UNIVERSITÁ DEGLI STUDI DI MODENA E REGGIO EMILIA

DIPARTIMENTO DI INGEGNERIA “ENZO FERRARI”

Corso di Laurea di Ingegneria Informatica

SIMNET, FRAMEWORK PYTHON PER L'EMULAZIONE DI RETI VIRTUALIZZATE

RELATORE
Prof. Riccardo Lancellotti

LAUREANDO
Tommaso Miana

ANNO ACCADEMICO:
2017/2018

INDICE

Abstract

SimNet è un framework scritto in Python che si pone come obbiettivo quello di sfruttare lo User Mode Linux (UML) e il vde_switch per la creazione di reti network simulate.

L'idea è nata dalla necessità di sostituire il programma Marionnet che viene ad oggi utilizzato nelle esercitazioni di laboratorio di Reti di Calcolatori[1].

In particolare, il mio compito all'interno del progetto è stato quello di definire e implementare il back-end di SimNet, che verrà approfondito in questo documento. Altre parti del progetto, relative al front-end non rientrano nel perimetro del presente elaborato.

Lo scopo del mio lavoro è stato quindi quello di interfacciarmi con il sistema operativo e in particolare con lo UML e il vde_switch e di rendere disponibile all'interfaccia grafica una struttura a oggetti che rappresentasse la rete simulata.

Nei prossimi capitoli, dopo qualche premessa sulle problematiche di Marionnet e sugli strumenti utilizzati, verranno descritti la sua struttura e il funzionamento del back-end di SimNet. Successivamente verrà inquadrato e discusso il futuro del programma come effettivo sostituto di Marionnet, considerando anche i bug emersi e i possibili miglioramenti.

[1] esame del terzo anno della Laurea Triennale di Ingegneria Informatica al Dipartimento di Ingegneria "Enzo Ferrari" di Modena

1 Introduzione

1.1 Marionnet e le sue problematiche

"Marionnet is a virtual network laboratory: it allows users to define, configure and run complex computer networks without any need for physical setup"

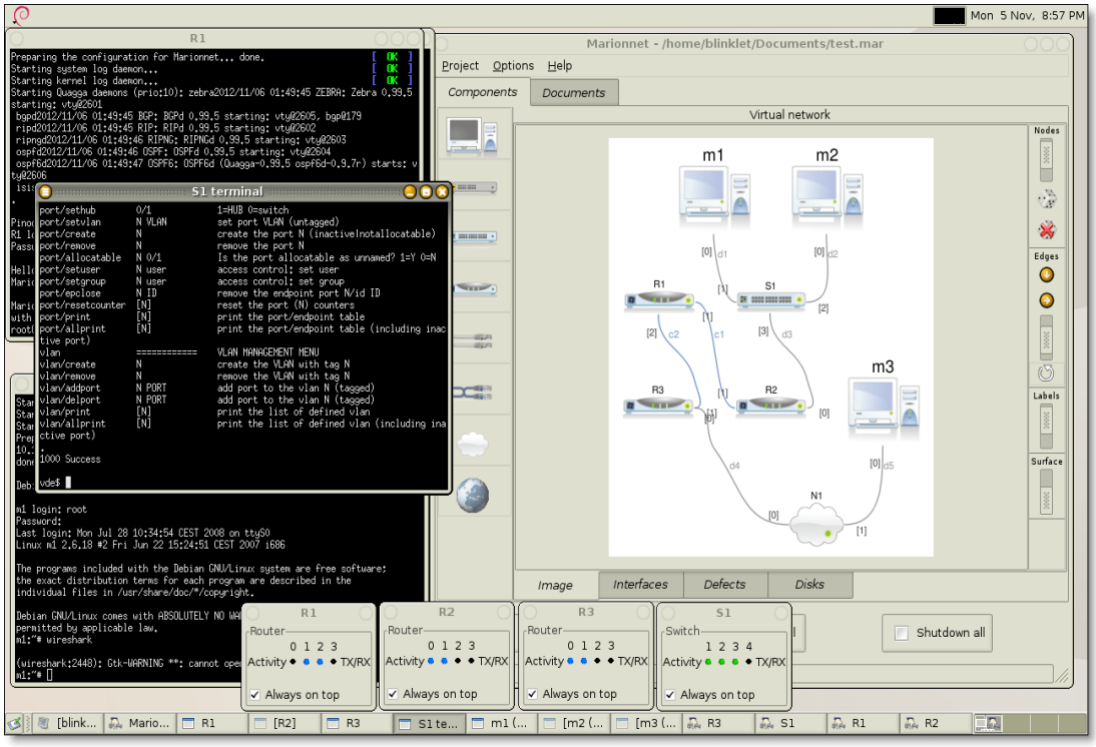


Figura 1: Marionnet

Marionnet è un software open source scritto da Jean-Vincent Loddo in linguaggio Ocaml, basato sull'utilizzo di User Mode Linux come metodo di avvio e gestione delle macchine virtuali.

La prima problematica che subito emerge è che tale linguaggio è molto complicato e poco intuitivo. Ciò ovviamente complica notevolmente la manutenibilità del framework e in pratica nel caso di aggiornamenti o estensioni

risulta molto difficile andare a metterci mano e modificare qualche stringa di codice.

Al livello funzionale invece, Marionnet presenta diversi problemi, alcuni abbastanza gravi, in fase di avvio. Può succedere infatti che all'avvio di una rete, dopo aver perso tempo per creare tutta la serie di oggetti, questa non riesce in nessun modo a comunicare. L'unica soluzione trovata a questa situazione risulta quella di spegnere tutto e riconfigurare una rete da zero.

È stato anche riscontrato che, in seguito all'avvio della rete, Marionnet non è spesso in grado di gestire il crash o la chiusura involontaria di un Host o uno Switch.

Il software è inoltre caratterizzato da un'interfaccia grafica minimale e poco intuitiva (Fig. 1) che comporta una serie di scomodità di utilizzo. Queste ultime non verranno analizzate in questo documento in quanto la risoluzione di tali problemi non rientra negli obbiettivi del mio lavoro.

Presa visione della situazione attuale l'idea del progetto è quella di creare un framework semplice ed intuitivo che partisse dall'idea di fondo di Marionnet ma che la migliorasse e la rendesse più efficiente. Nello specifico l'attenzione è stata posta soprattutto nel risolvere i problemi di avvio del network e di gestione dei crash delle macchine. Per farlo sono stati implementati metodi di controllo delle stringhe di configurazione dei singoli elementi (interfacce di rete, dischi...), metodi per la creazione del comando di lancio della macchina e dello switch. Tutto ciò verrà analizzato meglio in seguito.

1.2 UML, User Mode Linux

"User-Mode Linux is a safe, secure way of running Linux versions and Linux processes. Run buggy software, experiment with new Linux kernels or distributions, and poke around in the internals of Linux, all without risking your main Linux setup. "

UML esegue il kernel di Linux come se fosse un normale programma, aggiungendo uno strato software tra il kernel "Host" di sistema e un altro kernel UML che viene eseguito come un normale processo. In questo modo il kernel UML non ha accesso diretto all'hardware e viene eseguito in un ambiente protetto, analogamente a quello che avviene per le macchine virtuali. A questo proposito è importante distinguere fra la virtualizzazione di UML e quelle effettuate da software come ad esempio *Vmware*.

In sostanza con la User Mode Linux si crea un sistema operativo virtuale, ma la parte hardware resta sempre quella dell'Host, a differenza di *Vmware* che crea una vera e propria macchina virtuale, con il suo hardware e software.

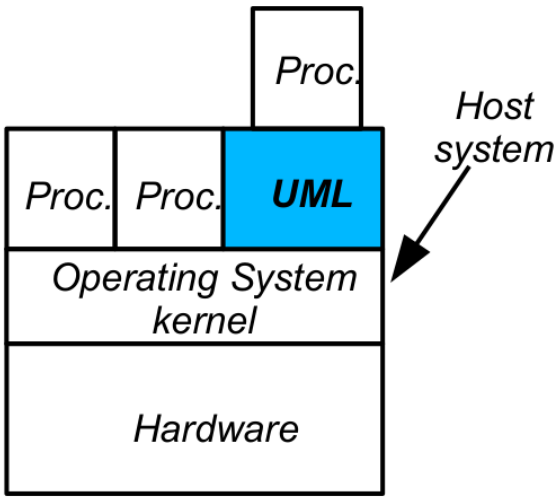


Figura 2: User Mode Linux

Se ne deduce quindi che UML è un processo per l'Host e allo stesso tempo un kernel per i processi che girano su di esso.

Questo tipo di virtualizzazione risulta utile in diversi campi d'utilizzo: testing di configurazioni di sistema, debugging del kernel UML e in particolare, quello che più ci interessa in questo contesto, creazione di reti virtuali.

1.2.1 Avvio e comandi

Per lanciare UML sono necessari due componenti fondamentali: un kernel e un filesystem. Entrambi possono essere trovati in rete e in particolare sul sito ufficiale dello User Mode Linux. Il primo consiste in un file eseguibile che funge quindi da comando di avvio della macchina. Una volta creato questo file, solitamente denominato *"linux"*, basterà eseguirlo passandogli i parametri necessari per creare il sistema operativo virtuale. Tale file è quello che avvia di fatto la UML e rappresenta il kernel del sistema operativo che verrà creato.

