int vde_connection_get_send_maxtries(vde_connection *conn)
{
  vde_assert(conn != NULL);
```

```
  return conn->send_maxtries;
}

/**
 * @brief Get the maximum timeout between two send attempts
 *
 * @param conn The connection to get the maximum timeout from
 *
 * @return A reference to maximum timeout
 */
static inline
struct timeval *vde_connection_get_send_maxtimeout(vde_connection *conn)
{
  vde_assert(conn != NULL);

  return &(conn->send_maxtimeout);
}

/**
 * @brief Set connection attributes, data will be duplicated
 *
 * @param conn The connection to set attributes to
 * @param attributes The attributes to set
 */
void vde_connection_set_attributes(vde_connection *conn,
                                   vde_attributes *attributes);

/**
 * @brief Get connection attributes
 *
 * @param conn The connection to get attributes from
 *
 * @return The attributes
 */
vde_attributes *vde_connection_get_attributes(vde_connection *conn);

#endif /* __VDE3_CONNECTION_H__ */
```

Appendice B

Ambiente di testing

Listato B.1: Script principale per la gestione dei test

```
#!/bin/bash
# for each test environment do RUNS runs of tests

set +x
set +e

[ -r vm_machines_manager.sh ] && . vm_machines_manager.sh

RUNS=${RUNS:-5}

launch_tests() {
  [ -z "$1" ] && [ -z "$2" ] && [ -z "$3" ] && return

  for i in $(seq 1 $RUNS); do
    $2
    OUTDIR="../results/$1" ./tests_runner.sh
    $3
  done
}

cleanup() {
  modprobe -r bridge || true
  modprobe -r kvde_switch || true
}

cleanup
modprobe bridge
launch_tests bridge start_bridge_env stop_bridge_env

cleanup
```

```
modprobe bridge
launch_tests virtio start_bridge_virtio_env stop_bridge_virtio_env

cleanup
launch_tests vde2_switch start_vde_env stop_vde_env

cleanup
launch_tests vde2_hub start_vde_hub_env stop_vde_hub_env

cleanup
launch_tests vde2_wf start_wf_env stop_wf_env

cleanup
launch_tests vde3_hub start_vde3_hub_env stop_vde3_hub_env

cleanup
launch_tests vde3_hub2hub start_vde3_hub2hub_env stop_vde3_hub2hub_env

cleanup
modprobe kvde_switch
launch_tests kvde start_kvde_notap_env stop_kvde_notap_env
```

Listato B.2: Script per l'esecuzione di un test e la raccolta dei dati

```
#!/bin/bash
# run *_test files in the current directory
# if tests are specified on the commandline only those are run instead of all

# EXAMPLE
# for i in 1 2 3; do
#   OUTDIR=../results/vde2_hub ./tests_runner.sh netpipe_test
#   sleep 10
# done

set -e
set -u
#set -x

TERMINAL="xterm"

[ -z "$OUTDIR" ] && { echo "set OUTDIR"; exit 1; }

SELECTED_TESTS=${@:-}

TAP1=tap1
TAP2=tap2

CLI_ADDR="10.10.20.1"
SRV_ADDR="10.10.20.2"
```

```
DSTAT_OUT="/tmp/dstat_output"
DSTAT_CMD="rm -f $DSTAT_OUT && dstat -M epoch,cpu,sys,proc,net -C total,0,1 \
           -N eth0,$TAP1,$TAP2 --output $DSTAT_OUT"

SSH_CMD="ssh -t -l root"

CLI_OUT="/tmp/client_output"

DSTAT_WAIT=3
SRV_WAIT=5

# Each test defines:
# - TESTNAME
# - CLI_CMD (needs foreground)
# - SRV_CMD (needs foreground)

# use this as TERMINAL to ignore live output
# /!\ relies on the fourth parameter to be the actual command
headless_term() {
  bash -c "$4" 2>&1 >/dev/null
}


[ -d "$OUTDIR" ] || mkdir -p "$OUTDIR"

NOW=$(date +%s)

for file in *_test ; do
  [ -r "$file" -a -x "$file" ] || {
      echo "$file not readable and executable, skipping" >&2
      continue
  }

  if [ ! -z "$SELECTED_TESTS" ]; then
    echo "$SELECTED_TESTS" | grep -q $file || {
      echo "$file not selected on commandline, skipping" >&2
      continue
    }
  fi

  echo "running test $file" >&2

  . "$file"

  $TERMINAL -T host_dstat -e "$DSTAT_CMD" &
  DSTAT_HOST_PID=$!

