Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea in Informatica

Studio e Realizzazione di Strumenti Automatici di Testing per VDE

Relatore:
Renzo Davoli

Presentata da:
Francesco Berni

Sessione I
2014-2015

A tutti coloro che mi hanno sopportato
e supportato in questi mesi.

Introduzione

La virtualizzazione pervade sempre di più il mondo nell'informatica.
L'irruenza con cui servizi che offrono Uptime sempre più elevati sono entrate nella vita di tutti i giorni ha dato una spinta in avanti alla ricerca sulle tecnologie di virtualizzazione inimmaginabile fino a pochi anni prima e, con essa, alle tecnologie di virtualizzazione di rete.
VDE, un progetto che promette l'interoperabilità tra tecnologie di virtualizzazione in modo completamente trasparente è di conseguenza sempre più attuale.

Perchè quindi lo sviluppo è portato avanti da una singola persone con l'assistenza di alcuni contributori storici? perchè così poche patch, modifiche, estensioni o semplici bugfix arrivano dall'infuori di questa cerchia, proprio ora che siti come Github hanno dato una spinta molto forte alla già ricca comunità OpenSoruce?

Questa tesi prova a dare una risposta a queste domande, tramite l'analisi del framework nella sua compelssità, e prova a dare una soluzione tramite lo studio e lo sviluppo di un Framework per creare Test automatizzati pensato per VDE, in modo da fornire un metodo di sviluppo moderno e semplificato.

Il primo capitolo analizza il quadro di riferimento di VDE insieme alle tecnologie che sono state studiate e usate per il design e lo sviluppo di questo Framework. Per prima cosa si analizzano le varie versioni di VDE, passando da VDE2, che è tutt'ora l'unica versione stabile, completa e supportata di

VDE, e per VDE3, forse il proof of concept più interessante dato che abbandona la struttura classica di VDE per analizzarne e metterne alla luce i problemi, arrivando fino alle varie versioni sperimentali che sono tutt'ora in sviluppo.
La seconda sezione del primo Capitolo analizza 2 approcci diametralmente opposti, ma compatibili, per il Testing di rete: Packetdrill e Mininet. L'analissi di questi tool, facilmente considerabili lo stato dell'arte in materia, permette di costruire questo framework tenendo presente cosa esiste già ed è quindi non necessario reimplementare, e, nel contempo, dare un idea di che struttura dare al framework implementato in questa tesi.
Infine, si spendono due parole su Go, il linguaggio usato in questo progetto, data la relativa giovinezza del linguaggio, è sembrato opportuno spendere due parole rigaurdo il perchè è stato scelto, sulle sue feature principali, e sulle librerie usate.

Il secondo Capitolo entra nel dettaglio dell'implementazione, descrivendo prima di tutto il Design del progetto e l'infrastruttura creata per svilupparlo, per poi dilungarsi nella descrizione dell'API proposta e della sua implementazione, infine facendo alcuni esempi di utilizzo di questa API per la costruzione di Test per VDE2.

Il capitolo conclusiovo riassume il lavoro svolto, e propongone una serie di migliorie e futuri sviluppi possibili per questo Framework, sperando di portare avanti il progetto VDE nel futuro.

Indice

Capitolo 1

Quadro di Riferimento

Prima di introdurre il Framework di Testing costruito in questa tesi, è bene parlare prima di cosa vogliamo testare(VDE), del perché, e dello stato dell'arte nell'ambito dei test di rete.

1.1 VDE: Virtual Distribuited Ethernet

Il progetto VDE, parte del progetto VirtualSqure[10], è definito spesso come "*a swiss army tool for emulated networks*[1, 2], per sottolineare le potenzialità del progetto nel creare reti virtuali tra macchine virtuali, fisiche, emulatori etc. tutto in modo completamente trasparente virtualizzando il layer Ethernet.
Tale Flessibilità ha permesso l'uso di VDE per la creazione di svariati applicativi di rete quali VPN[1], Overlay Network, Networking tra macchine virtuali, NAT traversal e molti altri, rendendo VDE uno strumento estremamente potente sia per l'utente che per chi amminsitra grandi reti di macchine fisiche e virtuali.
La versione più in uso al momento è VDE2, ma negli anni svariate versioni sperimentali sono state sviluppate, facendo evolvere il progetto in un vero e

[1] Virtual Private Network

proprio Framework.

1.1.1 VDE 2

Come accennato in precedenza la versione più usata di VDE è VDE2, una collezione di applicazioni interamente sviluppata per sistemi POSIX, sotto licenza GPL[1, 2].
Ogni componente è il più semplice e specializzato possibile, come richiede la filosofia UNIX, e comunica con gli altri tramite IPC[2], ad esempio *pipe* e *socket*.
Col tempo svariati componenti sono stati aggiunti a VDE2, alcuni per aggiungere funzionalità, quali comunicazioni cifrate o disturbate per simulare problemi di rete; altri per permettere l'utilizzo di VDE attraverso tipi di rete diversi da Ethernet, per esempio, il protocollo DNS[3].
Essendo Fulcro di VDE l'interoperaribilità, ogni componente è sempre stato pensato per funzionare insieme agli altri componenti, mantenendo un certo schema, che andremo ad illustrare nella sezione successiva.

Componeti Principali

Ci sono 4 tipi di componenti in VDE2[1]:

- **VDE Switch:** Uno switch, come in una rete fisica, è il componente principale per costruire una rete virtuale usando VDE. Esattamente come uno switch fisico uno switch VDE ha una serie di porte(in questo caso, virtuali) a cui le varie applicazioni (siano esse macchine virtuali, interfacce di rete, altri switch ecc.) possono collegarsi.
- **VDE Plug:** Rappresenta un connettore che viene collegato alle porte di uno switch e si comporta come una *pipe*; i dati che arrivano dalla re-

[2]Inter Process Comunication
[3]Domain Name System

te VDE dentro la plug vengono mandati su Standard Output, mentre quelli che vengono inseriti su Standard Input vengono mandati nella rete. Per collegare 2 plug è stato sviluppato il programma *dpipe*[4] che semplicemente collega Standard Input di un connettore allo standard output dell'altro e vice-versa.

- **VDE Wire:** è semplicemente un qualsiasi applicativo in grado di trasferire uno stream di dati (alcuni comunemente usati sono cat, ssh e netcat).

- **VDE Cable:** il condotto con cui gli switch VDE sono connessi tra di loro, cioè una composizione di 2 Plug e un Wire, esattamente come uno cavo fisico.

Possono esistere svariate implementazioni di ognuno di questi componenti, vde_cryptcab è per esempio un Manager di Cavi che permette due switch VDE di comunicare usando un canale cifrato di tipo blowfish, mentre kvde_switch è uno switch VDE che esiste in kernel space; e così via possono essere implementati componenti che hanno funzioni diverse finchè sono in grado di interoperare tra loro.