Il parametro più importante, l'unico obbligatorio, da passare è appunto un filesystem funzionante, organizzato come se fosse un disco rigido. Per farlo si utilizza il parametro *ubd<N>* che ha la seguente sintassi:

ubd<N>=<cow_file>,<image_file>

- Dove *ubd<N>* identifica il disco virtuale da aggiungere alla macchina. È obbligatorio indicare almeno ubd0 che indicherà il filesystem avviato come root. Successivamente è possibile aggiungere dischi secondari (ubd1, ubd2…) con i rispettivi filesystem.
- *<image_file>* è il nome del file con l'immagine del filesystem.
- *<cow_file>* è il nome del file di appoggio creato all'avvio di tipo copy on write, di cui parleremo in seguito.

Secondo parametro in ordine di importanza per la creazione di una rete virtuale è *eth<N>*, che identifica la n-esima interfaccia di rete della macchina. Questo parametro è necessario per interfacciarci con altre macchine in quanto definisce un collegamento tra di esse. Ogni interfaccia può lavorare in diverse modalità, tra cui *ethertap, tun/tap* e *mcast.* Quest'ultima permette all'host di rendersi visibile nella rete senza necessità di collegarsi ad un dispositivo di rete virtuale.

Un'altra possibile configurazione consiste appunto nel collegamento dell'interfaccia ad un dispositivo come uno switch (o hub). Questa in particolare è stata la soluzione adottata per la realizzazione del progetto, poiché permette la realizzazione di reti virtuali ben definite e molte complesse rispetto ad esempio alla semplice *mcast*.

Per quanto riguarda la sintassi del comando questa dipende dal dispositivo di rete da collegare. Uml_switch e vde_switch sono i due possibili switch virtuali con i quali Use Mode Linux può lavorare e si è scelto di adottare il secondo in questo contesto.

eth<N>=vde,<socket_dir>,<mac_address>,<port_number>

- *vde* sta appunto ad indicare che si sta utilizzando lo stack vde per comunicare con il vde_switch.
- *<socket_dir>* indica il nome del file rappresentante lo switch.
- Gli ultimi due parametri sono facoltativi.

Gli altri parametri facoltativi per il lancio del kernel sono:

- *mem=<MB>* ovvero la quantità di memoria centrale.
- *umid=<uml_mconsole>* definisce una console con la quale l'Host può comunicare con la macchina UML.

Qui un esempio di una possibile configurazione:

linux ubd0=h1.cow,rootfilesystem.ext4 mem=128M eth0=vde,/tmp/s1,,5 \ umid=h1_cmd

La *uml_mconsole* in realtà è estremamente importante in quanto permette di modificare e controllare la macchina una volta avviata. Per questo motivo mi appresto ad approfondire l'argomento.

1.2.2 UML console

Permette di gestire ogni istanza lanciata con UML attraverso una console dedicata. Come detto in precedenza il nome della uml_console viene specificato come parametro nel comando di avvio della macchina. Tale nome rappresenta la socket della console attraverso la quale l'host dovrà comunicare.

Se si vuole comunicare come Host con la macchina UML basterà eseguire il comando *uml_mconsole* seguito dal nome della console e il comando che si vuole lanciare. Es: *uml_mconsole h1_cmd version*.

I comandi eseguibili dalla console sono diversi, in particolare sono stati utilizzati:

- *version,* ritorna la versione del kernel dell'istanza UML. Utilizzato esclusivamente per il controllo di stato della macchina.
- *halt,* effettua uno shutdown sporco del kernel. Utile in situazioni di crash in cui lo shutdown pulito (vedi *cad*) non è possibile.
- *cad*, shutdown pulito del kernel.
- *reboot,* riavvio della macchina.
- *config dev=config,* aggiunge un nuovo dispositivo alla macchina. Usato soprattutto con *eth<N>* come dispositivo (*dev*) per riconfigurare le interfacce di rete.
- *remove dev,* rimuove dispositivo dalla macchina.

Nel framework, come verrà spiegato più in dettaglio in seguito, l'uml_mconsole è stata utilizzata principalmente per controllare lo stato di una macchina (*version*), gestirne la chiusura volontaria o meno e per configurare le interfacce di rete in seguito all'avvio delle macchine (*config eth0*).

1.2.3 COW file

Credo che il miglior modo per spiegare i COW file si citare l'esempio che riporta Jeff Dike nel libro User Mode Linux.

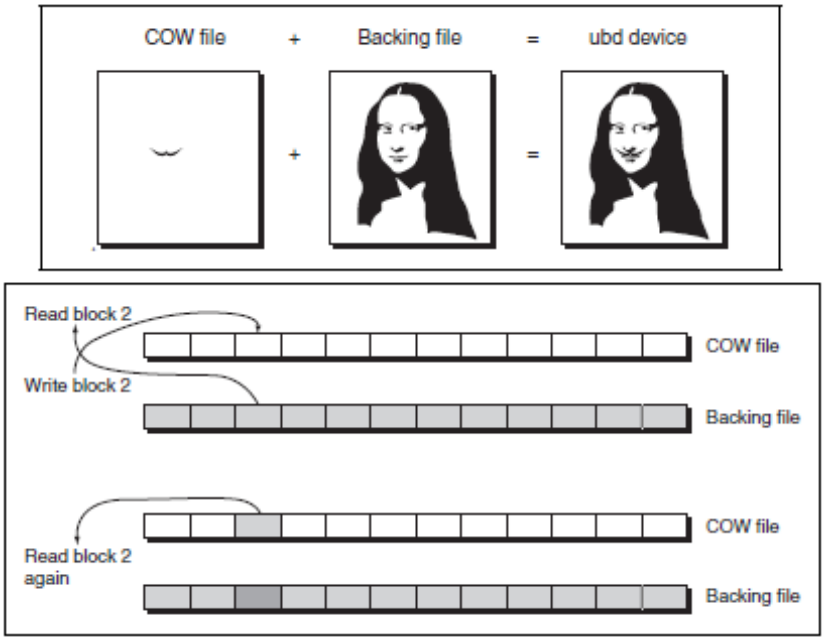


Figura 3: esempio di Jeff Dike, COW file

Questi file, spiega, funzionano come un foglio di plastica trasparente con dei baffi disegnati appoggiato sul dipinto della Gioconda. A prima vista sembra che la Gioconda sia stata rovinata con dei baffi, ma in realtà è solo una modifica presente sul foglio di plastica (file COW) mentre il quadro originale (file di partenza) è rimasto inalterato. Se si pensa al file di partenza come al filesystem delle nostre macchine virtuali questo spiega il loro utilizzo nello UML.

Questi particolari file sono fondamentalmente una copia modificabile di un file system che vogliamo lasciare inalterato in quanto potenzialmente utilizzabile da più di un host UML. Ogni volta che dal terminale della macchina UML avviata si andrà a scrivere su qualche file (ad esempio sul file di configurazione di rete /etc/network/interfaces) modificando il file system, questa modifica verrà salvata

all'interno del file COW, relativo a quella specifica macchina, senza andare ad intaccare il file system originale.

In pratica ogni volta che si vuole definire un disco ubd<N> assegnato alla macchina UML viene passato un filesystem e un relativo file COW dove verranno salvate le modifiche. Si noti che non è necessario creare il file COW prima dell'avvio: se la macchina trova il file leggerà le modifiche ed eseguirà le opportune configurazioni, altrimenti creerà il file come nuovo. Da notare che essendo questi file una copia del filesystem originale, è possibile riferire più dischi ubd allo stesso file di sistema senza creare conflitti, poiché ogni disco si riferisce a un file COW diverso.

Una caratteristica importante di questi file è che sono di tipo *sparse*. Ciò vuol dire che le dimensioni apparenti di questo file su disco sono uguali a quelle del file system a cui sono associati, ma la loro grandezza corrisponde in realtà alla quantità di dati che è possibile leggere da questo file. Se si pensa che un filesystem Linux generico pesa circa 1 o 2 GB, senza questa caratteristica, ad esempio, le reti create con SimNet (o con Marionnet) assumerebbero dimensioni considerevoli, complicando la gestione del salvataggio.

