Network Circus: un framework per la gestione di reti virualizzate mediante User Mode Linux

Zoboli Alex

Indice:

(2) Framework migliorabile: creare un titolo “più comodo” e inserire il nome della macchina all'interno del terminale : root@H1
(3) Interfaccia grafica più “comoda”
(4) Più flessibilità nelle caratteristiche delle macchine

1)Marionnet

Il progetto è nato dall'idea di prendere spunto e migliorare ove possibile un software pre esistente che si occupava della creazione e gestione di reti virtuali, il progetto 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 della macchine virtuali.
Tale linguaggio è molto complicato e poco intuitivo. Nel caso di aggiornamenti o estensioni risulta molto difficile andare a metterci mano e modificare qualche stringa di codice.

Il software è caratterizzato da un'interfaccia grafica minimale (Fig. 1), attraverso la quale è possibile inserire i vari oggetti di rete, configurarli e connetterli tra loro.
L'interfaccia non è implementata come un "drag&drop" ma attraverso un inserimento degli oggetti attraverso un barra degli strumenti dove selezionare Host, Switch(hub, bridge), Cavi e Internet.
La scomodità, oltre quella di non poter semplicemente trascinare un oggetto per crearlo, è che non è possibile modificarne la disposizione a proprio piacimento ma sono possibili solo una serie di disposizioni casuali create da un algoritmo interno al software stesso.
Inoltre non è possibile modificare l'oggetto all'iterno dello schema, ma si deve andare sull'oggetto generale nel pannello di inserimento, cercare l'oggetto in questione e modifinarne gli attributi.

Presenta inoltre 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.

"Marionnet is a virtual network laboratory: it allows users to define, configure and run complex computer networks without any need for physical setup. Only a single, possibly even non-networked GNU/Linux host machine is required to simulate a whole Ethernet network complete with computers, routers, hubs, switches, cables, and more. Support is also provided for integrating the virtual network with the physical host network. "

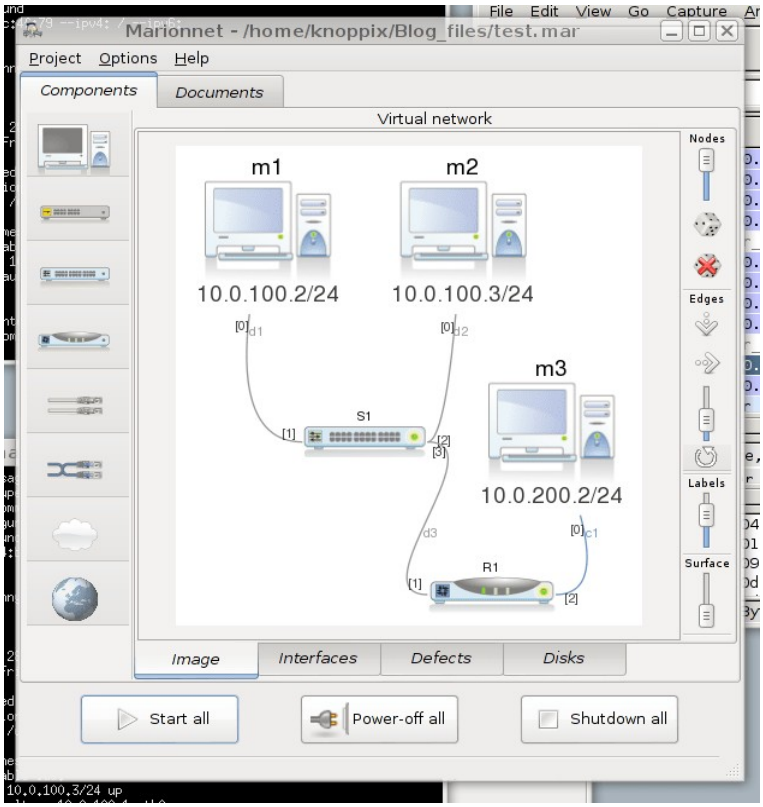

Fig.1 Marionnet GUI

2) Scopi e obiettivi del progetto

L'idea che ha portato avanti questo progetto era quella di creare un software semplice ed intuitivo, che partisse dall'idea di fondo Marionnet ma che la migliorasse e la rendesse più efficente. Implemetnare un framework che fosse in grado di creare delle reti virtuali, sempre in User Mode Linux, poterle avviare e configurare in modo rapido e allo stesso tempo il più variabili possibile. E una volta creata e configurata la rete, avviarla e testarne le diverse configurazioni con cui si potrebbe implementare e che potrebbero poi essere applicate nella realtà.

Per quanto riguarda l'interfaccia grafica, in questo progetto non verrà trattato nessun argomento o tipo di proposta di miglioramento, in quanto come detto in precedenza verrà implementato solo il framework di base che gestisce le reti, ma è comunque utile parlare anche di questo.
Durante lo sviluppo di questo framework mi sono accorto di quanto sia relativamente semplice creare e modificare gli oggetti di rete nella fase in cui la macchina è ovviamente spenta. Pensando quindi all'interfaccia grafica presente in Marionnet sono arrivato alla conclusione che sia stata implementata in modo forse troppo complicato, rendendo delle semplici operazioni (come ad esempio quella di creare uno switch che lavori come hub) molto più complesse di quello che in realtà sono.

I bug di avvio invece, sono forse il problema più grave riscontrato in Marionnet. In questo framework ho quindi cercato di gestire bene questo problema, cercando di limitare al minimo le operazioni da compiere nel caso, per qualche motivo, l'avvio della rete non vada perfettamente a buon fine.
Non dovendomi concentrare sulla parte grafica di interazione software/utente ho cercato di creare dei metodi che riuscissero a gestire bene e in modo semplice tutte le possibili configurazioni delle singole macchine, così da non incorrere in problemi in fase di avvio.

Ho inserito diversi metodi che controllano le stringhe di configurazione dei singoli elementi (interfacce di rete, dischi ...), metodi per la creazione del comando di lancio della macchina e dello switch, i quali analizzerò meglio in seguito.

3) Il framework: Network Circus

(Rif. Monty Python's Fliyng Circus)

Network Circus è un framework interamente scritto in linguaggio Python (versione 3.6) per la creazione e gestione di reti virtuali basate su User Mode Linux.

3.1)USER MODE LINUX

User Mode Linux (chiamata anche UML) è un modo per avviare Linux come "macchina" virtuale. Sostanzialmente è un kernel Linux che gira sopra ad un altro kernel Linux e solo uno di questi accede effettivamente all'hardware, mentre gli altri lavorano in user space, esattamente come gli altri programmi.
Solitamente in un sistema operativo si ha l'hardware che comunica con il kernel, che a sua volta comunica con i vari processi ad alto livello. User Mode Linux invece si posiziona a metà fra kernel dell'host e i processi, quindi il kernel UML non comunica direttamente con l'hardware ma con il kernel originale dell'Host. (Fig. 3)
UML è sia un processo Linux dell'host ma è anche un kernel per l'host virtuale.

E' molto utile per testare configurazioni di sistema, è infatti possibile lanciarlo con diversi tipi di kernel (l'architettura deve sempre essere rispettata, ovvero deve corrispondere a quella del kernel host) , con diversi file system, interfacce o dischi senza andare ad intaccare fisicamente la macchina originale, come ambiente di test per Disaster Recovery o a livello didattico.

"User Mode Linux (UML) is a virtual Linux machine that runs on Linux. Technically, UML is a port of Linux to Linux (Jeff Dike)"

E' importante sottolineare la differenza 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.
Per questo motivo, quando eseguo un kernel per lanciare la UML questo deve essere Linux e avere la stessa architettura dell'host. Questo limita in parte la gamma di virtualizzazioni possibili rispetto a quelle offerte da una macchina virtuale vera e propria ma in compenso migliora l'interazione fra host virtuale e host reale.

Come si esegue User Mode Linux
Per lanciare UML sarà necessario creare il file binario attraverso la configurazione e compilazione del kernel Linux. 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.
I primi due parametri, quelli più importanti, da passare come argomenti sono *mem* e *ubd0.* In ordine, *mem* rappresenta la memoria virtuale che la UML andrà ad allocare dinamicamente, è importante specificarlo altrimenti l'esecuzione non riesce, riportando un errore di memoria allocata errata o non disponibile.
Il secondo parametro *ubd0* rappresenta invece il disco n-simo (0 in questo caso) che deve essere inserito nella UML. Per ogni disco che vogliamo creare, dovrà essere associato un file system che verrà montato. Nel caso in cui vengano definiti più dischi (es. Ubd0=... e ubd1=...) sarà necessario specificare quale di questi sarà quello principale. Tale "precisazione" andrà effettuata inserendo il parametro *root* eguagliandolo al disco che vogliamo scegliere come root.
E' possibile scegliere con quale shell del file system si deve presentare specificandolo nel parametro opzionale *init*.
Se lo scopo di questo progetto è la gestione di reti virtuali, di conseguenza per far si che esista una

rete, ossia una serie di dispositivi connessi fra loro è necessario che ognuna di queste macchine sia fornita di almeno un'interfaccia che le consenta di conseguire questa funzione.
Questo è possibile attraverso il parametro *eth0* (*eth1, eth2 ...*) che identifica la n-esima interfaccia di rete. 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 come uno switch o un hub.
In particolare ho utilizzato la modalità switch, ovvero la connessione di un'interfaccia di rete ad uno switch tramite l'assegnazione della socket di quest'ultimo al relativo dispositivo *eth*i.