Limiti e Problemi

Tre sono i principali problemi architetturali di VDE2, la rigidità del Framework, l'estremo accoppiamento dei componenti e la mancanza di modularità all'interno dei singoli componenti.
La rigidità del framework, rende difficile aggiungere nuove feature dato che molto codice è complesso da riutilizzare per scrivere nuovi componenti, ad esempio, per scrivere una nuova vde_plug è spesso necessario copiare buona parte del codice di vde_plug e modificarlo[2].

[4] bi-directial pipe

L'estremo accoppiamento dei vari componenti, dovuto all'essere costruiti come applicativi separati e non come libreria[2], rendendo difficile l'estensione delle funzionalità, la comprensione e quindi la manutenibilità del codice.
Infine la mancanza di moduli ben definiti, rende difficile testare in isolamento le varie parti del programma, di conseguenza è difficile assicurarsi che l'aggiunta di una nuova feature o la risoluzione di un bug non ne inserisca di nuovi. Questo sembra contraddire quanto detto in precedenza riguardo la modularità di VDE, ma, per quanto vero che il framework sia pensato come una serie di applicativi modulari, i singoli applicativi sono praticamente monolitici.
Da questi tre problemi architetturali derivano una serie di altri problemi: il numero di context switch che un pacchetto deve subire per passare tra tutti i vari applicativi fino ad arrivare a destinazione[2]; la complessita nell'aggiungere funzionalità a vde_switch, e molti altri.

1.1.2 VDE 3

VDE3 nasce pensando a come risolvere i problemi sopra elencati; il suo focus è infatti, prima di tutto la facilità di sviluppo, di estensione e la manutenibilità del codice, in aggiunta alle caratteristiche classiche di VDE, quali le performance e l'interoperabilità con gli standard di rete[2].
Per rispettare questi principi, VDE3 è quindi una libreria ad eventi, che quindi fornisce un API in grado di creare componenti diversi in modo trasparente e interoperabile, per poi combinarli e creare un Nodo VDE(e non più una serie di applicativi).
I tre tipi di componenti di VDE3 sono:

- **transport:** si occupa di creare nuovi canali di comunicazioni tra il nodo VDE3 e il mondo esterno.

- **engine:** processa i dati da e per le varie applicazioni.

- **connection manager:** lega assieme transport(e quindi le varie connesioni create attraverso il transport) e l'engine.

È inoltre possibile configurare i singoli componenti a runtime attraverso comandi, mentre segnali asincroni vengono generati dai componenti per notificare l'avvenimento di un evento al suo interno.
Allo stesso modo si possono costruire interfacce di management semplicemente creando un engine apposito(che manderà poi segnali agli altri componenti all'iterno del Nodo) e un transport per ricevere le connesioni(siano esse http, la classica interfaccia di managment di VDE2 o un interfaccia OpenSwitch[25]).
Fornendo un API è inoltre possibile interfacciare VDE3 con linguaggi di scripting di alto livello, quali Perl o Python.

Problematiche

VDE3 è purtroppo stato abbandonato dagli autori per motivi di tempo, e quindi rimane il proof of concept presentato come tesi specialistica nell'anno 2009. Al momento gli unici moduli implementati sono infatti: un transport per VDE2, due engine, un hub e uno di managment, e un connection manager generico. Nel mentre molto è successo nell'ambiente e il progetto non è più altrettanto attuale, ma sarebbe interessante continuarne lo sviluppo.

1.1.3 Versioni Sperimentali

Dopo la presentazione di VDE3, molti esperimenti sono stati fatti nel mondo di VDE, in parte tentando di risolvere alcuni dei problemi che VDE3 ha portato alla luce, in parte per migliorarne le prestazioni; di seguito accenniamo ad alcuni di essi.

VDE in memoria condivisa

Una versione di VDE tutta in memoria condivisa è stato pensata per limitare il numero di context switch che un pacchetto deve subire durante il

suo ciclo di vita in una rete VDE.

VDE4

Un nuovo tentativo a modularizzare VDE, nello specifico a modularizzare quello che in VDE2 è lo Switch, permettendo di costruire piccoli moduli da combinare in uno Switch, questo rende più semplice l'estensione delle funzionalità di uno switch VDE, oltre a permettere di testatare separatamente i vari moduli migliorando e velocizzando il processo di sviluppo; VDE4 è l'ultimo tentativo in ordine cronologico e non è ancora in fase di Release.

Switchless VDE

Switchless VDE è forse il progetto più sperimentale tra tutti quelli visti fin ora; abbandona l'idea di uno switch, cercando il massimo delle prestazioni possibili, e tenta di lavorare direttamente in memoria, in modo non dissimile da VDE in memoria condivisa, ma portando il concetto all'estremo, limitandosi a reindirizzare pacchetti collegati con VDE plug vivendo in kernel space. Esistendo ancora pochissima documentazione a riguardo, è difficile dare informazioni maggiori, se non confermare la sua esistenza e il suo scopo.

1.2 Testing di reti, stato dell'arte

Il testing di rete è un argomenento complesso, varia molto a seconda di cosa si vuole testare, siano esse Prestazioni e Correttezza di un Protocollo di rete, di un Applicativo di rete o di uno Stack di rete.
Non esiste quindi un singolo Silver Bullet per questo problema, ma esistono svariate soluzioni che lo affrontano da diversi punti di vista. In questo capitolo verranno analizzate due di queste soluzioni OpenSource, per le quali sarebbe interessante fare moduli per VDE.

1.2.1 Packetdrill

Packetdrill è un progetto di Google per testare interi Stack di rete, dalle system call alle interfacce di rete, dal funzionamento dei socket, alla correttezza dei singoli pacchetti, scritto in C ed usato all'interno di Google per lo sviluppo di nuove feature per l'implementazione Linux di TCP.[3]
È quindi un ottimo tool di partenza quando si vuole testare la correttezza e le performance di interi stack di rete, siano essi virtuali o fisici. Essendo scriptabile è possibile personalizzare i test a cui siamo interessati completamente.
Probabilmente, per quanto riguarda questo tipo di test di rete, packetdrill è il leader assoluto, data la sua potenza e flessibilità. Non è tuttavia essente da problemi: il linguaggio di scripting è completamente personalizzato e, nonostante sia molto semplice, va imparato; la rete va impostata a mano, bisogna quindi avere packetdrill e lanciarlo su entrambi gli endpoint; non è possibile fare test diretti sull'applicazione, ma solo in modo trasparente attraverso 2 interfacce di rete.