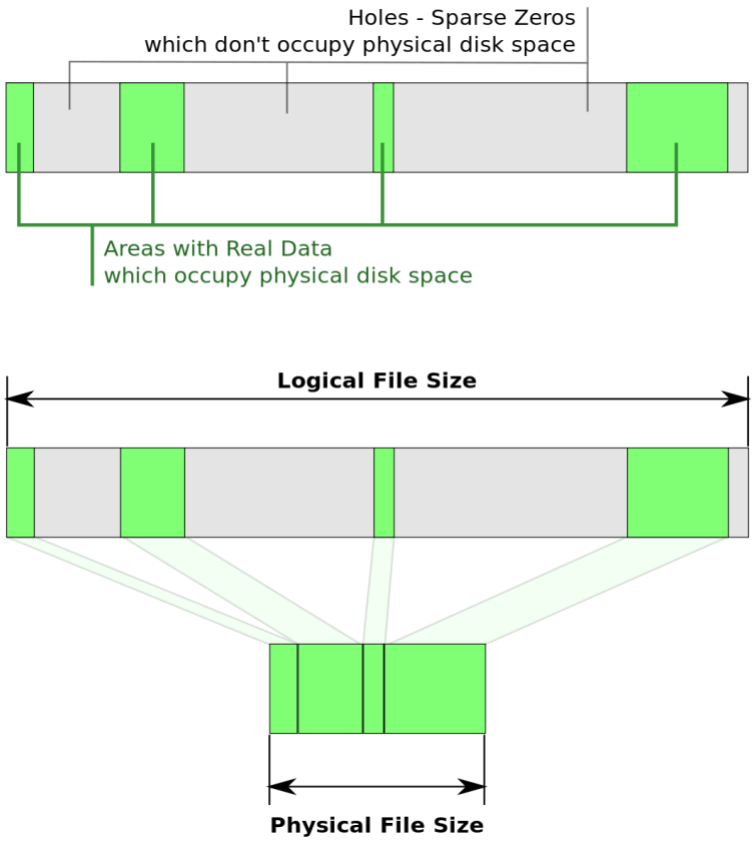


Figura 4: File Sparse

1.3 VDE switch

"VDE is the Virtual Distributed Ethernet. You can use it to connect virtual machines or linux boxes or any combination of the two. Like a real modern Ethernet network, a VDE is composed by switches and cables. Each switch has several sockets where machines can be "plugged-in". "

Come accennato in precedenza sono due le soluzioni possibili per simulare virtualmente un dispositivo di rete che permetta la comunicazione ethernet tra due host: uml_switch e vde_switch.

Uml_switch è lo switch dedicato per la UML, che non ho utilizzato in questo progetto. Ho preferito invece vde_switch in quanto presenta una configurazione più semplice ed è dotato di un terminale dedicato in fase di avvio.

Ogni vde_switch funziona come uno switch reale; ha una socket attraverso la quale una macchina (vde_switch o host UML) vi si connette e attraverso questo comunica con tutti gli altri host connessi (dopo una corretta configurazione dei dispositivi di rete).

Per comunicare con lo switch esiste invece una console analoga alla uml_console di una macchina UML, eseguibile col comando *vdeterm*, la quale verrà però approfondita più in dettaglio in seguito.

1.3.1 Avvio e comandi

Per il lancio dello switch viene usato il comando *vde_switch*. Non sono necessari particolari attributi se non il nome che si vuole assegnare allo switch, che rappresenterà anche la directory contenente i file temporanei, nonché la sua socket (-s sta appunto per socket):

vde_switch -s <socket_dir>

Oltre a questa configurazione standard è ovviamente possibile aggiungere parametri per modificare il funzionamento e le caratteristiche dello switch.

Mi appresto ad elencare i più importanti, nonché quelli utilizzati nella realizzazione del framework:

- *-n* numero di porte (di deafault ha 32 porte).
- *-d* serve per avviare lo switch in modalità deamon, consente di nascondere il terminale.
- *-x* avvia lo switch in modalità hub, il che consente di ridirigere il traffico ad ogni macchina connessa anche se i dati sono destinati solo ad una di queste.
- *--mgmt* specifica il nome della console assegnata allo switch.

Esempio di una possibile configurazione:

vde_switch -d -n 4 -s /tmp/switch1 –mgmt /tmp/s1_term.mgmt

Attraverso il terminale o la console dello switch è possibile eseguire comandi per il controllo e la configurazione dello switch, i quali verranno approfonditi nel prossimo capitolo.

1.3.2 VDE console

La comunicazione tra l'Host e il vde_switch è resa possibile grazie a una console dedicata, eseguibile, come accennato in precedenza, tramite il comando *vdeterm*.

Il ruolo è del tutto analogo a quello della uml_mconsole. Al contrario di quest'ultima però non supporta il lancio dei comandi aggiunti come parametri. L'unico modo per eseguirli è quindi quello di entrare nella command line interna tramite *vdeterm* ed inserire successivamente i comandi da lanciare (Figura 5).

```
$ vdeterm /tmp/myvde.mgmt
VDE switch V.2.3.1
(C) Virtual Square Team (coord. R. Davoli) 2005,2006,2007 - GPLv2

vde[/tmp/muvde.mgmt]:
```

Figura 5: vdeterm

Questo aspetto è risultato essere un problema ai fini del progetto poiché complicava notevolmente l'invio dei comandi allo switch. Il tutto è stato quindi risolto adottando il comando *unixcmd* che si prende carico di eseguire su una **socket** il **comando** scritto su un determinato **file**. La sintassi è la seguente:

unixcmd -s /tmp/s1_term.mgmt -f /etc/vde2/vdecmd showlist

In pratica quello che succede è che viene aperto in backgound il processo che apre il vdeterm dello switch s1 ed esegue il comando showlist, il cui output viene stampato secondo gli standard definiti nel file di configurazione vdecmd.

Sono diversi i comandi che è possibile inviare al vde_switch tramite la sua console. *Shutdown* e *showinfo,* sono gli unici che sono stati usati in questo progetto, ma per completezza mi appresto ad elencare i comandi principali.

- *fstp* è in realtà un gruppo di comandi (ftsp/print, fstp/setfstp 0/1 ecc.) per la gestione del protocollo Fast Spanning Tree all'interno dello Switch.
- *logout* permette di uscire dal vdeterm.
- *port* insieme di comandi che si occupa della configurazione delle porte dello switch.
- *showinfo* stampa una serie di informazioni generali riguardanti il dispositivo, in questo contesto è stato utilizzato unicamente per controllarne lo stato in modo tale da poter gestire eventuali crash.
- *shutdown* causa lo spegnimento pulito dello switch.
- *vlan* insieme di comandi per la creazione e la gestione di reti vlan.

2 SimNet

2.1 Panoramica

SimNet è un framework che sfrutta lo User Mode Linux e il VDE switch per la creazione di reti simulate. È stato scritto in Python utilizzando PyCharm come

ambiente di sviluppo. Tale scelta si basa sulla necessità di usare un linguaggio moderno e semplice per facilitare la futura manutenzione e aggiornamento del programma e di usufruire della numerosa quantità di librerie che Python mette a disposizione. In particolare, si sottolinea l'uso delle librerie *threading* e *os*, impiegate rispettivamente per il controllo dei dispositivi tramite lancio di thread e per chiamate al sistema operativo, necessarie al lancio e alla configurazione delle macchine UML e VDE switch.

Il framework è pensato per funzionare su Linux in quanto, lo UML così come il VDE switch non sono supportati da altri sistemi operativi ed il loro utilizzo richiede una conoscenza almeno minima dei comandi per configurazione dei due dispositivi di rete.

Come già accennato il mio lavoro è stato incentrato sull'implementazione del lato back-end del software e per tanto mi limiterò a descrivere la struttura e il funzionamento del medesimo.

L'assenza di interfaccia grafica è chiaramente molto limitante. La specifica delle caratteristiche dei dispositivi e la configurazione della rete sono possibili esclusivamente mettendo mano al codice. Allo stesso modo risulta scomodo cambiare la struttura del network in seguito all'avvio, così come modificare le specifiche delle macchine.

Insomma, il software che mi presto a descrivere si presenta come lo scheletro di SimNet, incaricato di interfacciarsi con il sistema operativo ma incapace di fornire funzionalità di tipo user friendly delle quali si prenderà carico il lato front-end.

Sono inoltre già presenti nel codice e in fase di implementazione diversi metodi per la gestione dei file di salvataggio e il caricamento di una rete pre-salvata. Di questo e di altre migliorie e funzionalità aggiuntive si parlerà nell'ultimo capitolo, riguardante le mancanze del software e del futuro del progetto.

Fatte le doverose premesse si sottolinea che non è stata data priorità alla gestione dei file di salvataggio, né a un tipo di utilizzo user friendly. Piuttosto, presa visione delle problematiche emerse dall'analisi di Marionnet, si è voluto dare importanza alla gestione e prevenzione di crash improvvisi, a un corretto avvio della rete e senza dubbio a un ammodernamento generale del codice.

NOTA: Nei paragrafi successivi con “SimNet” si intenderà per comodità esclusivamente il lato back-end dell’intero software, soggetto di questo elaborato.