Esempi di stringhe di configurazione:

linux ubd0=Debian-Jessie-AMD64-root_fs mem=128M eth0=vde,/tmp/vde_switch1

linux ubd0=Debian-Jessie-AMD64-root_fs ubd1=rootfs.ext4 root=/dev/ubd0 mem=128M eth0=mcast

Copy-On-Write (COW files)

Se provassimo ad eseguire i comandi riportati sopra, su due terminali diversi, uno dei due ritornerebbe un errore di lettura in quanto troverebbe il file system *Debian-Jessie-AMD64-root_fs* già "occupato" dall'altro processo. Per rendere possibile la condivisione di uno stesso file system fra più UML sono stati introdotti dei particolari file detti Copy-On-Write (COW files). Questi file lavorano in modo particolare, si accoppiano al file system al quale vengono associati e assumo l'aspetto di questo, ma senza andarlo a modificare. Ogni volta che dal terminale della UML avviata si andrà a scrivere su qualche file (es. Il file di configurazione di rete /etc/network/interfaces) modificando il file system, questa modifica verrà salvata all'interno del file COW senza andare ad intaccare il file system originale.
Citando l'esempio che riporta Dike nel libro User Mode Linux, 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 orginale (file system) è rimasto inalterato. Utilizzando questi particolari file non è necessario che il file system sia scrivibile, è necessario solamente che sia leggibile in quanto ogni modifica successiva verrà appunto fatta sul file COW.

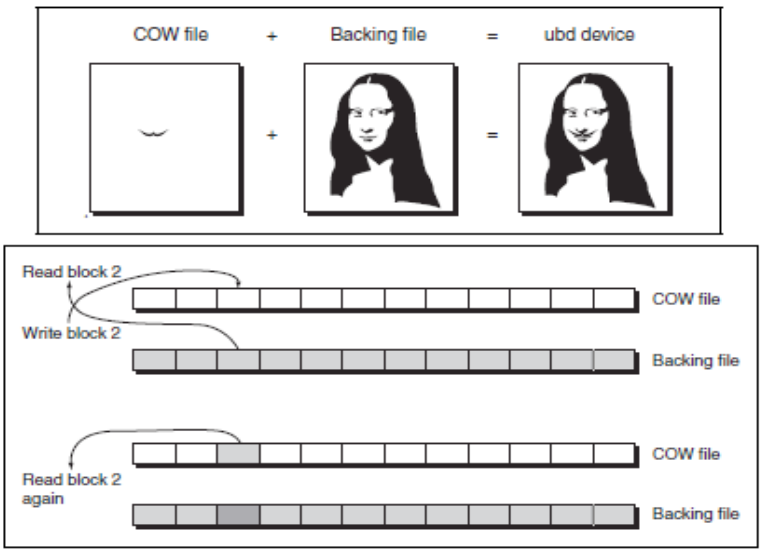

Fig 3.1

Questi file sono fondamentalmente una copia modificabile di un file system che vogliamo lasciare

inalterato in quanto potenzialmente utilizzabile da più di un host UML. Un'altra caratteristica è che si tratta di file "sparse". Ciò vuol dire che le dimensioni apparenti di questo file su disco sono uguali a quelle del file system a cui sono associati, che corrisponde in realtà alla quantità di dati che è possibile leggere da questo file.
Solitamente per i file la dimensione dei dati leggibili e la quantità di memoria allocata corrisponde, ma la particolarità dei file sparse è che la dimensione della memoria allocata è di gran lunga inferiore, corrisponde solitamente a qualcosa in più delle modifiche effettuate in fase di esecuzione.

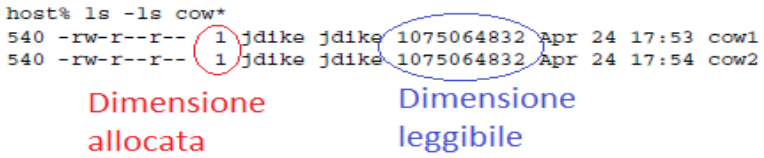

Fig 3.2

In pratica, per avviare la UML assegnando ad ogni disco, un file system col relativo COW basta semplicemente dichiararlo ogni volta che inserisco un dispositivo *ubd* nel comando di avvio.
Si osserva che non è necessario creare il COW prima dell'esecuzione, in quanto nel caso la UML non riesca a trovare il file dichiarato questa provvederà a crearlo in automatico.

```
Executing:  xterm -e "/home/alex/UML/Kernel/linux ubd0=tmp/prova2.cow,FileSystem/rootfs.ext4
Executing:  xterm -e "/home/alex/UML/Kernel/linux ubd0=tmp/prova3.cow,FileSystem/rootfs.ext4
Executing:  xterm -e "/home/alex/UML/Kernel/linux ubd0=tmp/prova4.cow,FileSystem/rootfs.ext4
```

Fig 3.3

In Fig 3.3 si notano 3 macchine eseguite assegnando lo stesso file system *rootfs.ext4* ma ognuna di queste lavora con COW diverso, evitando così di creare interferenze e problemi sul file in comune.
NOTA: la stringa è il comando eseguito dal framwork per lanciare più macchine virtuali su terminali differenti, *xterm -e* indica appunto di lanciare un terminale xterm ed eseguire la stringa fra doppi apici.

User Mode Linux Console

E' possibile inoltre gestire ogni istanza UML in esecuzione attraverso una console dedicata. Questa console, eseguibile dall'host, ha la possibilità attraverso alcuni comandi di gestire la macchina virtuale alla quale è associata.
In fase di esecuzione della UML, è possibile configurare per ognuna di queste una console assegnando al parametro *umid* il nome della socket della console attraverso la quale l'host dovrà comunicare (Fig 3.4).

```
Executing:  xterm -e "/home/alex/UML/Kernel/linux ubd0=tmp/prova2.cow,FileSystem/rootfs.ext4
                                              eth0=vde,/tmp/switch1 128m  umid=H2console H2"
Executing:  xterm -e "/home/alex/UML/Kernel/linux ubd0=tmp/prova3.cow,FileSystem/rootfs.ext4
                                    eth0=vde,/tmp/switch1 eth1=mcast 128m  umid=H3console H3"
Executing:  xterm -e "/home/alex/UML/Kernel/linux ubd0=tmp/prova4.cow,FileSystem/rootfs.ext4
                                              eth0=vde,/tmp/switch1 128m  umid=H4console H4"
```

Fig 3.4

Per mandare i segnali e gestire le macchine a livello di host invece basterà eseguire il comando *uml_mconsole* seguito dal nome della socket della console e il comando che si vuole eseguire.
Es. *uml_mconsole H3console version*

I comandi eseguibili dalla console sono i seguenti:

- *version* ritorna la versione del kernel dell'istanza UML.
- *halt* effettua uno shutdown del kernel. This will not perform a clean shutdown of the distribution. For this, see the cad command below. halt is useful when the UML instance can't run a full shutdown for some reason.
- *reboot* simile ad *halt*,ma in seguito riavvia la macchina.
- *config dev=config* aggiunge un nuovo dispositivo alla macchina.
- *config dev* ritorna la configurazione del dispositivo.
- *remove dev* rimuove il dispositivo dalla macchina.
- *sysrq letter* performs the sysrq action specified by the given letter. This is the same as you would type on the keyboard to invoke the host's sysrq handler.
- *cad* invokes the Ctrl-Alt-Del handler in the UML instance. The effect of this is controlled by the ca entry in the instance's /etc/ inittab. Usually this is to perform a shutdown. If a reboot is desired, /etc/inittab should be changed accordingly.
- *stop* mette in pausa la UML.
- *go* riprende l'esecuzione della macchina dopo uno *stop*.
- *log string* makes the UML instance enter the string into its kernel log.
- *log -f filename* is a uml_mconsole extension to the log command. It sends the contents of filename to the UML instance to be written to the kernel log.
- *proc file* returns the contents of the UML instance's /proc/file. This works only on normal files, so it can't be used to list the contents of a directory.
- *stack pid* returns the stack of the specified process ID within the UML instance. This is duplicated by one of the SysRq options—the real purpose of this command is to wake up the specified process and make it hit a breakpoint so that it can be examined with gdb.

3.2) VDE_SWITCH e UML_SWITCH

Uml_switch e vde_switch hanno la stessa funzione, quella di creare uno switch virtuale al quale le interfacce di rete delle varie macchine si andranno a connettere al fine di creare una rete.
Uml_switch è lo switch dedicato per la UML, che non ho utilizzato in questo progetto.
Al suo posto ho utilizzato invece vde_switch in quanto presenta una configurazione più semplice ed è dotato di un terminale dedicato in fase di avvio.
VDE è una sigla che sta per Virtual Distributed Ethernet e ognuno di questi switch ha delle sockets alle quali le macchine possono connettersi.