  $TERMINAL -T client_dstat -e "$SSH_CMD $CLI_ADDR \"$DSTAT_CMD\"" &
  DSTAT_CLI_PID=$!
```

```
  $TERMINAL -T server_dstat -e "$SSH_CMD $SRV_ADDR \"$DSTAT_CMD\"" &
  DSTAT_SRV_PID=$!

  sleep $DSTAT_WAIT # dstat settle

  $TERMINAL -T server -e "$SSH_CMD $SRV_ADDR \"$SRV_CMD\"" &
  SRV_CMD_PID=$!

  sleep $SRV_WAIT # server listening settle

  $TERMINAL -T client -e "$SSH_CMD $CLI_ADDR \"$CLI_CMD\" | tee $CLI_OUT"

  # ....

  kill $SRV_CMD_PID || true
  kill $DSTAT_SRV_PID
  kill $DSTAT_CLI_PID
  kill $DSTAT_HOST_PID

  # gather local/remote results
  cp $DSTAT_OUT $OUTDIR/${TESTNAME}_${NOW}_host_dstat
  scp root@$CLI_ADDR:$DSTAT_OUT $OUTDIR/${TESTNAME}_${NOW}_${CLI_ADDR}_dstat
  scp root@$SRV_ADDR:$DSTAT_OUT $OUTDIR/${TESTNAME}_${NOW}_${SRV_ADDR}_dstat
  cp $CLI_OUT $OUTDIR/${TESTNAME}_${NOW}_${CLI_ADDR}_data
done
```

Listato B.3: Esempio di test

```
TESTNAME="iperf_udp_limited"
SRV_CMD="iperf -u -s -l1472"
CLI_CMD="iperf -u -c 10.10.10.2 -i1 -t10 -fb -yc -l1472 -b70000000"
```

Listato B.4: Estratto dello script di management delle Virtual Machine

```
#!/bin/bash

set -e
set -x

TERMINAL="xterm"

BR_NIC=${BR_NIC:-e1000}
VDE_NIC=${VDE_NIC:-e1000}

# symlink {vmlinuz,initrd.img}-current to the correct files
KERNELVER="current"
```

```
VMDISK_BASE="../vms/"
KERNEL_BASE="${VMDISK_BASE}/kernel"

VM1_NAME="vde-test-1"
VM1_PIDFILE="/tmp/$VM1_NAME.pid"
VM1_MONITOR="/tmp/$VM1_NAME.monitor"
VM1_MAC0="52:54:00:12:34:56"
VM1_MAC1="52:54:00:12:34:57"
VM1_CTRL_IP="10.10.20.1"

VM2_NAME="vde-test-2"
VM2_PIDFILE="/tmp/$VM2_NAME.pid"
VM2_MONITOR="/tmp/$VM2_NAME.monitor"
VM2_MAC0="52:54:00:12:34:58"
VM2_MAC1="52:54:00:12:34:59"
VM2_CTRL_IP="10.10.20.2"

CTRL_TAP=tap1
CTRL_HOSTIP="10.10.20.3"
CTRL_SOCK="/tmp/ctrl.net"
CTRL_MGMT="${CTRL_SOCK}.mgmt"

BENCH_TAP=tap2
BENCH_HOSTIP="10.10.10.3"
BENCH_SOCK="/tmp/bench.net"
BENCH_MGMT="${BENCH_SOCK}.mgmt"

BENCH_BR=br0
BENCH_BR_SCRIPT="${VMDISK_BASE}/vde-test.net0"