2.2 Struttura

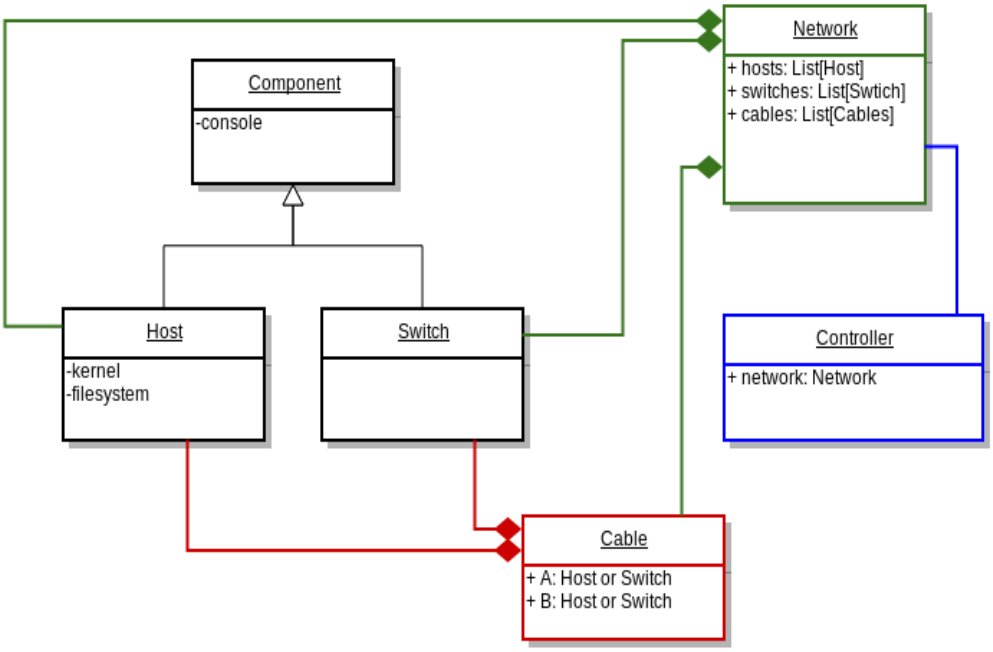


Figura 6: Class Diagram di SimNet

SimNet è costituito da una serie di classi, ognuna delle quali va ad identificare un oggetto della rete.

Nella Figura 6 è riportato il diagramma del framework in maniera sintetica, senza metodi e solo con gli attributi più rilevanti. Ogni classe verrà trattata in modo più approfondito nei diversi paragrafi, mentre in questo ci si limita a introdurre brevemente gli elementi costituenti di SimNet.

La classe principale *Network* rappresenta la rete ed è pertanto composta da tre liste contenenti ognuna una tipologia di elemento costitutivo di tale rete. Una composta da oggetti di tipo *Host,* ovvero macchine UML, una oggetti di tipo *Switch,* rappresentanti i VDE switch, e l'altra contenente oggetti *Cable*. Quest'ultima rappresenta di fatto le connessioni fisiche tra i due dispositivi ed è costituita infatti, da una coppia di valori rappresentante tali elementi. I due componenti, uno all'estremo A e l'altro all'estremo B, possono entrambi definiti indistintamente *Host* o *Switch*.

Le due classi rappresentanti i dispositivi sono entrambe figlie della classe *Component*. Tale scelta è stata presa in fase di sviluppo in quanto si è notato che le due classi presentano attributi e metodi comuni (*name*, *label*, *console*, *run* ecc..),

che risulta quindi fare ereditare dalla classe padre per una maggior chiarezza logica e pulizia del codice.

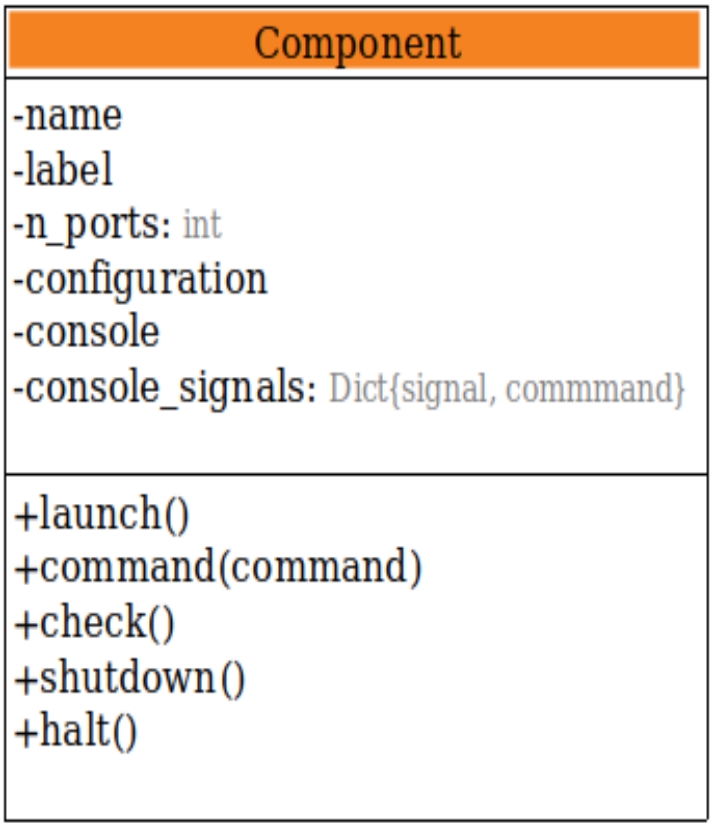
È inoltre presente la classe *Controller* che non rappresenta un componente del network ma si occupa di controllare l'avvio del network e tiene costantemente monitorato lo stato delle macchine, gestendone il crash o la chiusura accidentale.

Sono infine presenti variabili globali per la definizione dei filesystem, kernel ed eventuali file di salvataggio e un metodo indipendente per il controllo della correttezza di varie stringhe. Anche questo insieme i attributi e metodi verrà trattato in modo più approfondito nei prossimi paragrafi.

Due premesse prima di proseguire, la prima più tecnica mentre la seconda riguarda l'organizzazione del documento.

- Gli attributi il cui tipo non viene riportato in figura sono stringhe.
- Il contesto di utilizzo dei vari metodi e attributi che verranno introdotti, in particolare di Network e Controller, sarà approfondito nel prossimo capitolo riguardante il funzionamento del framework.

2.2.1 Component



La prima classe che andiamo ad analizzare è quella che definisce la struttura dei due dispositivi di rete *Host* e *Switch*, che, in quanto figli, ne ereditano tutti gli attributi e i metodi. Tale scelta si basa sul fatto che in alcuni punti del codice è comodo trattare le due classi in maniera analoga come si vedrà successivamente nel capitolo sul funzionamento.

Gli attributi di *Component* non sono tutti essenziali al livello funzionale in quanto verranno sovrascritti dalle classi figlie, ma sono stati mantenuti per una maggior chiarezza logica.

Oltre al banale *name*, troviamo *label*, ovvero un'etichetta, servirà a creare gruppi di dispositivi all'interno del network nell'ottica di una futura integrazione con l'interfaccia grafica. *n_ports* invece, definisce il numero di interfacce di rete assegnate al dispositivo, mentre *configuration* rappresenta la stringa di configurazione per il suo avvio. Quest'ultima è costruita secondo gli standard dello UML (vedi paragrafo 1.2.2) o del VDE (vedi paragrafo 1.3.1) a seconda che si tratti di lanciare rispettivamente un Host o uno Switch.

Gli ultimi due attributi riguardano la console della macchina: *console,* ovvero il nome della socket e *console_signals,* forse l'attributo più interessante, è un dictionary contenente i possibili comandi inviabili appunto al dispositivo.

Le sue chiavi (i *segnali*) sono mantenute uguali nell'*Host* come nello *Switch*, mentre i valori corrispondenti (i *comandi*) cambiano in base all'implementazione. Questo permette alle due console di interpretare un segnale nello stesso modo anche possedendo due sintassi di comandi differenti.

```
Component:  self.console_signals = {'stop': '',
                                    'halt': '',
                                    'check': ''}
```

Figura 7: console_signals, attributo di Component

Tale attributo gioca un ruolo fondamentale nel funzionamento del metodo *command()* che verrà approfondito più nello specifico al Paragrafo 2.3.2 sulla gestione dell'invio dei comandi. Il concetto alla base di questo metodo è comune in entrambi i dispositivi, nonostante venga implementato diversamente nelle due classi. Fondamentalmente si prende carico di comunicare con la console del dispositivo inviandole, attraverso una chiamata al sistema operativo, il *command* che gli viene passato come parametro (uno di quelli contenuti in *console_signals*).

Un dettaglio strutturale che contraddistingue questo metodo è che non viene mai chiamato da classi esterne ma viene utilizzato esclusivamente dai metodi della classe stessa. Questo circoscrive le chiamate al sistema operativo e alla console all'interno delle classi dei due dispositivi. Le classi esterne a questo punto non devono conoscere la sintassi dei comandi da inviare ma gli viene resa disponibile (attraverso *console_signals*) una sintassi di comunicazione comune ed intuitiva. I metodi *check(), shutdown()* e *halt()*, ad esempio, hanno il compito rispettivamente di inviare un segnale di controllo, di spegnimento e di chiusura forzata al dispositivo a cui fanno riferimento, e per farlo chiamano il metodo *command()* passandogli come parametro il comando corrispondente.

Il metodo *launch()* consiste invece in un'unica chiamata al sistema operativo tramite il comando *system* della libreria Python *os*. Questa chiamata consiste

nell'avvio del terminale Unix attraverso il comando *xterm* seguito dal parametro *-e* e da *configuration* stringa di lancio del Component.