Ogni vde_switch funziona come uno switch reale, ha una socket attraverso la quale una macchina vi si connette e attraverso questo comunica con tutti gli altri host connessi (dopo una corretta configurazione dei dispositivi di rete).
Di default hanno 32 porte a cui è possibile connettersi, lavorano come switch e all'avvio creano un vde_terminal dove è possibile inserire dei comandi di configurazione dello switch. Tra questi comandi troviamo *pdump* per eseguire il dumping del traffico e *wireshark* per monitorare il traffico.
Si nota che non è possibile selezionare "fisicamente" a quale porta dello switch è possibile connettersi, semplicemente è un contatore che indica il numero massimo di macchine che possono connettersi alla socket dello switch.
Si possono modificare queste configurazioni di default, ad esempio cambiando il numero di porte inserendo *-n n_ports*. Di particolare importanza è il parametro *-x* che avvia lo switch in modalità hub, il che consente di redirigere il traffico ad ogni macchina connessa anche se i dati sono destinati solo ad una di queste (la funzione dell'hub, Fig 3.5).

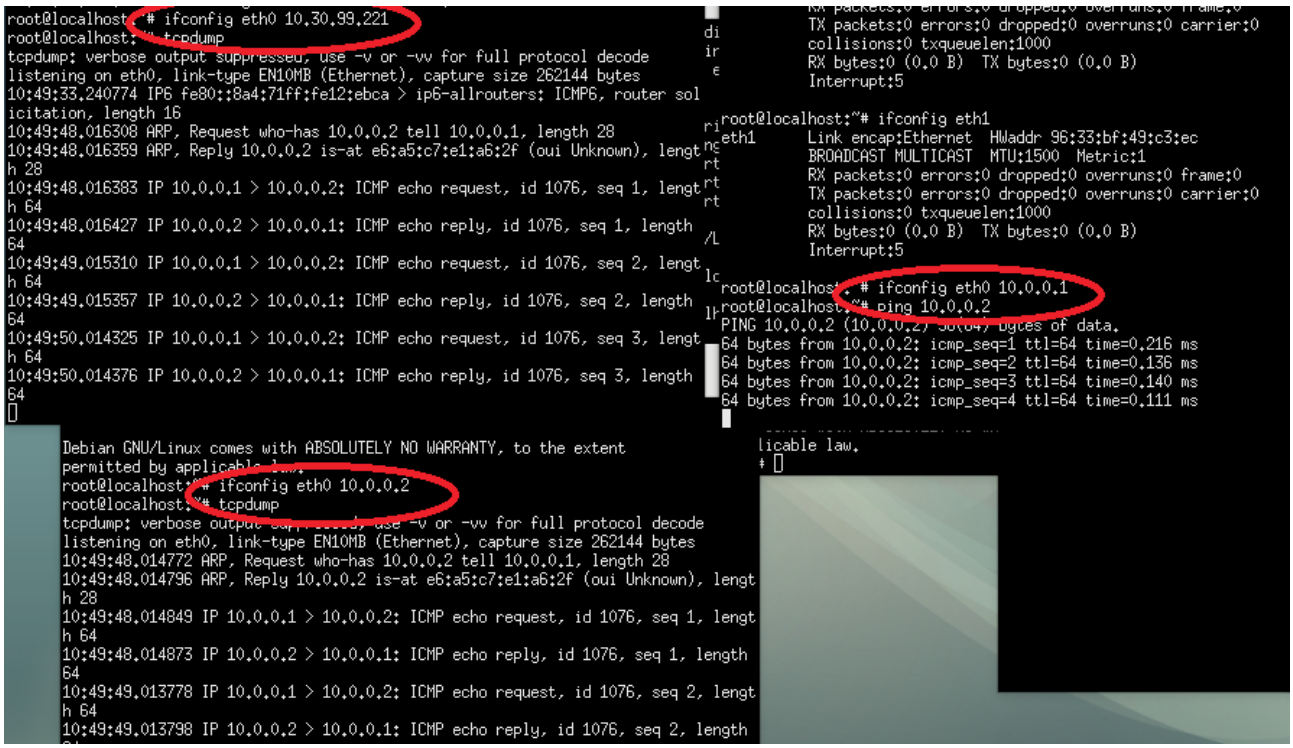

Fig 3.5

Infine è possibile avviare lo switch in modalità daemon, inserendo il paramentro *-d,* che consente di nascondere il terminale dello switch, nel caso non sia necessario usarlo durante la configurazione di rete.

4)Classi del framework Network Circus

Network Circus è costituito da serie di classi, con i relativi attributi e metodi, ognuna delle quali va ad identificare un oggetto della rete.
In fig 4.1 è riportato il diagramma, in maniera sintetica senza metodi e attributi , del framework.
In questo paragrafo verrà introdotto in modo sintetico tutto il framework, per dare un'idea chiara di come è strutturato, ogni classe verrà trattata in modo più approfondito nei diversi paragrafi.
La classe principale è rappresentata da Network, costitutita da diversi attributi, sono evidenziati in figura 3 indicanti le liste degli oggetti che andranno a costituire la rete vera e propria: Host, Switch e Cable (connessioni).

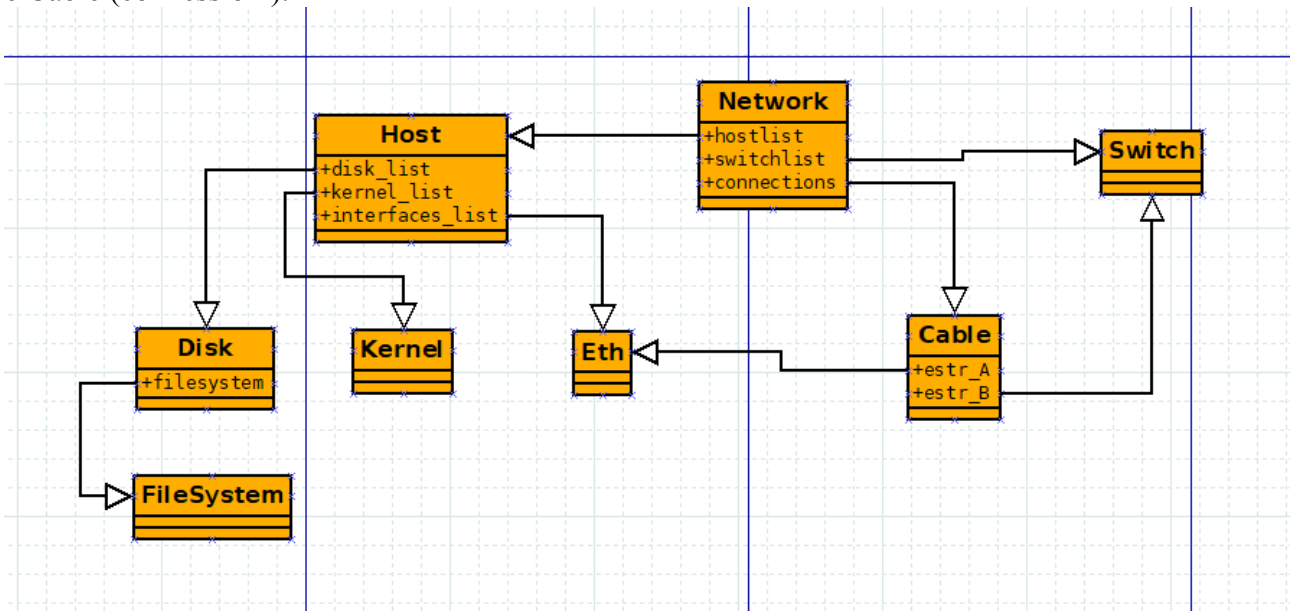

Fig 4.1 Diagramma sintetico del framework

Dalla classe padre network si passa poi alle 3 classi principali, prima fra queste la classe Host.
La classe Host, è a sua volta padre di altre 3 classi, che vanno a costituire la struttura reale di una macchina. Queste sono appunto Eth (interfaccia di rete), Kernel e Disk. Quest'ultima a sua volta ha è collegata alla classe FileSystem, infatti come spiegato in precedenza per ogni disco è possibile avere un file system differente.
Tornando alla radice, si ha poi la classe Switch e la classe Cable che costituisce le connessioni "fisiche" fra le interfacce di rete Eth e gli Switch.

-CLASSE Host (con Eth, File System, Kernel, Disks)
La prima classe che andiamo ad analizzare è la classe Host, questa rappresenta la singola macchina presente nella rete, con tutte le sue caratteristiche.
Questa classe è definita da diversi attributi, primo fra tutti il *nome* della macchina, che andrà a indetificarla all'interno della rete. Poi sono presenti 3 differenti liste: *interfacce di rete, kernel e dischi*. Ognuna di queste contiene appunto una lista di oggetti, rispettivamente Eth, Kernel e Disk che verranno trattati in dettaglio in seguito.
Mem va a indentificare la dimensione (in Mbyte) della memoria virtuale della macchina mentre *additional* rappresenta una eventuale stringa di configurazione aggiuntiva, oltre ai parametri standard, da andare ad aggiungere al comando di lancio.
Launcher_config possiamo definirlo come collegamento fra questo oggetto e la macchina user mode linux, in quanto questa stringa rappresenta il comando vero e proprio che permette di avviare una macchina UML (eseguibile e parametri già trattati nel paragrafo III.1).