1.2.2 Mininet

Mininet ha un approccio completamente diverso al testing di rete rispetto a Packetdrill.
Mininet scegle di costruisce una rete virtuale con un kernel e uno switch e caricando in queste VM[5] l'applicazione che si vuole testare, tutto su una singola macchina[5], con un singolo comando[6].
Inoltre, tramite la sua CLI[6] permette di iteragire con tutte le VM in questa rete e l'applicazione caricata: è quindi un ottimo strumento di sviluppo[7], insegnamento[9] e ricerca, per la facilità con cui si possono creare reti complesse in poco tempo, con poco sforzo. Supporta inoltre OpenSwitch[25] e Software-Defined Networking[26].

[5] Virtual Machine
[6] Command Line Interface

1.3 Go

È stato deciso di implementare questo framework in Go, un linguaggio nato all'interno di Google, che ha come principi fondanti l'essere conciso, espressivo, pulito ed efficiente. È un linguaggio Compilato, con tipi Statici, orientato alla programmazione concorrente, il cui sistema di tipi permette la costruzione di programmi modulari e flessibili. È Garbage collected, ma compila estremamente velocemente. È un linguaggio veloce, tipato e compilato che da la sensazione di essere un linguaggio tipato dinamicamente e interpretato durante l'uso[11].
Il Design di Go è molto interessante e molte delle sue caratteristiche peculiari vengono usate i questo progetto, quindi ci dilungheremo nell'analizzarle.

1.3.1 Compilazione

È interessante per prima cosa che il binario compilato abbia all'interno tutte le librerie che usa come dipendenza, non necessitando l'installazione di libreria aggiuntive per il deploy se non le minime necessarie di sistema; questo causa però la creazione di binari molto grandi. Il binario di esempio per questo progetto è infatti di 6.9MB nonostante sia di poche righe di codice. Di segutio inseriamo il risultato del comando ldd su tale file:

```
ldd example/example
linux-vdso.so.1 (0x00007ffd771dd000)
libpcap.so.0.8 => /usr/lib/x86_64-linux-gnu/libpcap.so.0.8 (0x00007f1349304000)
libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007f13490e7000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f1348d3e000)
/lib64/ld-linux-x86-64.so.2 (0x00007f1349546000)
```

Si puo' notare che l'unica libreria non di sistema è libpcap, dato che la libreria Gopacket chiama direttamente la libpcap scritta in C[12], operazione estremamente agevole, sopratutto rispetto ad altri adattamenti come quelli in python.

1.3.2 Sistema di tipi

Il sistema di tipi di Go sembra un sistema ad oggetti classico con alcune fondamentali differenze: non esiste ereditarietà, la composizione di oggetti viene fatta tramite embedding[13] che permette ad un oggetto di "ereditare" tutti i metodi dell'oggetto integrato al suo interno. Esiste un modello di classi astratte, chiamate **interface**[13], una lista di signature di metodi associate ad un nome; tutti gli oggetti che implementano quei metodi, rispettano quell'interfaccia e quindi possono essere usati quando si richiede quell'interfaccia. In questo modo è molto facile costruire un API, basta costruire una serie di metodi che accettano una data interfaccia. Un nuovo oggetto che vuole usare quel metodo dovrà implementare correttamente l'interfaccia in questione, pena errore in fase di compilazione.
Questa soluzione è molto elegante, sopratutto pensando a problemi di ereditarietà multipla tipica della programmazione ad oggetti, essendo la dichiarazione implicita, un singolo oggetto può implementare virutalmente infinite interfacce, potendo creare così oggetti molto versatili: un ottimo esempio di questo pattern è nella libreria net di Go, dove l'oggetto TCPConn, che rappresenta una connesione TCP tramite socket, implementa l'interfaccia Conn, e l'intefaccia di io.ReaderWriter (e di conseguenza l'interfaccia di io.Reader e io.Writer) che permette alcuni degli usi avanzati di una connesione come visto all'interno della Prima Appendice.

1.3.3 Modello Concorrente

Il Modello Concorrente[14] si incentra sulle **goroutines**, un astrazione di un thread gestito dal Runtime di Go, vengono lanciate semplicemente con l'istruzione **go**:

```
go f(x,y,z)
```

la funzione f con argomenti x, y e z viene lanciata in modo concorrente dal resto del codice, e non ne influenza la terminazione, le **goroutines** possono essere processi, thread o nessuno dei precedenti: il Runtime di Go insieme

alle variabli di ambiente del caso, decideranno cosa è meglio usare in quella situazione. Le **goroutines** sono leggerissime e lanciarne dozzine non influisce sulle performance di sistemi molto poco potenti quali possono essere dei processori arm single core.
Per avere un'architettura concorrente sono necessari metodi di comunicazione tra i vari "processi" concorrenti, per questo ci sono i **Canali**, dei condotti tipati dai quali si possono ricevere e inviare valori:

```
ch := make(chan int)
ch <- v //Invia v attraverso il canale ch
v := <- ch //ricevi un valore da ch e assegna il valore a v
```

i Canali sono bloccanti sia in invio che in ricezione, ma possono avere un buffer, in tal caso verrano dichiarati in questo modo:

```
ch := make(chan int, n)
```

e sono bloccanti in invio solo se il buffer è pieno, e in ricezione solo quando il buffer è vuoto. Inoltre nella libreria standard del linguaggio è presente il pacchetto sync che contiene varie utility per la sincronizzazione delle goroutines[1.3.5].

1.3.4 Toolchain

Molto interessante è inoltre la serie di tool a disposizione direttamente all'interno del linguaggio[16]: un sistema di UnitTest, la possibilità di usare diversi compilatori (essendo go una specifica non un implementazione), un packet manager in grado di recuperare i codici sorgenti direttamente da un repository insieme alle dipendenze e compilarli e, ultimo ma non per importanza, un formatter, che si occupa di formattare il codice in modo tale da avere un code style obbligato e meno difficoltà di lettura del codice.
Tutto questo rende Go un linguaggio potente e moderno, facile da leggere e da mantenere, nonostante sia ancora giovane e acerbo in molti aspetti.

1.3.5 Librerie

Go ha un enorme e sempre crescente quantitativo di librerie, e una libreria standard molto fornita; parleremo qui di alcune delle librerie di interesse a questo progetto. Sono state usate estensivamente molte delle librerie della libreria standard di Go, primo tra tutti il pacchetto **net**[17], che fornisce tutte le primitive di rete, per quanto l'uso sia estensivo non c'è molto da dire riguardo questo pacchetto che non sia stato già detto: è una libreria di rete molto standard, con delle astrazioni interessanti e potenti[7]
Secondo pacchetto molto importante della libreria standard è **sync**[18], che offre una serie di utility per gestire problemi di concorrenza quali Mutex, Condizioni, Pool, Locks e WaitGroup, quest'ultimo verra spiegato meglio nel Capitolo 2.3.2. Il pacchetto **io**[19] ha un uso più sottile, ma viene usato per ricevere e inviare dati in modo non standard (come accennato a 2.5), mentre il pacchetto **os**[20] contiene un interfaccia indipendente dalla piattaforma usata per accedere alle funzionalità del sistema operativo, rimandendo il più possibile in uno stile Unix-like.
Per finire accenniamo alle libreria che non fanno parte della standard library usate per questo progetto:

Gopacket

Gopacket è un libreria che permette il decoding di pacchetti di rete, costruendo in modo molto semplici decoder personalizzati per le proprio necessità[21], permette inoltre (tramite sottopacchetti) di catturare pacchetti, tramite libpcap, da un interfaccia di rete in tempo reale e di ricostruire stream tcp tramite il sottopacchetto tcpassembly, una libreria molto semplice da usare per recuperare per recuperare statistiche precise di tempo e dati ricevuti.