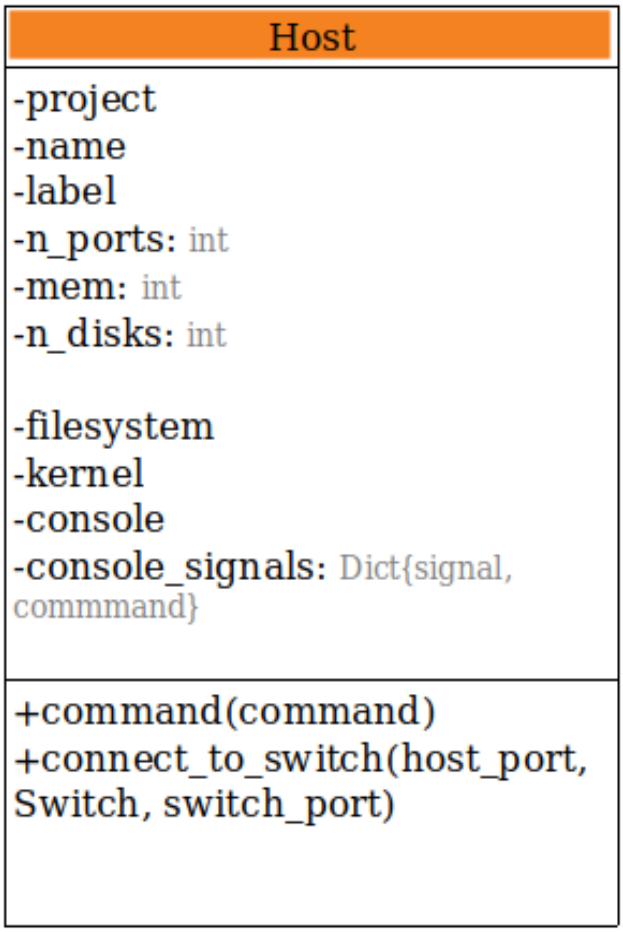
xterm -e "stringa" avvia un terminale ed esegue la stringa al suo interno, così facendo ogni terminale lanciato esegue l'avvio di una macchina UML, nel caso il dispositivo sia uno Host, o di un VDE switch, nel caso si tratti di uno Switch, secondo le specifiche descritte in *configuration.*

```
def launch(self):
    os.system("xterm -T " + self.name + " -e \"" + self.configuration + "\"")
```

Figura 8: launch, metodo del Component

Si noti l'aggiunta del parametro -T che ha la funzione, puramente estetica, di assegnare un titolo al terminale, in questo caso il nome del Component.

2.2.2 Host



La classe Host rappresenta la macchina UML. Il primo gruppo di attributi, fino a n_disk, viene definito dall'utente. Tralasciando gli attributi già descritti nel paragrafo precedente, in questo gruppo troviamo *mem* e *n_disk,* ovvero rispettivamente la quantità di memoria RAM e il numero di dischi che si vogliono assegnare al dispositivo.

I successivi attributi sono assegnati di default (*filesystem*, *kernel* e *console_signal*) o dedotti in base alle specifiche assegnate dall'utente.

In futuro sia il filesystem che il kernel su cui far partire le macchine potranno essere scelti in fase di configurazione, ma per adesso si è deciso per comodità e di assegnarli di default.

console_signals è così composto:

```
Host:       self.console_signals['stop'] = 'cad'
            self.console_signals['halt'] = 'halt'
            self.console_signals['check'] = 'version > /dev/null 2>&1'
```

Figura 9: console_signal, attributo di Host

Si noti che è stata effettuata una redirezione dello standard Output e standard error su /dev/null per il comando *version* poiché, come già accennato in precedenza non interessa il risultato di tale comando quanto la presenza o meno di una risposta da parte della uml_mconsole per controllare lo stato del dispositivo.

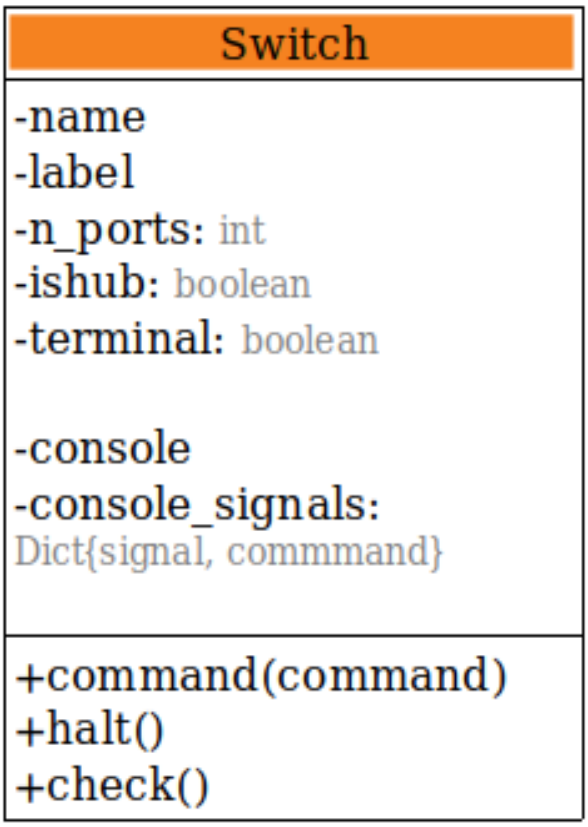
Per quanto riguarda i metodi, *command()* è già stato introdotto nel paragrafo precedente e non fa altro che inviare il comando *command* alla uml_mconsole secondo la sintassi descritta al paragrafo 1.2.2, ritornando True in caso di successo e False in caso contrario.

connect_to_switch() serve a connettere la porta *myport* dell'*host* alla porta *hisport* dello switch. Per farlo non gli basta che chiamare il metodo *command()* passandogli come parametro la giusta sintassi del comando *config* (si veda Figura 10).

```
def command(self, command):
    if os.system("uml_mconsole " + self.console + " " + command) == 0:
        return True
    return False

def connect_to_switch(self, myPort, component, hisPort):
    self.command('config eth' + str(myPort) + '=vde,/tmp/' + component + ',,' + str(hisPort))
```

Figura 10: meccanismo di connessione a uno switch da parte dell'host

2.2.3 Switch



La classe Switch è quella che rappresenta appunto il VDE_switch ed analogamente all'Host, il primo gruppo di parametri è quello passato in fare di costruzione dell'oggetto.

Tralasciando i primi 3 attributi già descritti al paragrafo sulla classe *Component* (2.2.1), troviamo due flag: *ishub* e *terminal.* Il primo indica al dispositivo se lavorare come Hub o come Switch. Se settato su *True* viene aggiunta alla stringa di configurazione (*configuration)* il parametro *-x*, che svolge appunto questa funzione. *terminal*, in maniera analoga, se uguale a False nasconde il terminale dello switch tramite l'aggiunta di *-d* (deamon).

Gli ultimi due parametri riguardano la console dello switch e sono del tutto analoghi a quelli presenti anche nell'Host ed analizzati in precedenza. *console_signals* questa volta è semplicemente composto dal comando *shutdown* corrispondente al segnale omonimo. Questo perché non esiste un comando della vdeterm per la chiusura forzata (*halt* nella *uml_mconsole*) e perché sono sorti dei

problemi con l'invio del segnale *check* (*version* nella *uml_mconsole*) alla vdeterm di uno Switch non più attivo (vedi Paragrafo 3.2). Questa è anche la ragione del perché i metodi *halt()* e *check()* vengano sovrascritti. Nello specifico il primo chiude i processi che fanno funzionare lo switch, mentre l'altro controlla l'esistenza della directory corrispondente alla socket dello switch.

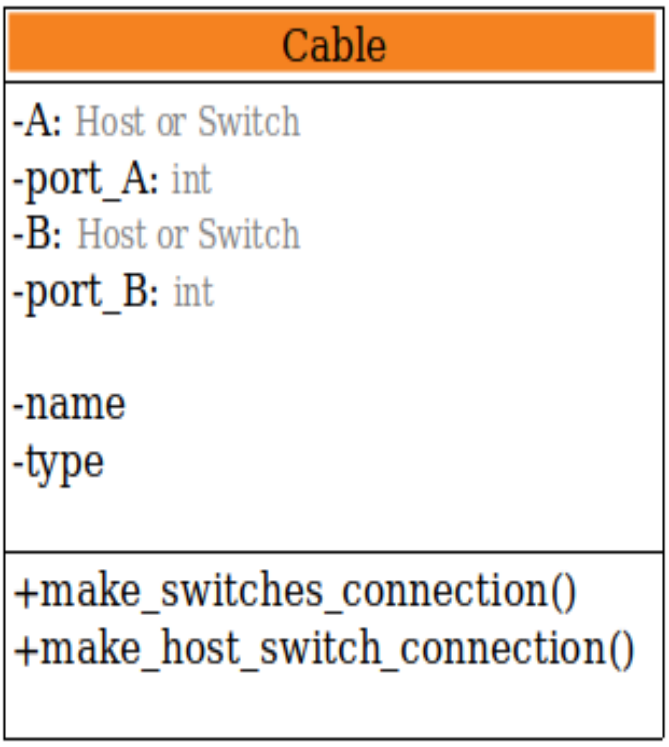
```
def halt(self):
    os.system("pgrep -f /tmp/" + self.name + " | xargs kill -TERM")

def check(self):
    return os.path.exists('/tmp/' + self.console + '.mgmt')
```

Figura 11: halt() e check(), metodi della classe Switch

Come si può notare entrambi i metodi sono entrambi composti da una semplice chiamata al sistema operativo. Mi soffermo in particolare sul comando chiamato dal metodo halt(). Si tratta di una pipeline nella quale il primo comando *pgrep* trova tutti i processi su cui gira lo switch e li passa al comando *xargs*, il cui compito è quello di eseguire il comando che lo segue, in questo caso *kill*, una volta per ogni parametro letto su standard input. In pratica ogni processo trovato viene "killato", causando la chiusura forzata dello Switch.