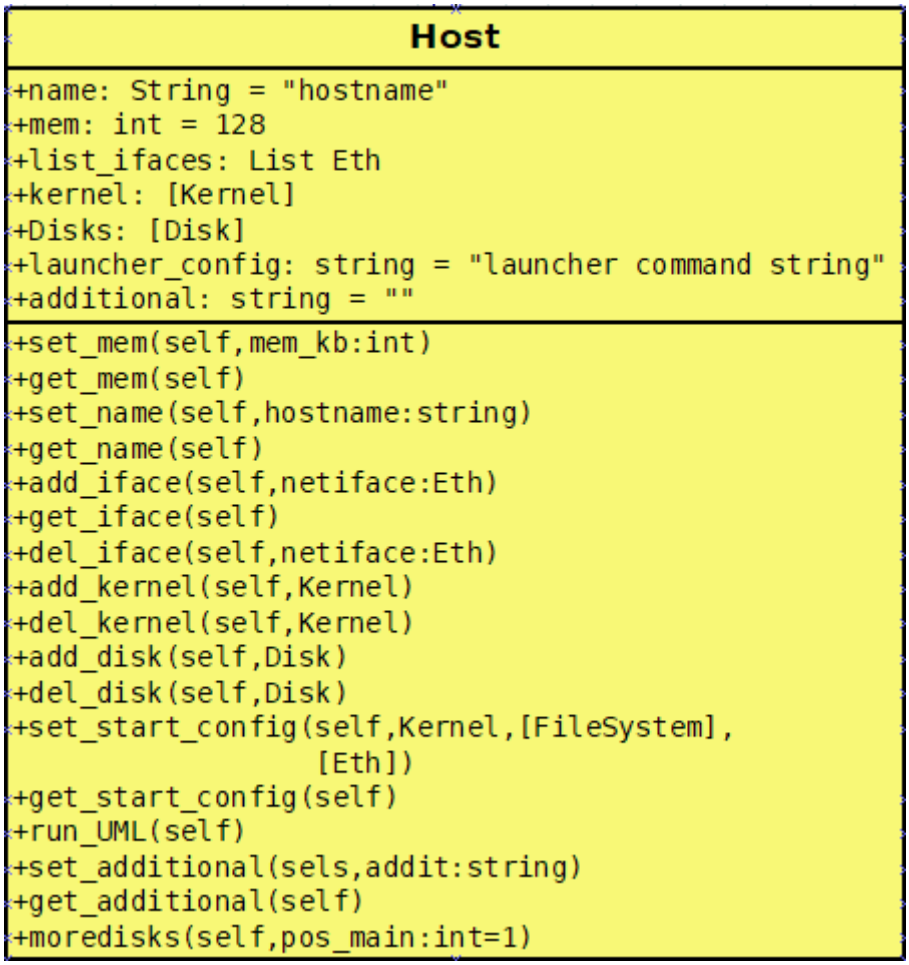

Fig 4.1 (Oggetto Host del diagramma DIA)

I metodi di questa classe sono quasi tutti dei getter e setter dei vari attributi: nei "*set_X"* passo come argomento il valore da assegnare al relativo attributo, mentre nei "*get_X"* ritorno tale valore. In particolare per le liste è presente anche il metodo che elimina l'oggetto passato come argomento.
Run_uml è il metodo che avvia il singolo host in modalità UML, è stato inserito per completezza, ma è consigliato per rendere più chiara la situazione della rete avviare la macchina attraverso il metodo *run_this_host* presente nella classe Network.
Il metodo *set_start_config* non è proprio un classico "setter" al quale passo un parametro e questo mi assegna tale valore all'attributo, infatti viene invocato con due paramentri che non ritroviamo negli attributi: un flag e un valore di indice.
Questo metodo analizza tutto l'oggetto Host, e seguendo l'ordine dei parametri del comando di

lancio della macchina UML spiegati nel punto III.1 va a creare la stringa di avvio con le specifiche configurate in questo oggetto, chiamando per ogni oggetto il metodo relativo di ritorno della propria sottostringa di configurazione.
Il flag *tmp_flag* serve ad identificare se la macchina appartiene ad una rete nuova (True di default), quindi senza cartella di salvataggio e quindi necessita di una cartella temporanea dove salvare i file oppure se è stata caricata quindi ha già una directory (False). *Ker_idx* indica quale Kernel di quelli presenti nella lista utilizzare nell'avvio della macchina, più precisamente indica la posizione nella lista dei kernel.

```
def set_start_config(self, tmp_flag=False, ker_idx=0):
    """
    Inizializza la stringa di lancio dell'host
    :param ker_idx: Indice di quale kernel utilizzare per il lancio
    :return:
    """

    kernel = self.kernel
    list_Disk = self.disks
    list_Eth = self.list_interfaces
    exec_string = kernel[ker_idx].launcher

    for dis in list_Disk:
        if tmp_flag is False:
            exec_string = exec_string + " " + dis.make_diskstring()
        else:
            exec_string = exec_string + " " + dis.make_tmp_launch()

    for iface in list_Eth:
        exec_string = exec_string + " " + iface.get_config()

    exec_string = exec_string + " " + str(self.mem) + "m " + self.additional_configs + \
                  " umid=" + self.name + "console" + " "
    self.launcher_config = exec_string
```

Fig 4.2 (Metodo set_start_config)

Infine abbiamo il metodo *moredisks* (Fig 4.3), da chiamare nel caso l'host debba lavorare con più di un disco e quindi è necessario configurare ogni disco con il giusto identificativo.
Viene chiamato con il parametro *pos_main,* questo va ad idenfiticare l'indice del disco nella lista che verrà considerato come disco principale (ubd0). Agli altri verrà semplicemente dato un valore ubdi corrispondente al valore del contatore .

```
def moredisks(self, pos_main):
    if len(self.disks) > 1:
        l = self.disks[:]
        main_disk = l.pop(pos_main - 1)
        main_disk.set_name('ubd0')
        for idx, d in enumerate(l):
            s_name = 'ubd' + str(idx + 1)
            d.set_name(s_name)
```

Fig 4.3 Metodo moredisks

Un host è caratterizzato essenzialmente dal Kernel sul quale gira e il File System presente su ogni disco. Per ognuno di questi tre elementi sono presenti delle classi che li vanno a creare in modo da semplificarne la gestione durante la creazione e configurazione della rete.

-CLASSE Kernel

La classe Kernel è identificata da 3 attributi, il più importante è il *launcher* che riporta il nome assoluto dell'eseguibile del kernel che si vuole usare per lanciare la UML.Mentre *ver* indica la versione del Kernel (2.16.4 è impostata di default in quanto l'ho usata durante i test, ma non rilevante ai fini dell'esecuzione della rete). Infine *type* indica il tipo di architettura del Kernel, personalmente non ho inserito alcun tipo di controllo perchè dato per scontato il fatto che vengano usati kernel compatibili con l'architettura della reale macchina Host dell'utente, è comunque possibile e di facile implementazione un metodo che controlla la compatibilità fra l'architettura della macchina Host e quella del kernel utilizzato per l'avvio della UML. Ho quindi inserito questo attributo per completezza e chiarezza, ma così come per *ver* è irrilevante (a livello di comandi) nel lancio della macchina virtuale.

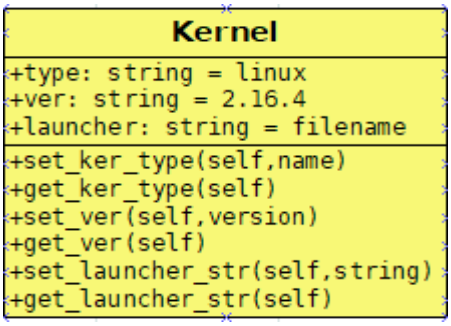

Fig 4.4

Essendo tutti attributi di tipo stringa, e il compito della classe è solo quello di rappresentare un oggetto che di fatto a livello di rete non deve gestire nulla (frase molto pesante....) i suoi metodi sono dei semplici getter e setter degli attributi.

-CLASSE Disk

La classe Disk è identificata da 3 attributi. Primo fra questi è *diskname*, ovvero il nome del dispositivo "fisico" che si creerà all'interno della macchina virtuale, di default è settato a "ubd0".
fs_cow è la stringa che identifica il file COW all'interno delle directory, tale file COW è associato al filesystem definito dall'oggetto *filesystem*.
Oltre ai getter e setter dei singoli attributi, sono presenti due differenti metodi che ritornano la stringa di configurazione del disco. Per comodità e maggior leggibilità dei metodi ho preferito separare il caso in cui si abbia una rete nuova, quindi lavora in una cartella tmp (*make_tmp_launch*), da quella in cui la rete abbia già una directory di lavoro (*make_disktring*)

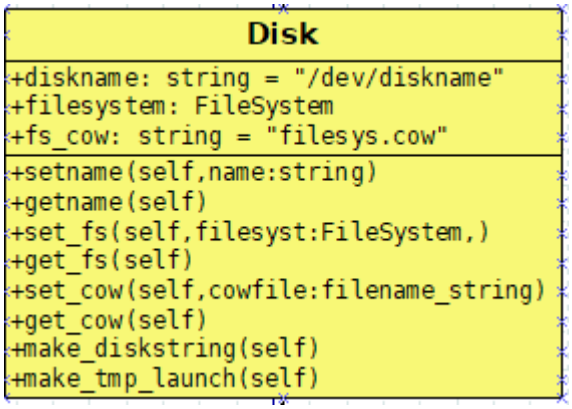

Fig 4.5

-CLASSE FileSystem

La classe FileSystem è molto semplice, costutita da tre attributi e i relativi getter e setter.
L'attributo *filesystem* indica semplicemente il nome dell'oggetto, per identificarlo in una lista. *fs_distr* è un attributo pensato per un'eventuale interfaccia grafica dove si voglia distinguere il nome effettivo del file system e la sua distribuzione più generale (Es. Linux Mint basato su distribuzione Debian).
L'ultimo *file* è invece il più importante, contiene il nome del file system all'interno della macchina dell'utente da passare all'oggetto Disk per compilare la sua stringa di lancio.