[7] il nostro uso verrà spiegato più in dettaglio nel capitolo 2.4

procfs

Procfs è invece una piccola liberira che permette di recuperare dati da /proc per i singoli processi a cui siamo interessati con un interfaccia adatta a Go. Sono state mandate alcune PullRequest a questa libreria (e rapidamente accettate) per aggiungere funzionalità apposistamente per questo progetto. Non venivano recuperate correttamente alcune delle informazioni necessarie, piuttosto che ripartire da zero è stato scelto di inviare patch a questo progetto in modo da risolvere i problemi rendendo procfs una liberia funzionale a questo progetto.

Capitolo 2

Testing VDE

Nel capitolo precedente, analizzando VDE, si è arrivati ad alcune conclusioni importanti: VDE è un tool estremamente potente, con pressochè infinite possibilità, ma con seri problemi di mantenibilità e difficoltà di sviluppo. Lo sviluppo continuo che subisce lo rende estremamente eterogeneo e difficile da testare e debuggare nella sua interezza, spesso nuove versioni vengono sviluppate per migliorarne le prestazioni; ma colli di bottiglia rimangono all'intero e non permettono miglioramenti come ci si aspetterebbe. Allo stesso tempo, è impossibile fare testing generalizzato data la varietà di versioni che esistono di VDE: il Profiling diventa complesso in casi come Switchless VDE e VDE in Memoria Condivisa dato che non hanno un vero e proprio processo; anche in altri casi, come in VDE2 spesso il collo di bottiglia è difficile da individuare anche con informazioni di profiling, dato che deriva da combinazioni di più applicazioni che lavorano in concomitanza.
Infine data la mancanza di modularità è difficile individuare i colli di bottiglia sopracitati e risolverli data l'impossibilità di testare le singole parti del framework, al massimo le singole applicazioni, ma come detto prima, spesso il problema è nelle interazioni delle stesse.
Soluzioni come Packetdrill possono trovare bug, e problemi di trasparenza, e individuare la presenza di colli di bottiglia in un infrastruttura, ma non possono indicare con esattezza da quale parte del codice sono causati, mentre

soluzioni come Mininet sono semplicemente un supporto allo sviluppo e non sono in grado di individuare i singoli problemi.

2.1 Scelte di Design

Di conseguenza è necessario sviluppare qualcosa di specializzato, flessibile e programmabile che possa risolvere questi problemi. Essendo un singolo tool è pressoché impossibile da sviluppare, la soluzione più ovvia diventa costruire un API che permetta di costruire test in modo flessibile, permetta di fare test di performance, usare al suo interno applicativi come packetdrill e, dove possibile, individuare colli di bottiglia, tentare di capire da cosa sono causati.
Per questo è stato deciso di costruire un framework per fare test di VDE, creando uno scheletro per creare Test in modo veloce e pratico, e costruire programmi che li eseguano in modo rapido e intuitivo; è stato deciso di implementarlo il Go, dato che il sistema di interfacce[1.3.2] rende facile costruire framework del genere, e il sistema concorrente aiuta l'uso di svariate operazioni che lavorano in modo concorrente tra di loro, necessario per fare svariate operazioni di rete contemporaneamente.

2.2 Infrastruttura

La scelta dell'Infrastruttura di test è stata scelta dopo alcuni esperimenti falliti nel fare test usando più di una interfaccia tap[22] sullo stesso device; Il comportamento standard di Linux in caso di invio di messaggi di rete a interfacce locali consiste nel dirottare tutto sull'interfaccia di loopback rendendo complesso l'invio di messaggi attraverso una rete VDE; per quanto possibile cambiare questo comportamento, è stato deciso che, era in primis controintuitivo, nonché scorretto a livello formale, chiedere ad un utente di cambiare impostazioni di sistema per usare un sistema di Test e Debug di un

applicazione, nonostante avrebbe reso il lavoro molto più pulito e semplice. Di conseguenza è stato scelto di usare un sistema più similare a quello di Packetdrill inizialmente, puntando a prendere idee anche da Mininet per sviluppi futuri; abbiamo quindi, per ogni Test, una parte server in ricezione e salvataggio dei dati, e una parte client in invio.
Viene configurata un interfaccia di rete tap tramite tunctl[23], lanciato uno switch VDE collegato a tale interfaccia e una VM[1] con l'indispensabile per lanciare l'eseguibile lato Client(dato che non usiamo pcap lato client non serve neppure l'installazione di pcap), viene poi lanciato il client, che come descritto successivamente attenderà i messaggi di controllo dal server per poter partire a inviare dati. Questa infrastruttura è provvisoria ed eventuali migliorie verrano toccate negli Sviluppi Futuri.

2.3 API

Affrontiamo più nel dettaglio l'API che è stata costruita.

2.3.1 Test

Vengono specificati prima di tutto 2 tipi di test, un test solo lato server **TestServer**, utile per tutti i tutti i test che non necessitano di un'architettura client server, e un **Test** generale sia lato client che lato server.
Dato che Test, deriva da TestServer, analizziamo prima TestServer, definito da questa interfaccia:

```
type TestServer interface {
    StartServer()
    AddStat(s Stat)
    statManager()
    IFace() *net.Interface
    Name() string
```

[1] Virtual Machine

```
    Address() net.Addr
}
```

ha una serie di metodi pubblici assimilabili a dei Getter (IFace, Name, Address), e 2 metodi specifici il primo **StartServer** verrà usato per far partire la parte che farà da server del test e AddStat ci permette di aggiungere le statistiche che ci interessa recuperare per quel dato test, infine statManager ci assicura la presenza di uno StatManager del quale parleremo più avanti.
Poi l'interfaccia **Test** più generale definito per embedding di TestServer al quale aggiungiamo il metodo StartClient per far partire il lato client del Test (altrimenti definibile come lato Guest):

```
type Test interface {
StartClient()
TestServer
}
```