WF_MGMT="/tmp/wf.mgmt"
WFSWITCH_SOCK="/tmp/wfswitch.net"
WFSWITCH_MGMT="${WFSWITCH_SOCK}.mgmt"

setup_wf() {
  local PLUG1=$1 ; local PLUG2=$2 ; local MGMT=$3
  if [ ! -z "$4" ] ; then local EXTRA_OPTS=$4 ; fi
  wirefilter -v $PLUG1:$PLUG2 -M $MGMT --daemon $EXTRA_OPTS
}

teardown_wf() {
  local MGMT=$1
  vdecmd -s $MGMT shutdown || true
}

setup_simple_switch() {
  local SOCK=$1 ; local MGMT=$2
  vde_switch -d -s $SOCK -M $MGMT
}
```

```
teardown_simple_switch() {
  local SOCK=$1 ; local MGMT=$2
  vdecmd -s $MGMT shutdown || true
}

setup_switch() {
  local TAP=$1 ; local HOSTIP=$2 ; local SOCK=$3 ; local MGMT=$4
  vde_switch -d -t $TAP -s $SOCK -M $MGMT
  ip l s $TAP up ; ip a a $HOSTIP/24 dev $TAP
}

teardown_switch() {
  local TAP=$1 ; local HOSTIP=$2 ; local SOCK=$3 ; local MGMT=$4
  ip a d $HOSTIP/24 dev $TAP ; ip l s $TAP down
  vdecmd -s $MGMT shutdown || true
}

setup_bridge() {
  local BR=$1 ; local HOSTIP=$2
  brctl addbr $BR ; brctl setfd $BR 0
  ip l s $BR up ; ip a a $HOSTIP/24 dev $BR
}

teardown_bridge() {
  local BR=$1 ; local HOSTIP=$2
  ip a d $HOSTIP/24 dev $BR ; ip l s $BR down
  brctl delbr $BR
}

format_vm_net() {
  local TYPE=$1 ; local VLAN=$2 ; local MAC=$3 ; local MODEL=$4
  local TYPEPARAMS=$5
  echo "-net nic,vlan=$VLAN,macaddr=$MAC,model=$MODEL, \
        -net $TYPE,vlan=$VLAN,$TYPEPARAMS"
}

setup_vm() {
  local VMNAME=$1 ; local VMPIDFILE=$2 ; local VMMONITOR=$3
  local KVM_NET_OPTS=$4 ; local KVM_PROBE_IP=$5
  local VMDISK=$VMDISK_BASE/${VMNAME}.disk
  local SSH_RESULT=255

  KVM="/usr/bin/kvm"

  # split VMs over two cores
  if [ "$(grep -c ^processor /proc/cpuinfo)" == "2" ] && [ -n "$(which taskset)" ]; then
    if [ $VMNAME = $VM1_NAME ]; then
      KVM="taskset -c 0 /usr/bin/kvm"
    fi
```

```
    if [ $VMNAME = $VM2_NAME ]; then
      KVM="taskset -c 1 /usr/bin/kvm"
    fi
  fi

  KVM_COMMON_OPTS="-m 256M -boot c -nographic -serial stdio -pidfile $VMPIDFILE \
                   -monitor unix:$VMMONITOR,server,nowait"
  KVM_KERNEL_OPTS="-kernel $KERNEL_BASE/vmlinuz-$KERNELVER \
                   -initrd $KERNEL_BASE/initrd.img-$KERNELVER \
                   -append \"root=/dev/vda1 ro console=ttyS0,38400\""
  KVM_DISK_OPTS="-drive file=$VMDISK,format=raw,if=virtio,boot=on"

  SSHOPTS="-q -oConnectTimeout=1 -oUserKnownHostsFile=/dev/null \
          -oStrictHostKeyChecking=no"