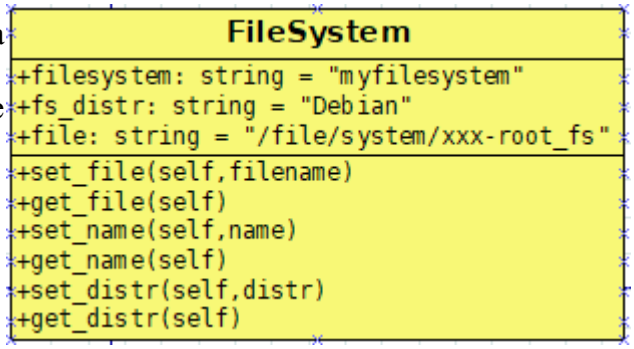

Fig 4.6

-CLASSE Eth

La classe Eth rappresenta le interfacce di rete degli Host. Come spiegato in precedenza, nel lancio della UML le interfacce sono identificate da un *id*, solitamente "eth" seguito dal numero in progessione dell'interfaccia. *Mode* indica la modalità con cui dovrà lavorare questa interfaccia, settata a multicast di default. Nel caso venga collegata ad uno switch, tale modalità verrà settata uguale al nome della socket di tale switch. *dev_of* indica su quale macchina è montata mentre *ip* è un aggiunta che potrà servire in una successiva implementazione grafica.
I metodi sono dei getter e setter dei vari attributi, ad eccezione di *get_config* che ritorna la stringa di configurazione da inserire nell'avvio della UML.

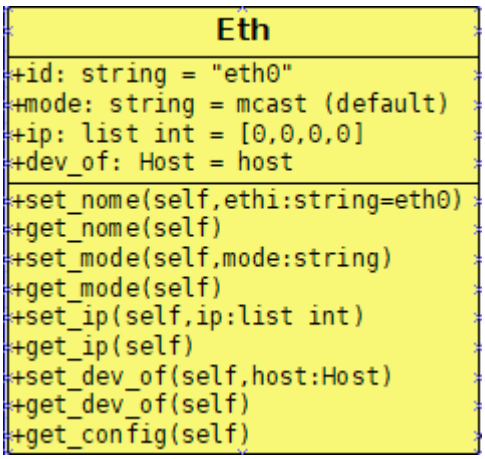

Fig. 4.7

-CLASSE Switch

La classe switch rappresenta appunto i vde_switch. Identificata da una serie di attributi, primo fra tutti è il nome *switchname* che oltre ad identificare l'oggetto switch indica anche il nome della socket che si crea nel lancio del vde. Troviamo poi *nport* che indica il numero di porte con cui viene creato, vde_switch ha come default 32 porte quindi in mancanza di parametri viene settato appunto a 32. *port* è una lista con i nomi delle porte, ai fini di funzionamento è intuile ma è comodo per la gestione delle porte e dei collegamenti a livello del framework, così come *available*.
vde_switch e le interfacce eth non necessitano nessuna specifica sul numero di porta a cui va "inserito il cavo", quindi la lista *available* viene in aiuto per indicare quante porte restano ancora disponibili oltre ad essere utile per un'eventuale interfaccia grafica dove è comodo selezionare a quale porta connettersi. *Hub* è un flag che indica se settare l'oggetto come normale switch o se farlo lavorare come hub, mentre *visible* indica se lanciare lo switch con terminale vde visibile oppure come daemon. *Additional_cfg* è una stringa in cui possono essere inseriti parametri addizionali da passare allo switch, al comando di lancio.
Per creare l'oggetto si deve semplicemente passare il nome della socket e il flag che indica se lavorare come switch (True) o hub (False). Di seguito il metodo *__init__* della classe.

```
class Switch:

    switchname = "/tmp/vde_switch1" #indica inoltre dove viene salvata la socket
    n_port = 32 #default
    port = ["p1", "p2", "p3", "p4", "p5", "p6", "p7", "p8", "p9", "p10", "p11", "p12", "p13", "p14", "p15", "p16", \
          "p17", "p18", "p19", "p20", "p21", "p22", "p23", "p24", "p25", "p26", "p27", "p28", "p29", "p30", "p31", "p32"]
    available = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
    hub = False
    launcher_command = "vde_switch -s"
    additional_cfg = ""
    visible = True

    def __init__(self, nome,  hub, additcfg="", nport = 32,
            lisport = ["p1", "p2", "p3", "p4", "p5", "p6", "p7", "p8", "p9", "p10", "p11", "p12", "p13", "p14", "p15", "p16", \
          "p17", "p18", "p19", "p20", "p21", "p22", "p23", "p24", "p25", "p26", "p27", "p28", "p29", "p30", "p31", "p32"],
            avail = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0] ,
            visibile = True,  launcher="vde_switch -s"):
        self.switchname = nome
        self.n_port = nport
        self.port = lisport[:]
        self.available = avail[:]
        self.hub = hub
        self.launcher_command = launcher
        self.additional_cfg = additcfg
        self.visible = visibile
```

Fig. 4.8

Tra i metodi troviamo i getter e setter degli attributi di tipo stringa, il metodo *set_as_hub* che in caso si voglia cambiare la modalità in fase di esecuzione cambia direttamente da switch ad hub. I metodi *launch_vdeswitch* e *get_launch_cmd* che rispettivamente avviano e ritornano il comando di lancio dello switch.
Di particolare importanza sono i metodi per la gestione delle porte, primo fra tutti set_portlist che nel caso venga modificato il numero di porte questo va ad aggiornare le liste *port* e *available* di conseguenza.
get_first_free è un metodo molto comodo che ritorna l'indice della prima porta dello switch disponibile. *reset_ports* resetta tutte le porte, rendendole disponibili.
Infine *set_port* modifica la disponibilità della porta n con la disponibilità passata come terzo parametro.

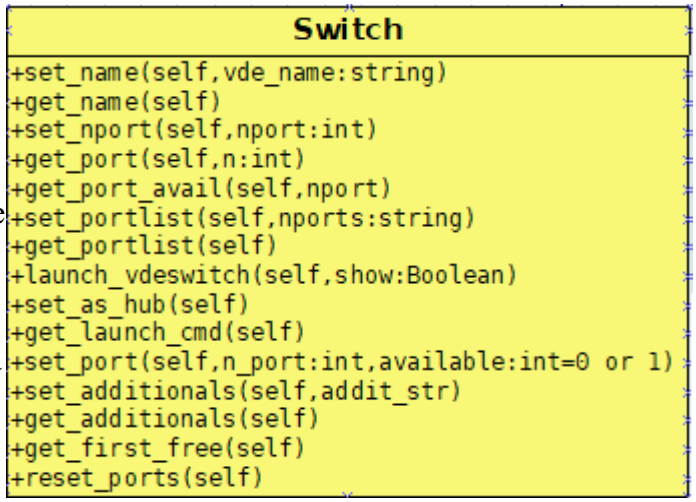

Fig 4.9

-CLASSE Cable
I cable rappresentano le connessioni fra host e switch. La classe Cable è identificata da un *id,* formato dall'unione dei nomi dell'host e dello switch. La connessione ha due estremi, A e B, che per maggior semplicità di gestione sono fissati come A per il lato host e B per il lato switch.
switchname è l'oggetto switch al quale il cavo è connesso, alla porta n.
In fase di creazione dell'oggetto Cable devono essere passati tutti i parametri, ma per evitare poi andare a modificare manualmente gli oggetti Eth e Switch coinvolti conviene in seguito utilizzare i metodi *set_Estr*, in quanto vanno anche ad aggiornare le configurazioni degli oggetti.
I due metodi *free_estr* vanno a liberare le disponibilità della porta dello switch e a reimpostare la modalità dell'interfaccia di rete, di fatto "scollego" l'host dallo switch.

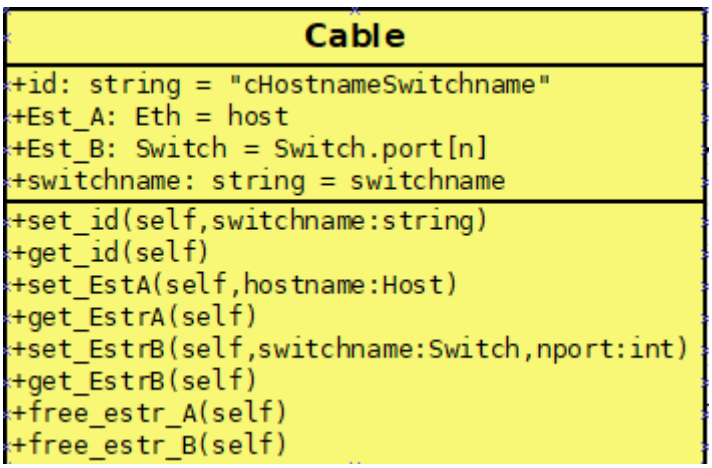

Fig. 4.10

-CLASSE PADRE: Network

La classe "padre" Network, che costituisce la rete, quindi l'oggetto che conterrà tutti gli altri elementi della rete.