Per quanto riguarda gli altri metodi, launch() viene ereditato dalla classe Component, mentre command() viene sovrascritto, analogamente a come accade nella classe Host. In particolare la chiamata al sistema operativo del metodo command() è scritta secondo la sintassi per comunicare con il vdeterm descritta nel Paragrafo 1.3.2.

2.2.4 Cable



La classe Cable rappresenta il cavo di collegamento tra un Component e l'altro. I primi quattro parametri, riguardanti i dispositivi coinvolti e le relative porte sono assegnati dall'utente, mentre il nome (*name*) e il tipo di cavo (*type*) vengono dedotti. In particolare, il cavo è di tipo *cross* se i due dispositivi appartengono alla stessa classe, *straight* in caso contrario.

Tornando al primo gruppo di parametri si è scelto che nel caso le due macchine siano di tipo diverso (cavo *straight*) all'estremo A verrà sempre attaccato l'Host. Questo, come si vedrà tra poco, facilita la gestione dei collegamenti.

I metodi sono due, *make_switches_connection()* e *make_host_switch_connection()* servono rispettivamente per instaurare una connessione Switch to Switch e una Host to Switch. Purtroppo, quella Host to Host non è ancora stato possibile realizzarla perché non è stato trovato il comando da inviare alla uml_mconsole per realizzare tale collegamento.

Neanche per la connessione tra due switch esiste in realtà un vero e proprio comando per il vdeterm, ma la documentazione ufficiale offre comunque una

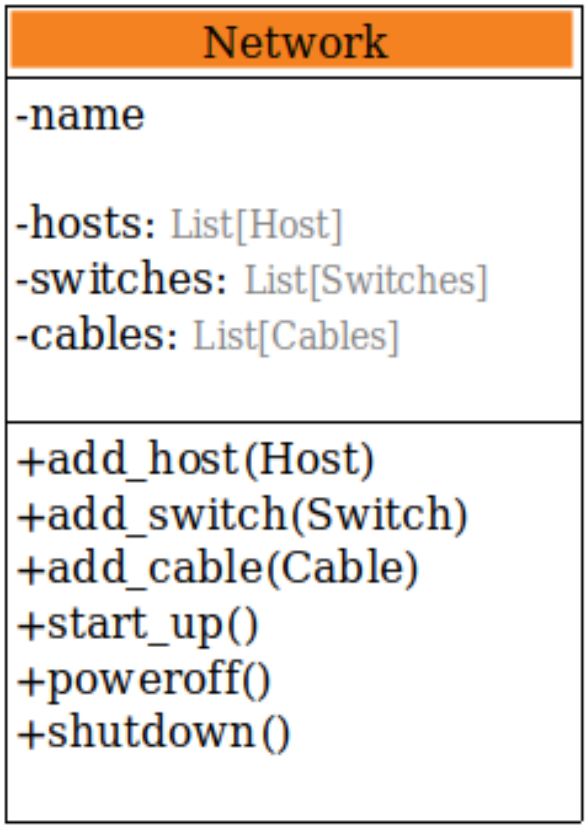
soluzione, ovvero la creazione di una pipe bidirezionale tra i due dispositivi attraverso il comando *dpipe*. *make_switch_connection()* non fa altro che chiamare questo comando uguagliando le *vde_plug* dei due switch coinvolti, che in sostanza rappresentano le prese ethernet simulate dei due dispositivi (Figura 11).

```
def make_switches_connection(self):
    os.system("dpipe vde_plug /tmp/" + self.A.name + " = vde_plug /tmp/" + self.B.name)
```

Figura 12: make_switch_connection, metodo della classe Switch

make_host_switch_connection() invece chiama semplicemente il metodo *connect_to_switch()* della classe Host passandogli i parametri opportuni. Tale metodo è stato precedentemente introdotto al paragrafo 2.2.2

2.2.5 Network



La classe Network rappresenta la rete e contiene quindi tutti gli elementi che la compongono.

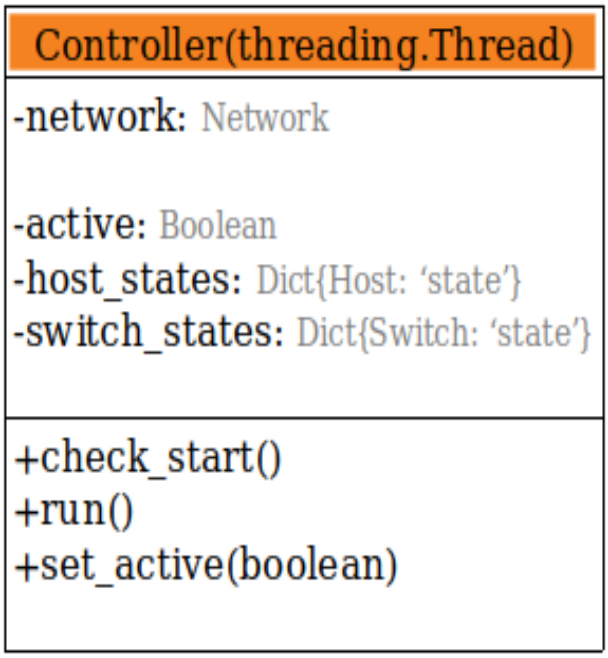
Troviamo dunque, oltre l'attributo *name* che identifica il nome del progetto, una lista di Host, una di Switch e una di Cable.

Gli oggetti vengono intuitivamente aggiunti alla rete tramite i metodi *add_host()*, *add_switch()* e *add_cable()*, ai quali viene passato l'oggetto da annettere alla rete, che tramite un semplice *append* viene inserito nella lista corrispondente.

Gli ultimi tre metodi *start_up()*, *poweroff()* e *shutdown()* si occupano rispettivamente dell'avvio, dello spegnimento e della chiusura forzata della rete. Tutti e tre sono costituiti per lo più da chiamate a metodi delle classi relative agli

oggetti della rete e il loro funzionamento verrà discusso in modo approfondito nel prossimo capito riguardante il funzionamento del programma.

2.2.6 Controller



La classe Controller non fa propriamente parte della rete, ma attua operazioni di controllo su di essa. Per svolgere tale funzione ha bisogno di agire in background ed è per questo che eredita la classe *Thread* della libreria *threading*, rendendosi di fatto un processo assestante.

In fase di costruzione gli viene passato come parametro un oggetto Network, ovvero la rete su cui effettuare i controlli. Per farlo vengono estratti gli Host e gli Switch che la compongono e inseriti in due dictionary le cui coppie chiave valore sono definite dal binomio *dispositivo : stato*. Le due strutture permettono quindi di monitorare costantemente la condizione del dispositivo che può essere *opened* o *closed*. Dopodiché l'attributo *active,* settabile attraverso il metodo *set_active(),* permette al Controller di capire quando avviare e bloccare il controllo sulla rete. Troviamo poi *check_start(),* il quale si prende carico di controllare il corretto avvio dei dispositivi tramite semplici chiamate al metodo *check()* delle relative classi.

Infine, il metodo *run()* merita una nota di riguardo in quanto definisce di fatto il lavoro che il thread deve compiere una volta avviato dall'esterno tramite il lancio del comando thread.start(). In pratica, secondo gli standard definiti dalla libreria *threading* chiamando *Controller.start()* viene eseguito il metodo *run()*.

2.3 Funzionamento

2.3.1 Avvio

L'avvio è di certo la fase più importante e anche più delicata in quanto deve permettere la partenza simultanea di svariate macchine collegate e configurate in maniera corretta.

Nonostante la complessità dell'operazione l'avvio viene eseguito da un unico e semplice metodo: start_up() della classe Network.

```
def start_up(self):

    for c in self.hosts + self.switches:
        t = threading.Thread(target=c.launch, args=())
        t.daemon = True
        t.start()
        time.sleep(0.2)

    for c in self.cables:
        if c.type == 'straight':
            c.make_host_switch_connection()
        else:
            if type(c.A) == Host:
                print('collegamento Host to Host impossibile, mettici uno switch in mezzo :-)')
            else:
                t = threading.Thread(target=c.make_switches_connection, args=())
                t.daemon = True
                t.start()
```

Figura 13: Network.start_up(), avvio della rete

La semplicità di tale metodo è resa possibile dal corretto incapsulamento delle chiamate al sistema operativo all'interno dei metodi delle classi.

Analizzando il codice salta subito all'occhio il massiccio uso della libreria threading. Viene infatti lanciato un thread in modalità *deamon* per ogni Component che si vuole attivare. Tale modalità permette di avviare un qualsiasi numero di thread in parallelo, senza bloccare quello padre e consente inoltre di gestire indipendentemente ognuno di questi. È importante distinguere però questo tipo di

lancio di un thread da quello precedentemente descritto al paragrafo sulla classe *Controller* (2.2.6).

In questo caso infatti non troviamo più il meccanismo start()-run() definito in precedenza, non abbiamo a che fare con una classe figlia di *threading.Thread* ma con la classe *Thread* stessa. Si crea infatti un oggetto *Thread*, al quale viene passato *component.start()* che imposta di fatto come metodo *run()*, e lo si fa partire tramite *thread.start()*.