  $TERMINAL -T $VMNAME -e \
            "$KVM $KVM_COMMON_OPTS $KVM_KERNEL_OPTS $KVM_DISK_OPTS $KVM_NET_OPTS" &

  set +e
  while [ $SSH_RESULT != 0 ]; do
    sleep 1
    ssh $SSHOPTS root@$KVM_PROBE_IP /bin/true
    SSH_RESULT=$?
  done
  set -e
}

teardown_vm() {
  local MONITOR=$1 ; local PIDFILE=$2
  local PID=$(< $PIDFILE) ; local WAITFOR=60
  # try first pushing the power button on the VM
  echo "system_powerdown" | socat STDIO UNIX-CONNECT:$MONITOR || true
  # wait a few seconds and then kill the vm if it did not shutdown
  while [ $WAITFOR -gt 0 ]; do
    sleep 1
    if [ ! -d "/proc/$PID" ]; then break ; fi
    WAITFOR=$(( $WAITFOR - 1 ))
  done
  if [ $WAITFOR -le 0 ]; then
    echo "VM at $MONITOR (pid: $PID) is too slow, killing.."
    kill $PID
  fi
}

start_bridge_virtio_env() {
  setup_switch $CTRL_TAP $CTRL_HOSTIP $CTRL_SOCK $CTRL_MGMT
  setup_bridge $BENCH_BR $BENCH_HOSTIP
  VM1_NET=$(format_vm_net tap 0 $VM1_MAC0 virtio "script=$BENCH_BR_SCRIPT")
  VM1_NET+=" "
  VM1_NET+=$(format_vm_net vde 1 $VM1_MAC1 $VDE_NIC "sock=$CTRL_SOCK")
```

```
  setup_vm $VM1_NAME $VM1_PIDFILE $VM1_MONITOR "$VM1_NET" $VM1_CTRL_IP
  VM2_NET=$(format_vm_net tap 0 $VM2_MAC0 virtio "script=$BENCH_BR_SCRIPT")
  VM2_NET+=" "
  VM2_NET+=$(format_vm_net vde 1 $VM2_MAC1 $VDE_NIC "sock=$CTRL_SOCK")
  setup_vm $VM2_NAME $VM2_PIDFILE $VM2_MONITOR "$VM2_NET" $VM2_CTRL_IP
}

stop_bridge_virtio_env() {
  teardown_vm $VM1_MONITOR $VM1_PIDFILE
  teardown_vm $VM2_MONITOR $VM2_PIDFILE
  teardown_switch $CTRL_TAP $CTRL_HOSTIP $CTRL_SOCK $CTRL_MGMT
  teardown_bridge $BENCH_BR $BENCH_HOSTIP
}

start_vde_env() {
  setup_switch $CTRL_TAP $CTRL_HOSTIP $CTRL_SOCK $CTRL_MGMT
  setup_switch $BENCH_TAP $BENCH_HOSTIP $BENCH_SOCK $BENCH_MGMT
  VM1_NET=$(format_vm_net vde 0 $VM1_MAC0 $VDE_NIC "sock=$BENCH_SOCK")
  VM1_NET+=" "
  VM1_NET+=$(format_vm_net vde 1 $VM1_MAC1 $VDE_NIC "sock=$CTRL_SOCK")
  setup_vm $VM1_NAME $VM1_PIDFILE $VM1_MONITOR "$VM1_NET" $VM1_CTRL_IP
  VM2_NET=$(format_vm_net vde 0 $VM2_MAC0 $VDE_NIC "sock=$BENCH_SOCK")
  VM2_NET+=" "
  VM2_NET+=$(format_vm_net vde 1 $VM2_MAC1 $VDE_NIC "sock=$CTRL_SOCK")
  setup_vm $VM2_NAME $VM2_PIDFILE $VM2_MONITOR "$VM2_NET" $VM2_CTRL_IP
}

stop_vde_env() {
  teardown_vm $VM1_MONITOR $VM1_PIDFILE
  teardown_vm $VM2_MONITOR $VM2_PIDFILE
  teardown_switch $CTRL_TAP $CTRL_HOSTIP $CTRL_SOCK $CTRL_MGMT
  teardown_switch $BENCH_TAP $BENCH_HOSTIP $BENCH_SOCK $BENCH_MGMT
}