Come si vede in figura 4.X , la classe Network è costituita da pochi attributi:

-Nome della rete: Identifica, in una ipotetica lista di diverse reti pre-configurate, la rete in questione con il proprio nome identificativo (Es. "LaMiaReteX")

I tre successivi sono delle semplici liste, dalle quali estraggo o inserisco elementi

-Lista degli host: Contiene tutti gli oggetti Host della rete

-Lista degli Switch: Contiene tutti gli oggetti Switch della rete

-Lista delle connessioni: E' la lista delle connessioni, tali connessioni sono identificate dagli oggetti Cable

-Flag di on/off: Flag pensato per una implementazione successiva in un'interffaccia grafica, indica semplicemente se la rete è avviata o no.

-Directory: Indica la directory da dove "sto lavorando", in questa troverò tutti i file della rete.

Per quanto riguarda i metodi (Fig 4.11) non sono presenti particolari getter e setter. In fase di creazione dell'oggetto Network si passa semplicemente il nome che si vuole dare alla rete, poi si assegna la directory in cui lavora la rete tramite il metodo *set_dir*.

In seguito verranno riempite le varie liste di Host, Switch e Cable con i relativi oggetti precedentemente creati. Tali liste possono, ovviamente, venire aggiornate durante la creazione iniziale dell'oggetto Network, ma nel caso si debba inserire una discreta quantità di oggetti uno alla volta, per evitare di avere una stringa di inizializzazione eccessivamente lunga e confusa con i metodi "add_*oggetto*" consentono di inserire singolarmente all'interno della lista l'oggetto in questione, rendendo così più chiaro il procedimento. Ovviamente sono presenti i metodi che ritornano le liste o che eliminano un elemento da queste.

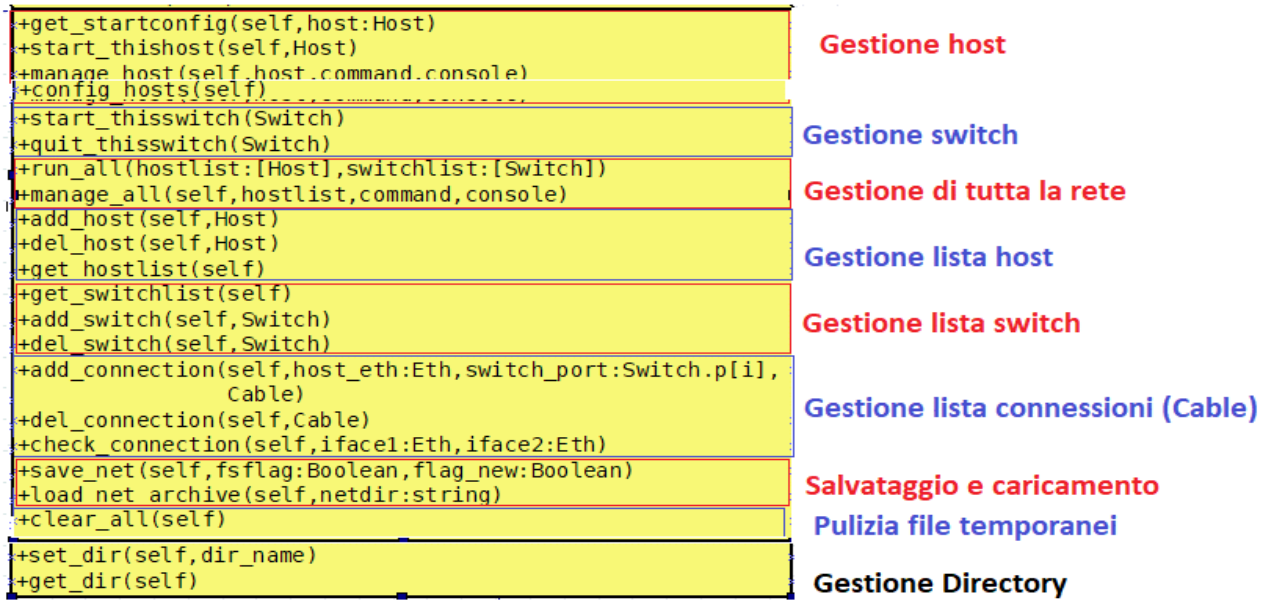

Fig 4.11 (Frammento del diagramma DIA: Operations della classe Network)

I metodi di gestione di host e switch singoli sono dei metodi che vanno ad agire su un oggetto singolo. I metodi "start_" vanno ad avviare il singolo host (in modalità UML) o switch (come vde_switch) che viene passato come argomento.

Il metodo "manage_host" invia un comando *command* all'host passato come argomento, questo avviene tramite un segnale della uml console, di cui ho parlato in precedenza.

Config hosts è un metodo che aggiorna la stringa di lancio di ogni singolo host presente nella lista degli host della rete.

Run_all è forse uno dei metodi più importanti di questa classe, in quanto questo avvia tutti gli host e switch presenti nella rete.

Add_connection prende semplicemente come argomenti quelli necessari per creare un oggetto Cable: interfaccia di rete, indice dello switch nella lista, numero di porta. In seguito andrà a creare un oggetto Cable e lo inserirà automaticamente nella lista.
Check connection è un metodo che serve sostanzialmente a controllare se due macchine sono connesse allo stesso switch, ritorna un Boolean True se le due interfacce sono collegate allo stesso dispositivo switch.
"Save" e "Load" sono i due metodi che rispettivamente creano e caricano l'archivio compresso contenente tutti i file e le configurazioni della rete.
Durante l'esecuzione della rete è possibile che si creino (in base alle modalità e varie configurazioni) dei file temporanei all'interno delle varie cartelle, il metodo *clear_all* va a ripulire le directory della rete da tutti questi file temporanei.
Una volta creati tutti gli oggetti della rete ed inseriti nelle rispettive liste, sia "manualmente" sia tramite un caricamento da una rete salvata in precedenza, non si dovrà fare altro che lanciare il metodo di configurazione degli host (*config_hosts)* per essere sicuri che tutte le macchine siano settate nel modo giusto. Dopodichè basterà eseguire il comando *run_all (* o i singoli *start_this_switch/start_this_host* a seconda del caso) per avviare la tutta la rete.

Di seguito la serie di comandi per creare una semplice rete:
Nel dettaglio abbiamo due host con un'unica interfaccia di rete collegati al primo switch, un terzo host con due interfacce collegato ad entrambi gli switch e un ultimo host collegato solo al secondo switch.

OGGETTI DELLA RETE

```
k = Kernel.Kernel("/home/alex/UML/Kernel/linux")
f = FileSystem.FileSystem("FileSystem/rootfs.ext4")
d = Disk.Disk("ubdo", f, "prova1.cow")
d2 = Disk.Disk("ubdo", f, "prova2.cow")
d3 = Disk.Disk("ubdo", f, "prova3.cow")
d4 = Disk.Disk("ubdo", f, "prova4.cow")
e = Eth.Eth()
e2 = Eth.Eth()
e3 = Eth.Eth()
e31 = Eth.Eth("eth1",)
e4 = Eth.Eth()
h = Host.Host("H1", 128, [], [k], [d], "")
h.add_netiface(e)
h2 = Host.Host("H2", 128, [], [k], [d2], "")
h2.add_netiface(e2)
h3 = Host.Host("H3", 128, [], [k], [d3], "")
h3.add_netiface(e3)
h3.add_netiface(e31)
h4 = Host.Host("H4", 128, [], [k], [d4], "")
h4.add_netiface(e4)
s1 = Switch.Switch("/tmp/switch1", False)
s2 = Switch.Switch("/tmp/switch2", False)
```

OGGETTO "NETWORK"

```
n = Network.Network("NetOne")
n.add_host(h)
n.add_host(h2)
n.add_host(h3)
n.add_host(h4)
n.add_switch(s1)
n.add_switch(s2)
n.add_connection(n.get_hostlist()[0].get_netiface()[0], 1, 1)
n.add_connection(n.get_hostlist()[1].get_netiface()[0], 1, 2)
n.add_connection(n.get_hostlist()[2].get_netiface()[0], 1, 3)
n.add_connection(n.get_hostlist()[2].get_netiface()[1], 2, 1)
n.add_connection(n.get_hostlist()[3].get_netiface()[0], 2, 2)
print("Configurazioni degli host:")
n.config_hosts()
print("\n \nAvvio della rete: ", n.netname)
n.run_all(n.lis_host, n.lis_switch)
```

Fig 4.12 (Screenshot dei comandi, schermata dell'IDLE di python)

5) Esecuzione, salvataggio e spegnimento della rete

5.1) AVVIO

L'avvio della rete è la fase più delicata da gestire, è importante garantire diverse caratteristiche in questa fase. Prima di tutto è necessario che nessuno dei processi avviati blocchi o interferisca con un altro, per fare questo ho sfruttato la libreria "*threading*" di Python 3.6, lanciando ogni *thread* in modalità *daemon.* Così facendo, ogni processo lavora in background e nessuno di questi va a bloccare o interferire con gli altri.
Lanciare un processo daemon permette di avviare un qualsiasi numero di thread in parallelo, senza bloccare quello padre e consente inoltre di gestire indipendentemente ognuno di questi.

```
def run_all(self, host_list, switch_list):
    """
    Avvia tutte le macchine della rete
    :param host_list:
    :param switch_list:
    :return:
    """

    list = []

    for item in host_list:
        command = 'xterm -e "' + item.launcher_config + item.name + '"'
        list.append(command)

    for item in switch_list:
        command = 'xterm -e "' + item.get_launch_cmd() + '"'
        list.append(command)

    for idx, l in enumerate(list):
        idx = threading.Thread(target=Network.run, args=(l,))
        idx.daemon = True
        idx.start()
```