Per quanto riguarda invece la funzione *sleep()* della libreria *time*, il suo ruolo all'interno del codice verrà discusso nel capitolo seguente.

Il metodo *start_up()* prosegue poi con l'attivazione dei collegamenti tra le macchine. Per ogni Cable viene chiamato uno dei suoi due metodi (entrambi descritti al Paragrafo 2.2.4): quello per la connessione Host to Switch nel caso di cavo *straight* e quello per la connessione Switch to Switch nel caso di cavo *cross*, escludendo quella Host to Host non ancora implementata.

Si noti che è necessario lanciare un ulteriore thread per la creazione del collegamento tra i due Switch poiché, come già detto al Paragrafo 2.2.3, questo processo include l'apertura di una doppia pipe. Quest'ultima, se non gestita in modo indipendente, bloccherebbe inevitabilmente l'esecuzione del programma.

2.3.2 Gestione dell'invio dei comandi

Nei paragrafi relativi alla descrizione delle classi si è spesso parlato del metodo *command()* e in generale dell'invio di comandi alle console dei vari dispositivi. Vediamo ora come effettivamente viene realizzata questa comunicazione.

L'implementazione di tale comando nelle classi delle due macchine è diversa ma concettualmente identica.

```
def command(self, command):
    if os.system("uml_mconsole " + self.console + " " + command) == 0:
        return True
    return False
```

Figura 14: implementazione metodo command() nella classe Host

```
def command(self, command):
    # 65280 exit status comando unixcmd
    if os.system("unixcmd -s /tmp/" + self.console + ".mgmt -f /etc/vde2/vdecmd " + command) == 65280:
        return True
    return False
```

Figura 15: implementazione metodo command() nella classe Switch

Come si può notare, entrambe le implementazioni seguono la stessa logica: viene lanciata una chiamata al sistema operativo per eseguire un comando sulla socket del dispositivo utilizzando le sintassi relative alle due console descritte nel primo capitolo. L'unica particolarità che salta all'occhio è che l'exit status del comando *unixcmd* in caso di successo non è il classico 0 ma 65280. Peraltro, si è notato che anche in caso di insuccesso quest'ultimo risulta essere l'exit status del comando. Questo è infatti il motivo per cui è stato necessario sovrascriver il metodo

check() nello switch, poiché attraverso il metodo *command()* non è possibile stabilire se la comunicazione con la console dello switch sia andata a buon fine o meno.

Il metodo *command()* come già accennato viene sfruttato internamente alle due classi ma mai chiamato dall'esterno. Entrambe le classi rendono disponibili dei metodi che chiamano tale funzione passandoli il segnale opportuno da mandare alla console.

Un aspetto importante di come è stato gestito l'invio dei comandi è il fatto che spesso gli oggetti possono essere trattati come semplici Component, indipendentemente dal fatto che essi siano Host o Switch. Questo è reso possibile in particolare dall'attributo *console_signals* introdotto al Paragrafo 2.2.1. Lo schema riportato di seguito è esemplificativo di come interagiscono tra loro *command()* e *console_signals*.

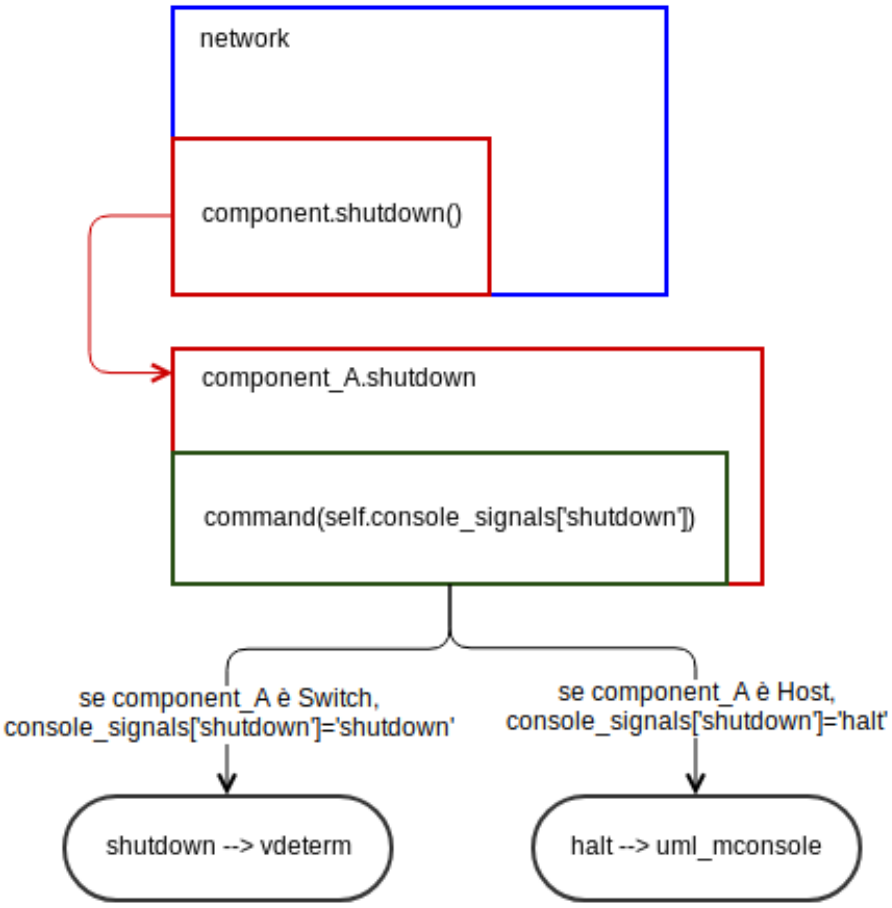


Figura 16: gestione dell'invio dei comandi alle console dei dispositivi

Nell'esempio riportato in Figura 15 dalla classe network si vuole spegnere un generico dispositivo. Il segnale di spegnimento viene eseguito dal comando *halt* nella uml_mconsole e da *shutdown* nel vdeterm. Se si tratta il dispositivo come un Component indipendentemente che questo sia un Host o uno Switch per spegnerlo basterà inviare il generico segnale di shutdown, che verrà poi interpretato in modo differente a seconda del tipo di dispositivo.

Purtroppo, l'attributo *console_signals* svolge bene la sua funzione di generalizzazione solo per l'invio del segnale di shutdown. Questo è dovuto al fatto che sono sorti problemi nella gestione dell'invio dei segnali di controllo (*check*) e di chiusura forzata (*halt*) al vdeterm, comportando la sovrascrittura dei due metodi

nella classe Switch. Tale fatto è già stato affrontato al Paragrafo 2.2.3 relativo alla classe Switch e verrà approfondito nel capitolo successivo.

2.3.3 Gestione e controllo degli errori

Attraverso una piccola attività di testing si sono riscontrate tre situazioni a rischio di errori: l'avvio di una macchina, la chiusura volontaria o meno di un dispositivo e il settaggio di alcune stringhe importanti.

Per controllare che all'avvio ogni elemento della rete parta correttamente si utilizza il metodo *check_start()* della classe Controller.

```
def check_start(self):
    time.sleep(1)
    for h in self.net.hosts:
        if not h.check():
            print(h.name + ' non è partito correttamente')
            return False
        else:
            self.hosts_states[h] = 'opened'
            print(h.name + " è partito correttamente")
    for s in self.net.switches:
        if not s.check():
            print(s.name + ' non è partito correttamente')
            return False
        else:
            self.switches_states[s] = 'opened'
            print(s.name + " è partito correttamente")
    return True
```

Figura 17: check_start metodo della classe Controller

Come si può notare con l'ausilio del metodo *check()* della classe Component viene verificato lo stato dei dispositivi che, se attivi vengono segnati come *opened*, mentre *closed* in caso contrario. Se una macchina, per un qualche motivo, non è partita correttamente questo viene segnalato all'utente.

check_start() viene chiamato subito dopo l'avvio della rete, per questo aspetta un secondo (*time.sleep(1)*) prima di partire, in modo da lasciare il tempo ai dispositivi di creare i file necessari al loro avvio. In caso contrario, si potrebbero creare dei conflitti tra i vari processi. Si noti che questo tipo di approccio è stato spesso adottato (come si può vedere anche nel paragrafo precedente) poiché lavorando con i thread, può succedere che si incappi in errori runtime dovuti ad un non corretto avvicendamento dei processi. Per evitare questo tipo di conflitti si è scelto per comodità di adottare il metodo *sleep()* che assicurare una corretta

successione dei thread. Tale scelta implementativa compresa di settaggio del tempo di attesa è stata trovata empiricamente.

Per quanto riguarda il controllo dei dispositivi durante l'utilizzo della rete, si è scelto di sfruttare un processo in background che monitori costantemente il loro stato tramite chiamate al metodo *check()* del dispositivo corrispondente. Tale processo, come già accennato al Paragrafo 2.2.6, non è altro che la classe Controller stessa. Essendo quest'ultima figlia della classe Thread della libreria threading ne eredita tutte le caratteristiche, tra cui anche il metodo run(). Quest'ultimo, analogamente a check_start() monitora i dispositivi e aggiorna i due dictionary *host_states* e *switch_states* nel caso si riscontri un cambiamento di stato.