start_wf_env() {
  setup_switch $CTRL_TAP $CTRL_HOSTIP $CTRL_SOCK $CTRL_MGMT
  setup_switch $BENCH_TAP $BENCH_HOSTIP $BENCH_SOCK $BENCH_MGMT
  setup_simple_switch $WFSWITCH_SOCK $WFSWITCH_MGMT
  setup_wf $BENCH_SOCK $WFSWITCH_SOCK $WF_MGMT
  VM1_NET=$(format_vm_net vde 0 $VM1_MAC0 $VDE_NIC "sock=$BENCH_SOCK")
  VM1_NET+=" "
  VM1_NET+=$(format_vm_net vde 1 $VM1_MAC1 $VDE_NIC "sock=$CTRL_SOCK")
  setup_vm $VM1_NAME $VM1_PIDFILE $VM1_MONITOR "$VM1_NET" $VM1_CTRL_IP
  VM2_NET=$(format_vm_net vde 0 $VM2_MAC0 $VDE_NIC "sock=$WFSWITCH_SOCK")
  VM2_NET+=" "
  VM2_NET+=$(format_vm_net vde 1 $VM2_MAC1 $VDE_NIC "sock=$CTRL_SOCK")
  setup_vm $VM2_NAME $VM2_PIDFILE $VM2_MONITOR "$VM2_NET" $VM2_CTRL_IP
}