Fig. 5.1 Metodo run_all della classe Network

In figura 5.1 è riportato il metodo *run_all* della classe *Network* . E' un metodo molto semplice, in quanto per ogni oggetto Host e Switch presente nelle liste della Network viene ritornata la relativa stringa di avvio ed inserita in una lista *list.* Dopo aver creato la lista, per ogni elemento presente al suo interno verrà creato un thread daemon che andrà ad eseguire il comando run, passando come unico parametro l'elemento i-esimo della lista, indipendentemente che sia relativo ad uno switch o ad un host.
Il metodo run è un semplice comando *system* della libreria Python *os*, che mi esegue nella shell il comando passato come argomento *l*.
Il comando è costituo dall'avvio del terminale Unix *xterm* seguito dal parametro *-e* e la stringa di lancio dell'host o switch (Fig 5.2).
Il comando ' *xterm -e "stringa" '* avvia un terminale ed esegue la stringa al suo interno, così facendo ogni terminale appena avviato esegue, a seconda del caso, una macchina UML o un vde_switch.

```
Executing:  xterm -e "/home/alex/UML/Kernel/linux ubd0=tmp/prova1.cow,File
Executing:  xterm -e "/home/alex/UML/Kernel/linux ubd0=tmp/prova2.cow,File
Executing:  xterm -e "/home/alex/UML/Kernel/linux ubd0=tmp/prova3.cow,File
Executing:  xterm -e "/home/alex/UML/Kernel/linux ubd0=tmp/prova4.cow,File
Executing:  xterm -e "vde_switch -s /tmp/switch1 "
Executing:  xterm -e "vde_switch -s /tmp/switch2 "
```

Fig 5.2 Comandi di lancio

NOTA: Non è stato possibile assegnare, in fase di avvio, un titolo esplicito al terminale in quanto i comandi *-t* e *–title* sono deprecati da qualche versione di Debian (sistema operativo usato per scrivere questo framework).

Una volta finita la lista e lanciati tutti i thread con i relativi comandi di lancio delle UML o vde si avrà una schermata simile a quella in figura 5.3.

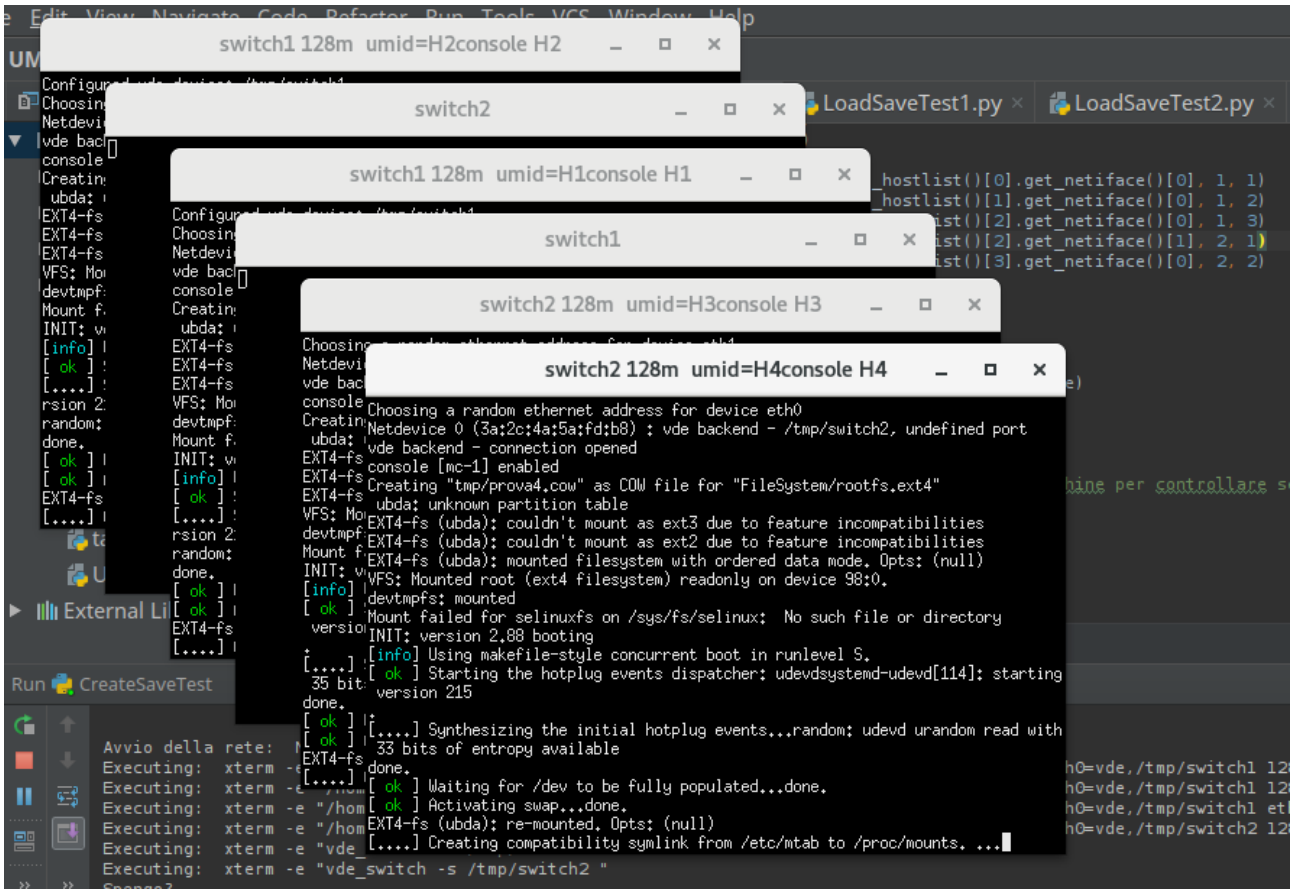

Fig 5.3 Quattro macchine e due vde_switch avviati

Nella Fig 5.2 sono state avviate 4 macchine in modalità UML e 2 switch in modalità vde. Come già espresso nella NOTA non è possibile esplicitare il titolo del terminale attraverso il comando di lancio di *xterm* ma tale terminale riconosce le ultime stringhe del comando e le inserisce come titolo. Ho risolto al problema del titolo inserendo in coda al comando un parametro "inutile" alla UML, ovvero il nome dell'oggetto Host, così facendo in ogni titolo in coda ho il nome della macchina a cui si riferisce tale terminale.

5.2)ESECUZIONE

Se tutti i terminali si sono avviati con successo, è possibile comunicare con le macchine UML attraverso la *uml_console* di cui ho già parlato nel capitolo 3.

Per gestire la console ho creato un metodo, all'interno della classe Network, che permette di inviare un segnale a tutte o solo ad una macchina.

```
def manage_host(self, host, command, console="uml_mconsole"):
    """
    Gestisce la uml console della macchina host, lanciando il comando command
    :param host:
    :param command:
    :param console:
    :return:
    """
    socket = host + "console"
    s = console + " " + socket + " " + command
    print(s)
    os.system(s)

def manage_all(self, hostlist, command, console="uml_mconsole"):
    """
    Invia un comando command a tutte le console della lista hostlist
    :param hostlist:
    :param command:
    :param console:
    :return:
    """

    for host in hostlist:
        socket = host.name + "console"
        s = console + " " + socket + " " + command
        os.system(s)
```

Fig 5.3 metodi di gestione della console della UML

Ho creato due metodi per comodità, così da rendere evidente quando si lavora con solo una macchina o con tutte. Ma è comunque possibile utilizzare solo il metodo *manage_all* e passare come primo parametro una lista con un solo elemento.
Il secondo parametro è il comando che la console deve inviare alla macchina mentre il terzo è opzionale, fissato a default come *uml_mconsole* che è il comando per avviare la console della UML.

In Fig 5.4 riporto un esempio di comunicazione fra console e macchina. E' stato inviato un semplice comando *version* che chiede alla macchina la versione del Kernel utilizzato.

```
Mando un segnale a tutte le macchine per controllare se sono avviate e funzionanti
OK Linux localhost 3.16.4 #5 Tue Nov 4 14:04:09 CET 2014 i686
OK Linux localhost 3.16.4 #5 Tue Nov 4 14:04:09 CET 2014 i686
OK Linux localhost 3.16.4 #5 Tue Nov 4 14:04:09 CET 2014 i686
OK Linux localhost 3.16.4 #5 Tue Nov 4 14:04:09 CET 2014 i686
```

Fig 5.4 Output del comando *version* passato alla *uml_mconsole*

5.3)CONFIGURAZIONE DI RETE COME JSON

In questo paragrafo andremo a parlare dei metodi di salvataggio delle configurazioni della rete e dei file che la costituiscono. Come struttura di file ho utilizzato il JSON (JavaScript Object Notation), basato sul concetto "Chiave:Attributo", che si prestava perfettamente per l'utilizzo che dovevo farne.