2.3.2 Stat

Altra interfaccia molto importante è **Stat** definita come segue:

```
type Stat interface {
    Start()
    Stop()
    SetWaitGroup(wg *sync.WaitGroup) error
}
```

Ogni statistica deve poter partire (metodo Start) e finire (metodo Stop): inoltre deve essere possibile impostare un **WaitGroup**[18] da parte del Test di cui fa parte, in modo tale da essere sicuri che tutte le statistiche, siano essere profiling, prese con PCAP o da un programma esterno siano correttamente terminate prima di continuare con il test successivo.
WaitGroup è semplicemente un oggetto del pacchetto sync nato per attendere la fine di una o più goroutine: chiunque lanci il metodo **Wait** attenderà che il

WaitGroup raggiunga il valore 0, le goroutine lanceranno i metodi Add(i int) e Done() per segnalare quando iniziano una nuova task e quando la finiscono, rispettivamente alzando e diminuendo il valore del WaitGroup. Quando le task finiscono, tutte le goroutine che sono in Wait() verranno sbloccate e continueranno la loro eseguzione, ogni statistica avrà il suo file in cui salvare le informazioni richieste.

2.3.3 StatManager

Uno StatManger è invece un oggetto concreto che non contiene altro che una array di Stat e un WaitGroup (lo stesso che verrà impostato nelle singole Stat) e un booleano che controlla se le statistiche sono in corso o meno; questo oggetto altro non fa che occuparsi di lanciare e terminare le statistiche ed è così definito:

```
type StatManager struct {
tatManager struct {
    stats []Stat
    wg    *sync.WaitGroup
    started bool
}
```

Essendo questo il primo oggetto concreto che vediamo, scopriamo che Go non ha il concetto di costruttore, quindi è necessario usare un qualche tipo di Pattern di creazione, il linguaggio ti spinge ad usare un Factory Method:

```
func NewStatManager() StatManager
```

verrà usato questo pattern per tutto il progetto. Questa è forse la versione più semplice dato che tutti gli StatManger sono uguali quando vengono istanziati. Ha tre metodi pubblici: Add, Start e Stop. Il primo aggiunge una nuova statistica, gli altri due si spiegano da soli, servono infatti a mandare i segnali di inizio e fine a tutte le statistiche.

2.3.4 TestRunner

Infine **TestRunner** ha la stessa funzione di StatManager ma applicata ai Test, si occupa di eseguirli in sequenza, ma dove le Stat sono eseguite in goroutine separate da quella principale, i Test sono esequiti sequenzialmente sulla stessa goroutine, per non interferire l'uno con l'altro.
Sia StatManger che TestRunner hanno anche un ulteriore scopo: dato che il loro metodo Add aggiunge un interfaccia (rispettivamente Stat e Test), ci permette di controllare che tale interfaccia sia rispettata quando scriviamo test e statistiche.

2.4 Implementazione di Test e Statistiche

Per provare il funzionamento del framework sono stati costruiti alcuni Test e Statistiche, pensati per lavorare su VDE2, e individuare colli di bottiglia, problemi e regressioni di prestazioni.
Partiamo spiegando le Statistiche dato che verranno successivamente usate nei test:

2.4.1 ProfilingStat

Questa statistica si occupa di recuperare le informazioni di profiling dello switch VDE da /proc[24] usando la libreria procfs [1.3.5]; nello specifico recupera cputime del processo, il numero di context switch (volontari e involontari), la memoria occupata dal processo e il numero di thread che il processo usa.
Come tutte le statistiche ha una funzione di Start, che lancia una in Goroutine il metodo che effettivamente si occupa di recuperare le statistiche, e ha canale di sincronizzazione che attende un segnale per terminare la goroutine di recupero delle informazioni di profiling.
Il metodo Stop altro non fa che mandare questo segnale, mentre alla gorouti-

ne è dato il compito di gestire il WorkGroup, all'inizio dell'eseguzione dovrà essere iniviato il comando Add(), e alla fine (usando defer[2]) un comando Done(), questa statistica e' molto semplice, ma comunque molto importante dato che ci permette di controllare durante tutta la durata di un test il comportamento dello switch VDE e individuare quali operazioni possono causare problemi.

2.4.2 TCPStat

Questa è la statistica più complessa da implementare, perchè necessita di un uso intensivo della libreria gopacket [1.3.5][21] e di svariate funzionalità di Go.
Questo test recupera informazioni relative ad un singolo stream di dati in ricezione, come, nello specifico, la Larghezza di banda, il numero di pacchetti ricevuti, il totale di dati ricevuti, il bitrate e il numero di pacchetti scartati o outOfOrder, tutto questo grazie alla possibilità di usare pcap per sniffare i pacchetti dall'interfaccia di rete e di ricostruire degli stream di dati TCP.
Come ProfilingStat, TCPStat implementa correttamente una statistica, gestisce un WorkGroup e attende un segnale di fine per poter terminare, ma, dovendo ricostruire uno stream TCP, è molto più complesso.
Prima di tutto abbiamo bisogno di creare il nostro assembler, per farlo abbiamo bisogno di un uno streamPool, che abbiamo deciso di chiamare **statStreams**, e una factory per crearne di nuovi quando necessario.
Per implementare correttamente statStream sono necessari due metodi: Reassembled, chiamato ogni volta che ci siano nuovi dati disponibili in lettura, e ReassmblyComplete, chiamato quando uno stream è stato riassemblato completamente; il primo metodo si occupa di aggioranre i dati dello stream, il secondo di salvarli su file.

Così inizializziamo il nostro assembler:

[2]La funzione defer permette di eseguire un istruzione al termine della funzione in cui è stato dichiarato

```
streamFactory := &statsStreamFactory{s.logger}
streamPool := tcpassembly.NewStreamPool(streamFactory)
assembler := tcpassembly.NewAssembler(streamPool)
```

Da notare che passiamo anche un logger quando creiamo la factory, in modo che ogni stream possa salvare i dati correttamente nel file di log. Per usare correttamente il nostro assembler abbiamo necessità di un decoder di pacchetti:

```
var eth layers.Ethernet
var dot1q layers.Dot1Q
var ip4 layers.IPv4
var ip6 layers.IPv6
var ip6ext layers.IPv6ExtensionSkipper
var tcp layers.TCP
var payload gopacket.Payload
parser := gopacket.NewDecodingLayerParser(layers.LayerTypeEthernet,
                  &eth, &dot1q, &ip4, &ip6, &ip6ext, &tcp, &payload)
```

e di recuperare i pacchetti dall'interfaccia di rete che andiamo ad usare:

```
handle, err := pcap.OpenLive(s.iface.Name, int32(s.snaplen), true, flushDuration/2
source := gopacket.NewPacketSource(handle, handle.LinkType())
```

ogni pacchetto, infine, dopo essere stato validato dal decoder, verrà inviarlo all'assembler:

```
assembler.AssembleWithTimestamp(packet.NetworkLayer(.NetworkFlow(, &tcp, packet.
```

l'assembler si occuperà poi di salvare i dati riguardo questo stream e dirci quando ha terminato.