stop_wf_env() {
```

```
  teardown_vm $VM1_MONITOR $VM1_PIDFILE
  teardown_vm $VM2_MONITOR $VM2_PIDFILE
  teardown_wf $WF_MGMT
  teardown_switch $CTRL_TAP $CTRL_HOSTIP $CTRL_SOCK $CTRL_MGMT
  teardown_switch $BENCH_TAP $BENCH_HOSTIP $BENCH_SOCK $BENCH_MGMT
  teardown_simple_switch $WFSWITCH_SOCK $WFSWITCH_MGMT
}
```

Bibliografia

[1] C. Agosti and S. Zanero. Sabbia: a low-latency design for anonymous networks.
http://www.s0ftpj.org/docs/sabbia-pet2005.pdf.

[2] K. Beck. Simple Smalltalk Testing: With Patterns.
http://www.xprogramming.com/testfram.htm.

[3] F. Bellard. QEMU, a fast and portable dynamic translator. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 41–41, 2005.

[4] L. Bigliardi. Supporto alla mobilità e alla autoconfigurazione per il framework Virtual Distributed Ethernet. Mar. 2007.

[5] D. Crockford. RFC 4627: The application/json Media Type for JavaScript Object Notation (JSON), 2006.

[6] R. Davoli. IPN - Inter Process Networking.
http://wiki.virtualsquare.org/index.php/IPN.

[7] R. Davoli. VDE: Virtual distributed ethernet. In *First International Conference on Testbeds and Research Infrastructures for the DEvelopment of NeTworks and COMmunities (TRIDENTCOM)*, pages 213–220, Feb. 1995.

[8] R. Davoli et al. Virtual Square. http://www.virtualsquare.org.

[9] R. Davoli and M. Goldweber. View-OS: Change your View on Virtualization. In *Linux Kongress*, Dresden, Germany, 2009.

[10] J. Dike. User-mode Linux. In *ALS '01: Proceedings of the 5th annual Linux Showcase & Conference*, pages 2–2, 2001.

[11] R. Dingledine, N. Mathewson, and P. Syverson. *Tor: The Second-Generation Onion Router*. http://www.torproject.org/tor-design.pdf.

[12] U. Drepper. How to write shared libraries. 2006. http://people.redhat.com/drepper/dsohowto.pdf.

[13] K. Fogel. *Producing Open Source Software*. O'Reilly Media, Oct 2005.

[14] Free Software Foundation. GNU General Public License. http://www.gnu.org/licenses/.

[15] D. Gasparovski et al. Slirp. http://slirp.sourceforge.net.

[16] F. Giunchedi. Integrazione di SNMP per il remote monitoring di reti virtuali basate su Virtual Distributed Ethernet. Oct. 2007.

[17] M. Goldweber and R. Davoli. VDE: an emulation environment for supporting computer networking courses. In *13th annual conference on Innovation and Technology in Computer Science Education (ITiCSE)*, 2008.

[18] O. Hondaa, H. Ohsakia, M. Imasea, M. Ishizukab, and J. Murayamab. Understanding TCP over TCP: Effects of TCP Tunneling on End-to-End Throughput and Latency. 2005.

[19] IEEE 802.1 working group. IEEE Standards for Local and Metropolitan Area Networks: Media Access Control (MAC) Bridges, 2004.

[20] IEEE 802.1 working group. IEEE Standards for Local and Metropolitan Area Networks: Virtual Bridged Local Area Networks, 2005.

[21] Institute of Electrical and Electronics Engineers. POSIX specification. http://standards.ieee.org/regauth/posix/.

[22] V. Jacobson, C. Leres, S. McCanne, et al. libpcap.

[23] D. Kegel. The ten thousand clients problem. http://www.kegel.com/c10k.html.

[24] J. Loddo and L. Saiu. Marionnet: A virtual network laboratory and simulation tool. In *First International Conference on Simulation Tools and Techniques for Communications, Networks and Systems*, Marseille, France, 2008.

[25] A. Malec, C. Pickett, F. Hugosson, and R. Lemmen. Check: A unit testing framework for C, 2001.

[26] N. Mathewson. *Fast portable non-blocking network programming with Libevent*. 2009. http://www.wangafu.net/~nickm/libevent-book/.

[27] N. Mathewson and N. Provos. libevent — An event notification library.

[28] G. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 2004.

[29] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, 2007.

[30] R. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, Boston, MA, USA, 2005.

[31] B. Schneier. Description of a new variable-length key, 64-bit block cipher. In *Fast Software Encryption, Cambridge Security Workshop Proceedings*, pages 191–204, Dec. 1993.

[32] G. Sliepen and I. Timmermans. tinc Manual.

[33] W. R. Stevens. *UNIX Network Programming: Networking APIs: Sockets and XTI*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[34] D. Thomas and A. Hunte. Mock objects. *IEEE Software*, 19(3):22–24, 2002.

[35] R. Thurlow. RFC 5531: Remote Procedure Call protocol specification version 2, 2009.

[36] O. Titz. Why TCP over TCP is a Bad Idea. 1999. http://sites.inka.de/~W1011/devel/tcp-tcp.html.

[37] F. von Leitner. Source code optimization. In *Linux Kongress*, Dresden, Germany, 2009.

Ringraziamenti

Ringrazio i miei genitori che mi hanno sempre sostenuto.

Vorrei inoltre ringraziare A. Turing e J. von Neumann, senza il loro lavoro tutto questo non sarebbe possibile oggi, e probabilmente io non sarei single.

Ringrazio il mio compare di s?[1]venture Luca che nonostante tutto è riuscito a s[ou]pportarmi.

Ringrazio inoltre i componenti di OSD per i preziosi consigli, in particolare: `alberti`, `dallavia`, `deneb`, `diegobilli`, `gardengl`, `gasparin`, `lacamera`, `lbigliar`, `mondi`, `raggi`, `renzo`, `sback`, `seraghit` e `stanisci`[2]. Senza il loro lavoro questo non sarebbe *davvero* stato possibile.

Ringrazio anche il Dipartimento di Scienze dell'informazione che ha dato la possibilità di costruire una *comunità per nerd* (ab)?usando dell'ex corri(do|od)io Ercolani prima della Grande Diaspora.

Ringrazio Donald E. Knuth per TeX e Leslie Lamport per LaTeX con i quali si ottengono splendidi risultati, dopo tante imprecazioni.

Ringrazio il lettore anonimo[3], che non capendo niente della tesi sta leggendo soltanto questa pagina.

Un ringraziamento particolare va a tutti quelli che ho dimenticato di citare, loro sanno chi sono.

[1] *«Everybody stand back. I know regular expressions.»* http://xkcd.com/208

[2] Sì, è la lista degli iscritti.

[3] Il lettore attento noterà che i ringraziamenti sono simili a quelli della triennale.