Questo in pratica è quello che avviene: una volta avviato il network e controllato che sia partito correttamente, tramite il comando start() viene lanciato un thread di tipo Controller sul quale viene eseguito il metodo run().

Sono stati inoltre introdotti una serie di controlli sulle stringhe relative ai componenti delle macchine e in generale riguardanti la rete. Tali controlli vengono effettuati da una serie di semplici funzioni i cui compiti sono ad esempio controllare la presenza di caratteri speciali, di stringhe nulle o che il nome di un dispositivo non si uguale a quello di uno già esistente.

2.3.4 Chiusura

La chiusura del network è un procedimento piuttosto semplice e può avvenire in modo pulito tramite *network.shutdown()* o in maniera forzata con *network.poweroff()*. Quest'ultima opzione chiaramente è da tenere in considerazione solo in caso di crash improvvisi o di mancata risposta da parte del network; si tende ad usare quando possibile la chiusura pulita.

```
def shutdown(self):
    for c in self.hosts + self.switches:
        if c.check():
            print('spengo in modo pulito' + c.name)
            c.shutdown()

def poweroff(self):
    for c in self.hosts + self.switches:
        if c.check:
            print('chiudo forzatamente ' + c.name)
            c.halt()
```

Figura 18: shutdown() e poweroff() due metodi della classe Network

Come si evince dall'immagine sopra riportata i due metodi funzionano in maniera del tutto analoga. Entrambi sfruttano il polimorfismo della classe Component, chiamando il metodo *check()* per controllare che il dispositivo non sia già stato chiuso in precedenza, indipendentemente che questo sia di tipo Host o Switch. Successivamente, in maniera analoga vengono chiamati i metodi *shutdown()*, nel caso di uno spegnimento pulito, o *halt()* nel caso di chiusura forzata, i quali sono stati precedentemente trattati dei paragrafi relativi ai due dispositivi.

Le procedure di spegnimento, per la loro semplicità e chiarezza, sono quelle che maggiormente fanno apprezzare la struttura padre-figlio che intercorre nelle classi Component, Host e Switch.

3. Conclusioni

3.1 Obbiettivi raggiunti

Presa visione dei problemi emersi dall'analisi di Marionnet, i due principali obbietti con i quali si è iniziato lo sviluppo di SimNet sono stati l'ammodernamento del codice e una migliore gestione degli errori. L'utilizzo di Python come linguaggio di programmazione ha sicuramente aiutato in questo senso, essendo esso ampiamente usato al giorno d'oggi e di facile comprensione.

Inoltre, come già spiegato in precedenza si è cercato di impacchettare azioni di basso livello come chiamate al sistema operativo dentro ai metodi dei dispositivi, in modo da semplificare la struttura e la comprensione del funzionamento del software. In sostanza si è preferito concentrare la complessità del codice all'interno delle classi a favore di uno snellimento del business-code. In più il codice è stato commentato in maniera standard così da facilitarne la futura manutenzione e aggiornamento[2].

È stata posta una particolare attenzione anche sul corretto avvio della rete e sulla gestione dei crash dei dispositivi, costruendo metodi per il monitoraggio della rete, capaci di segnalare e gestire la presenza di un crash.

Complessivamente si può quindi dire che il software sia caratterizzato da un alta resilienza e manutenibilità, così come ci si era prefissati.

[2]I commenti non sono presenti nel codice mostrato in questo documento, in quanto aggiunti successivamente

3.2 Problemi da risolvere e possibili migliorie

Chiaramente il codice non è totalmente privo di imperfezioni e non sono pochi i miglioramenti che si possono effettuare.

Innanzitutto, si è notato che il massiccio uso di thread, come prevedibile, incrementa notevolmente il lavoro della CPU. Con tutta probabilità il maggior responsabile è il processo che costantemente monitora lo stato dei dispositivi. Per contrastare il problema potrebbe essere utile capire come settare correttamente il tempo di riposo del processore attraverso la funzione *time.sleep()*.

Riguardo al controllo dei dispositivi si è inoltre notato, attraverso un considerevole numero di test, che può capitare (diciamo una volta su 20) che il processo Controller non riconosca la chiusura di un dispositivo. Anche questo è probabilmente legato al corretto settaggio della funzione *sleep()* per cui se l'utente chiude un terminale in un determinato momento, il Controller non se ne accorge.

In generale la funzione della libreria time viene spesso usata per la gestione dei thread impostando il tempo di riposo della CPU in modo empirico. Sarebbe invece più corretto svolgere un'analisi più approfondita dei vari processi, per raggiungere una maggior sicurezza ed efficienza del software.

Altro aspetto da migliorare è la comunicazione con il vdeterm. Come accennato dei capitoli precedenti non è possibile lanciare comandi alla socket senza entrare nel terminale dello switch. Tale problema è stato aggirato con l'utilizzo del comando *unixcmd*, che però è risultato essere piuttosto inutile per la rilevazione di errori poiché il suo exit status, come accennato al Paragrafo 2.3.2, è costantemente 65280. La risoluzione di tale problema tramite l'adozione di una nuova modalità di comunicazione semplificherebbe la comprensione del codice e migliorerebbe la comunicazione con la console.

Un altro tema con ampio margine di miglioramento è quello dei collegamenti tra dispositivi. Oltre all'impossibilità di connettere due Host tra di loro, è risultato impossibile selezionare le porte di collegamento tra due Switch. Questo è dovuto al

fatto che il comando *dpipe* che rende possibile tale collegamento (descritto nel Paragrafo 2.2.4) non prevede la specifica delle due porte di collegamento, le quali vengono assegnate in automatico.

Per quanto riguarda le macchine UML non è stato possibile settarle come attive ma non collegato a nessun dispositivo. Per cui l'utente può indicare il numero di porte che un Host dovrà avere, ma queste non verranno attivate se non verranno collegate ad alcun dispositivo. Questo di per sé non rappresenta un vero problema di funzionamento in quanto ogni volta che l'utente proverà a collegare una qualche macchina *x* alla porta *n* (con $n < n\_ports$) del dispositivo, tale porta verrà aperta e connessa correttamente alla macchina *x*. In ogni caso rimane un problema di tipo logico/concettuale che sarebbe bene risolvere.

Se verrà ritenuta necessaria l'implementazione della connessione Host to Host e di un più preciso collegamento Switch to Switch occorrerà ricercare tali soluzioni nelle documentazioni ufficiali dello User Mode Linux e del VDE.

3.3 Futuro del progetto

Nell'ottica di un futuro utilizzo nel laboratorio universitario a grandi linee saranno tre i prossimi passi da compiere: gestire il salvataggio e il caricamento di una rete, integrare il lato back-end con l'interfaccia grafica e permettere la modifica del network in seguito all'avvio, il tutto accompagnato da una continua attività di testing.

Per quanto riguarda il primo obbiettivo due metodi per il salvataggio e il caricamento di una rete sono già in fase di implementazione. Nello specifico si è pensato di utilizzare un file json per il salvataggio della struttura della rete e conservare i file COW degli Host nel caso l'utente voglia mantenere le modifiche effettuate su tali macchine (es: modifiche al file /etc/network/interfaces).

Dopo di che, essendo SimNet di fatto un unico progetto, l'integrazione tra i due lati, back e front-end, era prevista fin dall'inizio e per questo motivo è stata condivisa e mantenuta la stessa struttura di partenza. Questa premessa, unita al fatto che entrambi i progetti software sono stati scritti in Python, faciliterà sicuramente l'integrazione tra le due interfacce. Allo stesso tempo lo sviluppo indipendente delle due parti del framework, fa sì che esse rimangano logicamente e strutturalmente separate favorendo la manutenibilità generale del software.

Con l'aggiunta di un'interfaccia grafica potrebbe poi essere comodo permettere all'utente di modificare la rete senza doverne creare una nuova. Si tratterà sostanzialmente di rendere disponibile al front-end alcuni metodi di tipo "setter" (già in fase di implementazione) per la modifica degli attributi delle classi che rappresentano la rete.

Sarà infine effettuata una doverosa attività di testing e creato uno script di installazione per poter garantire il corretto funzionamento di SimNet su altre macchine Linux.

Bibliografia

Dike Jeff, User Mode Linux, Bruce Perence's Open Source Series, Prentice Hall, 2006

Tesi di laurea di Zoboli Alex, Network Circus: un framework per la gestione di reti virualizzate mediante User Mode Linux

Sitografia

Documentazione ufficiale di Python

https://docs.python.org/3/library/

Forum consultati per dubbi generici di programmazione e sul sistema operativo Linux

https://stackoverflow.com

https://www.html.it

https://stackexchange.com/

https://askubuntu.com/

https://www.tecmint.com/

Documentazione ufficiale dello User Mode Linux

http://user-mode-linux.sourceforge.net

Documentazione ufficiale del VDE switch

http://wiki.v2.cs.unibo.it/wiki/index.php?title=Main_Page

Pagina ufficiale di Marionnet

http://www.marionnet.org/site/index.php/en/

Pagina ufficiale comando *unixcmd*

https://www.scottklement.com/unixcmd/

Sito utilizzato per la realizzazione di UML diagrams

https://go.gliffy.com