2.4.3 BandwidthTest

BandwidthTest è pensato per ottenere informazioni relative alla Larghezza di Banda; altro non fa che inviare grandi quantità di dati tra il client e

il server attraverso una singola connesione TCP che invia grandi quantità di dati, in tal modo possiamo vedere come si comporta lo switch durante una singola connessione che supera il quantitativo di dati che può processare al secondo, nello specifico inviamo 1GB di dati usando i metodi spiegati nella Sezione 2.5.
Questo Test lavora sia lato guest che lato host, rispettivamente client e server nel nostro modello. Il client viene lanciato per primo e attende un messaggio di controllo per iniziare ad inviare dati, mentre il server manda il messaggio di controllo una volta che il setup del test è stato preparato e appena prima di lanciare le statistiche, una volta lanciato il comando di controllo si prepara a ricevere i dati; l'esecuzione terminerà quando le stastiche saranno state tutte terminate correttamente, si passerà quindi al test successivo in coda a TestRunner.
Per quanto riguarda il lato client, invece una volta ricevuto il messaggio di controllo, si inizierà ad inviare i dati, una volta fatto il Test termina e si attenderà l'arrivo del prossimo messaggio di controllo dal prossimo test in coda a TestRunner.
Le statistiche associate a questo test sono sia ProfilingStat sia TCPStat, dato che siamo interessati sia alla banda, sia al comportamento dello switch in rispota a questo tipo di richieste.

2.4.4 LatencyTest

LatencyTest è invece un test solo lato server, non ha necessità di nessun ricevitore, ne usa particolari statistiche. Usa un implementazione di fastping per go, e recupera le informazioni necessarie direttamente dal ping, principalmente latenza e RTT[3].

[3]Round Trip Time

2.4.5 StressTest

Quest'ultimo Test è quello più complesso: è pensato per usare tutte le risorse possibili dello switch e per catturare informazioni di profiling sotto questo tipo di stress.
Vengono aperte svariate porte TCP sulla macchina host dove verranno inviati dati di svariate dimensioni continuamente da parte del client per un certo quantitativo di tempo, scaduto il quale, il server invia un messaggio di controllo al client per segnalare di fermarsi, una volta chiuse tutte le connessioni attive, un messaggio di ack viene mandato indietro e il server può quindi chiudere le porte e finire di loggare statistiche.
La complessità in questo Test è tutta nella gestione concorrente delle connessioni e degli scambi di informazioni necessari tra client e server, sono infatti necesassari svariati metodi di sincronizzazione, ogni goroutine di invio e di ricezione necessita un canale per dare il segnale di terminazione, 2 WorkGroup sono usati oltre che per le statistiche (come da protocollo) per attendere la terminazione di ogni singolo socket in ascolto e ogni Goroutine di invio per assicurarsi che non ci siano memory leak e errori di connesione che possono causare un crash del Test.
Per il resto il metodo di invio è similare a quello di BandwidthTest, semplicemetne con l'opzione di inviare quantità di dati randomiche.

2.5 Utils

Infine un sotto pacchetto della nostra libreria definisce una serie di utilità, quali inviare un certo quantitativo di dati attraverso la rete, ricevere quei dati senza perdere tempo a leggerli, leggere e ricevere messaggi di controllo, recuperare l'indirizzo locale dalle interfacce di rete a cui siamo collegati e così via, in modo aiutare a costruire, il più possibile, Test uniformi.
Particolarmente interessante sono SendData e DevNullConnection, due funzioni accoppiate: entrambe utilizzano il sistema di interfacce di Go per inviare e ricevere dati attraverso la rete, senza il delay di doverli leggere, randomiz-

zare o copiare. Una prima implementazione di SendData leggeva da Urandom, ma era estremamente lenta; questa nuova implementazione permette di saturare il canale, arrivando ad un centinaio di MB[4] al secondo, quando l'implementazione che utilizza Urandom faticava ad arrivare ai 40.
L'implementazione della funzione di ricezione, chiamata **DevNullConnection** è la seguente:

```
func DevNullConnection(conn net.Conn, wg *sync.WaitGroup) error {
    if wg != nil {
        wg.Add(1)
        defer wg.Done()
    }
    _, err := io.Copy(devNull, conn)
    if err != nil {
        return err
    }
    return nil
}
```

da notare come si usa io.Copy per ricevere dati, invece di usare il metodo Read di net.Conn: questo è possibile grazie al sistema di di interfacce implicite di go, infatti una Connessione di rete implementa l'interfaccio io.Reader e io.Writer permettendo usi di questo tipo senza la necessità di esprimere esplicitamente relazioni di sottotipo. L'implementazione di SendData è equivalente, ma usa devZero invece di devNull e usa CopyN per mandare una speicifica quantità di dati invece di Copy
Interessanti sono **devZero** e **devNull**, essi infatti permettono, il primo, di inviare una struttura dati vuota, che si legge direttamente dalla ram, leggendo il minor quantitativo di dati possibili, e si può usare in CopyN perchè implementa l'interfaccia io.Reader, il secondo, devNull invece è un implementazione di io.Writer e permette di ricevere dati da uno stream, solo scartando quei dati. Di seguito l'implementazione di entrambi:

[4]Megabyte

```
type zeroFile struct{}
type nullFile struct{}

func (d *nullFile) Write(p []byte) (int, error) {
    return len(p), nil
}

func (d *zeroFile) Read(p []byte) (int, error) {
    return len(p), nil
}

var devNull = &nullFile{}
var devZero = &zeroFile{}
```

La semplicità con cui si può costruire questo tipo di costrutti è uno dei punti forti di Go, ed uno dei motivi per cui è stato scelto per questo progetto. Il resto delle utility verrano trovate nell'appendice A

Conclusioni

Con questo lavoro, si spera di migliorare lo sviluppo di VDE, rendendolo più semplice ed efficace, sopratutto ora che, con lo sviluppo di VDE4 e, si spera, la continuazione di VDE 3, un'architettura più modulare è all orizzonte.
Facilitare la creazione di Test permette non solo di testare le prestazioni di nuovi componenti, ma anche di controllarne la correttezza durante lo sviluppo ed assicurarsi di non aver introdotto bug e colli di bottiglia.
Mettere sotto Stress un componente alla fine di ogni ciclo di sviluppo permette di controllare la mancanza di errori significativi a Runtime, e il corretto funzionamento in tutti i possibili test case.

Le tecnologie di virtualizzazione di rete negli ultimi anni, sia aperte che chiuse, si sono moltiplicate, e VDE, nonostante sia uno dei primi esempi di questo tipo di tecnologia e sia tutt'ora un software molto potente e versatile, è stato lasciato indietro a causa del piccolo numero di sviluppatori che ad oggi ci lavora. Questo è dovuto alla difficolta di sviluppo di nuovi componenti e di estensione dei vecchi, come illustrato nel Capitolo 1.1.1. Un sistema di Test standard aiuta a costruire un metodo di sviluppo e quindi ad attirare nuovi sviluppatori, che sono necessari a VDE per continuare a crescere come progetto.

Questo progetto tenta di offrire uno scheletro per creare questo tipi di Test; ogni test andrà poi pensato per i diversi tipi di componenti. i test al momento

implementati possono essere utili per gli Switch di VDE2, ma, ad esempio, le statistiche di profiling non sono utilizzabili in progetti quali Switchless VDE o VDE in memoria condivisa.

Sviluppi Futuri

Introdurre un sistema di test ad un Framework complesso e vario come è VDE non è compito banale, per questo è stato deciso di costruire un Framework di Test.
Perciò gli sviluppi futuri sono molti, a partire da quelli architetturali. Alcune di essi prendono ispirazione da Mininet, e sono consistono principalmente nel dare più controllo dell'ambiente di test ai Test stessi:

- Lasciare impostare e gestire l'interfaccia tap direttamente dal framework lato server, in modo da limitare ulteriormente il quantitativo di lavoro di chi è interessato a fare test.

- Lanciare VDE direttamente dal programma, specificando l'eseguibile; questo ci permetterà di usare uno switch nuovo ogni volta che facciamo un nuovo test, così da assicuare un Test pulito, senza influenze di nessun tipo sull'applicativo che andiamo a Testare.

- Controllare la VM direttamente dal programma, facendo redirect dei delle tty[15], in modo da diminuire il numero di iterazioni e problemi di sincronizzazione, infatti potrà essere il test stesso a decidere quando e come lanciare la sua parte client senza necessità di messaggi di controllo.

Queste sono alcune ipotetiche modifiche interessanti, sull'infrastruttura, che dovranno essere studiate a dovere prima di essere implementate, in quanto, nonostante siano di aiuto durante l'eseguzione dei test, potrebbero complicarne molto la scrittura.

Un progetto futuro, in particolare tentendo in mente VDE 3 e VDE4, è costruire dei test specifici per le varie tipologie di moduli, così da essere in grado di fornire anche dei test d'integrazione.

Sempre relativamente allo sviluppo di Test, sarebbe interessate integrare Packetdrill in modo da sfruttare l'infrastruttura di VDETesting per velocizzarne e standardizzarne l'uso nello sviluppo di VDE: in questo caso sarebbe necessario sviluppare anche uno script packetdrill apposito [8].

Altra modifica strutturale che necessità di una libreria a parte sarebbe una libreria goVDE per collegarsi direttamente ad uno switch senza passare da tap, questo necessita di una libraria Go di livello molto basso, nonché dell'implementazione di una Packet Source per Gopacket, ma aiuterebbe a garantire il funzionamento corretto di VDE in un maggior numero di situzione, e per tutte quelle versioni che tutt'ora non hanno la possibilità di collegarsi ad un interfaccia tap.

Questo Framework è giovanissimo, e nonché molto aperto a modifiche e ripensamenti, è una struttura con la quale affrontare il problema del testing della virtualizzazione di rete, pensato per il modello che offre VDE, quindi ha innumerevoli possibilità di sviluppo futuro, quelle qui presentate sono solo una selezione tra quelle ritenute più interessanti da chi scrive.

Appendice A

Prima Appendice

```
package utils

import (
  "encoding/binary"
  "errors"
  "io"
  "log"
  "net"
  "sync"
  "time"
)

const (
  kb int64 = 1000
  mb int64 = 1000 * kb
  gb int64 = 1000 * mb
)

type zeroFile struct{}
```

```
type nullFile struct{}

func (d *nullFile) Write(p []byte) (int, error) {
  return len(p), nil
}

func (d *zeroFile) Read(p []byte) (int, error) {
  return len(p), nil
}

var devNull = &nullFile{}
var devZero = &zeroFile{}

//DevNullConnection take a connection on the receive end, get all data
//and put into an empty reader
func DevNullConnection(conn net.Conn, wg *sync.WaitGroup) error {
  if wg != nil {
    wg.Add(1)
    defer wg.Done()
  }
  _, err := io.Copy(devNull, conn)
  if err != nil {
    return err
  }
  return nil
}

//WaitForControlMessage open a listener on port 8999 to get control messages
func WaitForControlMessage(address string, msg int) error {
  var arrived = false
  clistener, err := net.Listen("tcp", address+":8999")
```

```
  if err != nil {
    return err
  }
  for !arrived {
    conn, err := clistener.Accept()
    if err != nil {
      log.Fatal(err)
    }
    var buf int32
    binary.Read(conn, binary.LittleEndian, &buf)
    if buf == 2 {
      arrived = true
      clistener.Close()
    }
  }
  return nil
}

//SendControlSignal send a message to a TCP address
func SendControlSignal(address string, msg int32) error {
  log.Printf("sending control message to %v", address+":8999")
  conn, err := net.Dial("tcp", address+":8999")
  if err != nil {
    return err
  }
  defer conn.Close()
  err = binary.Write(conn, binary.LittleEndian, msg)
  if err != nil {
    return err
  }
  return nil
```

```
}

//SendControlSignalUntilOnline repeat SendControlSignal until no error return from
func SendControlSignalUntilOnline(address string, msg int32) {
  for {
    if err := SendControlSignal(address, msg); err != nil {
      time.Sleep(1 * time.Second)
      continue
    } else {
      log.Print("control message delivered")
      break
    }
  }
}

//SendData send size data (in megabytes)to the string addr
func SendData(addr string, size int64) error {
  _, err := net.ResolveTCPAddr("tcp", addr)
  if err != nil {
    return err
  }
  conn, err := net.Dial("tcp", addr)
  if err != nil {
    return err
  }
  defer conn.Close()
  n, err := io.CopyN(conn, devZero, size*(mb))
  if err != nil {
    return err
  }
  if n != size*mb {
```

```
    log.Printf("couldnt send %v Megabytes", float64(n)/float64(mb))
    return nil
  }
  return nil
}

//Localv4Addr get the first local ipv4 address that is not loopback
func Localv4Addr() (string, error) {

  addrs, err := net.InterfaceAddrs()
  if err != nil {
    return "", err
  }
  for _, address := range addrs {
    if ipnet, ok := address.(*net.IPNet); ok && !ipnet.IP.IsLoopback() && ipnet.
      return ipnet.IP.String(), nil
    }
  }
  err = errors.New("No non local Ip adress found")
  return "", err
}

//Localv6Addr get the first local ipv4 address that is not loopback
func Localv6Addr() (string, error) {

  addrs, err := net.InterfaceAddrs()
  if err != nil {
    return "", err
  }
  for _, address := range addrs {
    if ipnet, ok := address.(*net.IPNet); ok && !ipnet.IP.IsLoopback() && ipnet.
```

```
      return ipnet.IP.String(), nil
    }
  }
  err = errors.New("No non local Ip adress found")
  return "", err
}

//InterfaceAddrv4 Get the ipv4 address of a specific interaface
func InterfaceAddrv4(iface *net.Interface) (string, error) {
  addrs, err := iface.Addrs()
  if err != nil {
    return "", err
  }

  for _, address := range addrs {
    if ipnet, ok := address.(*net.IPNet); ok && !ipnet.IP.IsLoopback() && ipnet.IP
      return ipnet.IP.String(), nil
    }
  }
  err = errors.New("No non local Ip adress found")
  return "", err
}
```

Bibliografia

[1] Renzo Davoli, Micheal Goldweber. VirtualSquare: Users, Programmers and Developers Guide.
http://www.lulu.com/product/paperback/virtual-square-users-programmers-developers-guide/14698606

[2] Luca Biliardi. Design e Implementazione del nuovo Framework Virtual Distribuited Ethernet: Analisi delle Prestazione e Validazione sulla Precedete Architettura. Tesi di Laurea Specialistica in Informatica 2008/2009.

[3] Neal Cardwell, Yuchung Cheng, Lawrence Brakmo, Matt Mathis, Barath Raghavan, Nandita Dukkipati, Hsiao-keng Jerry Chu, Andreas Terzis, Tom Herbert. packetdrill: Scriptable Network Stack Testing, from Sockets to Packets. 2013.
https://www.usenix.org/conference/atc13/packetdrill-scriptable-network-stack-testing-sockets-packets

[4] Google. Packetdrill https://code.google.com/p/packetdrill/

[5] Mininet Team. Mininet: An Instant Virtual Network on your Laptop http://mininet.org

[6] Mininet Samble Workflow http://mininet.org/sample-workflow/

[7] Bob Lantz, Brandon Heller, Nick McKeown. A Network in a Laptop: Rapid Prototyping for Software-Defined Netowrk

HotNets 2010. https://github.com/mininet/mininet/wiki/pdf/mininet-hotnets2010-final.pdf

[8] packetdrill Syntasctic Structure https://code.google.com/p/packetdrill/wiki/Syntax

[9] SIGCOMM 2014 Tutorial: Teaching Computer Networking with Mininet. https://github.com/mininet/mininet/wiki/SIGCOMM-2014-Tutorial%3A-Teaching-Computer-Networking-with-Mininet

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

[11] The Go Programming Language. https://golang.org/doc/

[12] Andreaw Gerrand. C? Go? CGo! https://blog.golang.org/c-go-cgo

[13] Google. Effective Go. https://golang.org/doc/effective_go.html

[14] Rob Pike. Go Concurrency Patterns. https://www.youtube.com/watch?v=f6kdp27TYZs

[15] Fabrice Bellard. qemu-doc(1) man page http://linux.die.net/man/1/qemu-kvm

[16] About the go Command. https://golang.org/doc/articles/go_command.html

[17] The Go Programming language Documentation. Package net. https://golang.org/pkg/net/

[18] The Go Programming language Documentation. Package sync. https://golang.org/pkg/sync/

[19] The Go Programming language Documentation. Package io. https://golang.org/pkg/io/

[20] The Go Programming language Documentation. Package os. https://golang.org/pkg/os/

[21] GoDoc: package gopacket. https://godoc.org/github.com/google/gopacket

[22] Maxim Krasnyansky. Universal TUN/TAP device driver. https://www.kernel.org/doc/Documentation/networking/tuntap.txt

[23] Jeff Dike, Matt Zimmerman, Henrik Nordstrom. tunctl(8) man page http://linux.die.net/man/8/tunctl

[24] proc(5) Linux man page http://linux.die.net/man/5/proc

[25] Pfaff, B., Pettit, J., Amidon, K., Casado, M., Koponen, T., & Shenker, S. (2009, October). Extending Networking into the Virtualization Layer. In Hotnets.

[26] Software-Defined Networking (SDN) Definition https://www.opennetworking.org/sdn-resources/sdn-definition

Ringraziamenti

Ed arriviamo finalmente ai ringraziamenti.
Partiamo ringraziamo i miei genitori, che mi hanno supportato economicamente in questi anni.

Per poi Ringraziare i Tapiri Duma, Jonus e Lollum, che con sono stati i miei compagni di avventura molto in questa corso, un gruppo affiatatisimo per tutti i progetti, ma tutti con specialità e interessi molto diversi.
Ringrazio il resto della combricola reggiana, uovo e bagna, che dall'alto dell'esserci passati prima di noi, hanno aiutato parecchio nella sopportazione e nella formazione di questo nerd, e Donj che ora è in irlanda a fare il lavoro che si merita.
Ringrazio la vecchia guardia di CS che ha accolto il marmocchio di turno e formato a colpi di acido, calci e odio in un informatico degno di questo nome.

Non possono mancare nei ringraziamenti lo Zippolo e Talisa che hanno riletto l'orrore che era questa tesi per aiutarmi a correggerla, insieme e tutti gli amici su Gente che Telegramma che hanno sopportato i miei improperi durante tutto il lavoro fatto per questa tesi, ridendo delle mie disgrazie e facendomi pat pat sulla testa o distraendomi fancedomi bere o giocare.
Grazie all'Alex e la Lavi, due delle persone che ho più care, per tenere a bada il mio Ego e sopportarmi da così tanti anni, ad Alberto, che nonostante la distanza mi intrattiene sempre con discussioni interessanti, al Vellu e a Ezio per le battute sarcastiche che fanno sempre bene e ti ricordano di non

prenderti troppo sul serio.
Grazie ai compagni di Aikido che hanno sopportato i miei deliri in questi giorni e mi hanno ridato energie mentali sbattendomi sul tatami manco fossi un tappeto, rilassandomi come non mai.
Un ringraziamento al buon Ghini e Giovanni, che mi hanno, rispettivamente, spedito in America e tenuto in America, per un esperienza estremamente formativa; ringrazio anche i miei Colleghi lì, che mi hanno sopportato in un periodo molto difficile.
Grazie a chi ha deciso di sviluppare software opensource, che mi permette di imparare ogni giorno qualcosa di nuovo su una materia che amo.
Grazie a tutti coloro che mi hanno dato occasione di imparare, di crescere come informatico e come persona in questi anni, chi mi è stato vicino fino alla fine, chi non ha potuto, non sarei chi sono senza di voi.