5.3.1) Salvataggio della configurazione di rete

Per salvare la configurazione ho creato due metodi, esterni alle classi sopra citate, che gestiscono la lettura e scrittura di un file JSON.
Partendo dalla creazione, il primo metodo *save_network_tofile* riceve in input tre parametri: un oggetto Network, un flag che indica se la rete è nuova e un flag che indica se salvare o meno i file system.

```
h_list = self.lis_host
s_list = self.lis_switch
c_list = self.lis_connections

#print(h_list, "\n", s_list, "\n",c_list)

# Creo il dictionary degli host

h_dict = []
if h_list is not None:
   for h in h_list:
      name_h = h.name
      memory = h.mem
      eth = []
      for e in h.list_interfaces:
         obj = "Eth"
         id_eth = e.id
         mode = e.mode
         ip = e.ip
         devof = e.dev_of
         details = {'object': obj, 'id': id_eth, 'mode': mode, 'ip': ip, 'devof': devof}
         eth.append(details)
                                    ....
                                                                    ....

                                    kernel = []
                                    for k in h.kernel:
                                       ker = {'object': 'Kernel',
                                              'kernel_type': k.ker_type,
                                              'version': k.ver,
                                              'launcher': k.launcher}
                                       kernel.append(ker)

                                    dis = []
                                    for d in h.disks:
                                       obj = "Disk"
                                       name = d.diskname
                                       cow = d.fs_cow
                                       cow = cow.split('/').pop()
```

Fig 5.5

Partendo da un oggetto Network, in modo iterativo e ricorsivo per ogni oggetto va a creare un oggetto dictionary, caratterizzato da una chiave e un argomento:
dict = { key : arg , key2 : arg2 , ... }
La chiave sta ovviamente ad indicare il significato dell'argomento a cui è associato.
Durante questo processo si creano diversi dictionary e liste che andranno a creare l'oggetto dictionary finale.
Il procedimento di base consiste nel trasformare ogni oggetto viene in un dictionary mentree ogni lista rimane tale.
Partendo dalla classe padre Network, si crea una serie di liste di dictionary, una per ogni lista di oggetti: host, switch e cable.
In seguito analizzo tutte le liste dell'oggetto Network e per ogni elemento creo l'oggetto dictionary da andare ad inserire nella lista dedicata. Nel caso si incontri un'altra lista all'interno si procede in modo iterativo.
Finite le tre liste principali si identificano gli ultimi attributi dell'oggetto Network e si crea il dictionary generale che andrà a contenere tutti i precedenti che sono stati creati. Fig 5.5

Questo dictionary che si viene a creare viene alla fine del processo scritto su un file, il cui nome è il nome della rete, con l'estensione appunto *.json*.

In figura 5.6 sono presenti le ultime righe di codice di questo metodo. *network_dict* è il dictionary finale, che è creato inserendo come argomenti i dictionary di ogni lista di oggetti da cui è creata la rete (host, switch e cable).

Si apre quindi un file in modalità scrittura, denominato con la procedura descritta in precedenza, e viene scritto il dictionary al suo interno, rispettando le indentazioni rendendo il file di più facile lettura. Alla fine viene spostato tale file

```
network_dict = {'network_name': self.netname,
                'host_list': h_dict,
                'switch_list': s_dict,
                'connections_list': c_dict}

# Creo e scrivo il file json
f_name = self.netname +".json"


with open(f_name, 'w') as outfile:
    json.dump(network_dict, outfile, indent=4)
outfile.close()

if tmp_flag:
    s = 'mv ' + f_name + ' tmp/'
    os.system(s)
else:
    s = 'mv ' + f_name + ' ' + self.dir + '/'
    os.system(s)
```

Fig5.6 all'interno della directory in cui la rete sta lavorando.

5.3.2) Caricamento di una configurazione di rete

Il caricamento avviene ovviamente in modo inverso rispetto a come avviene il salvataggio. Prende come input il nome di un file JSON, un nome assoluto di directory (quella in cui si trova tale file) e il flag che indica se è stato salvato o meno anche il file system.
Il metodo *load_network_fromfile* legge da un file JSON una configurazione e la salva all'interno di un dictionary. Procedendo in maniera inversa a quella del salvataggio, estrae le liste di dictionary di host, switch e cables e ne crea le liste di oggetti. Dopo aver compleato tutte le liste di oggetti crea l'oggetto Network e inserisce tali liste e gli ultimi parametri mancanti non ancora letti dal dictionary. Alla fine del procedimento ritorna l'oggetto Network appena creato (Fig. 5.7).

```
c_list = []

while True:
    try:
        c_item = cl.pop()
        cid = c_item.get('cableid')
        cestb = c_item.get('estr_B')
        for S in s_list:
            if (S.switchname == c_item.get('switchname')):
                break
        swi = S
        hn = c_item.get('estr_A_do')
        ifacename = c_item.get('estr_A_id')
        for ho in h_list:
            if ho.name == hn:
                for et in ho.list_interfaces:
                    if et.id == ifacename:
                        break
                break

        c = Cable.Cable(cid, et, cestb, swi)

        c_list.append(c)
    except Exception:
        print("End cable connectionss")
        break
nname = net_config.get('network_name')

net = Network.Network(nname, h_list, s_list, c_list)
print(net)

return net
```

Fig 5.7 Blocco finale del metodo *load_network_fromfile*

5.4) CREAZIONE ARCHIVIO DELLA RETE

Oltre alla configurazione degli oggetti della rete, è presente anche un metodo ulteriore che consente di salvare la configurazione JSON e tutti i file della rete all'interno di un unico archivio. Così facendo si andrà a creare un unico file dove al suo interno saranno presenti configurazione, file system e file cow e sarà quindi possibile caricare e avviare la rete salvata in modo rapido e semplice.

Data la consistente dimensione dei file, si avrebbero degli archivi dalle dimensioni eccessivamente grandi (<10GB) per una rete da pochi host. E' stato necessario quindi ricorrere anche ad una compressione di questo file per rendere la dimensione di questo file molto più piccola.
Di seguito sono presenti i risultati dello script usato come test fatto per selezionare quale metodologia di compressione sia migliore.
Ho analizzato la compressione "xz", "bz2" e "gz", valutando dimensioni di output e tempo di compressione.
E' stato usato come file di prova un singolo file system di Debian 8.9.

Output dello script:

```
                    /usr/bin/python3.5 /home/alex/UML/tartests.py
        Starting files size:  2646605824  bytes ( 2584576.0  kbytes)

                    Compression time with xz :  515.9539415836334
                      Archive size (in kbytes) :  363225.51171875

                   Compression time with bz2 :  180.07360076904297
                      Archive size (in kbytes) :  455966.80859375

                    Compression time with gz :  81.90987873077393
                      Archive size (in kbytes) :  498505.11328125

     The fastest compressor is:  gzip  with time:  81.90987873077393 s
          The lighter archive is:  xz with  363225.51171875 kbytes

                                    Process finished with exit code 0
```

I risultati del test riportano come metodo più veloce la compressione *gz* mentre come riduzione di dimensioni la migliore è *xz*. Analizzando tali risultati, si nota che il tempo di compressione di *xz* è eccessivamente alto (per 10 GB di file può impiegare fino a 15 minuti) mentre *gz* è quello con la dimensione di output maggiore.
Sono giunto alla conclusione che usando *bz2*, ovvero andando a creare archivi con estensione "*.bzip2*" sia il miglior compromesso, in quando il tempo di compressione è molto inferiore a quello di *xz* e la dimensione è inferiore a quella di *gz*. Nei paragrafi successivi, quando si parlerà di file compressi si intenderà attraverso questa modalità.

5.5) SALVATAGGIO E CARICAMENTO DELLA RETE

All'interno della classe Network sono presenti due metodi, *save_net* e *load_net_archive* che rispettivamente creano e caricano l'archivio contenente tutte le informazioni della rete.

Partendo dal salvataggio, il metodo *save_net* prende come input il flag per il file system e il flag che indica se la rete è nuova o è una pre-esistente.
Caso 1) Rete nuova
Viene creato il file di configurazione JSON della rete (che viene salvato nella cartella *dir* in cui la rete sta lavorando). Successivamente, viene creata una cartella il cui nome è quello della rete, poi

con un comando tar vengono archiviati e compressi i file cow, il json di configurazione. Se il flag *fsflag* è settato a True, viene salvato all'interno dell'archivio anche il file system. Il file system viene salvato in una sottocartella dedicata (es. FileSystem/fsname.extn) quindi ne verrà creata solo una copia per ogni tipo di file, senza creare doppioni che andrebbero solo ad appesantire inutilmente l'archivio.

```
if flag_new:

    JsonNet.save_network_tofile(self, self.netname, flag_new)
    jsonname = self.netname + ".json"
    fs = ""

    pwd = self.dir[:-4]

    cmd = "mkdir " + pwd + self.netname
    os.system(cmd)

    tarcommand = "cd " + self.dir + " && tar cSjf " + pwd + self.netname + '/' + self.netname + ".tar.bz2 " \
                 + jsonname + " "
    for h in self.lis_host:
        for d in h.disks:
            if fsflag is True:  # Controllo qui o nel comando i doppioni???
                if d.filesystem.file in tarcommand:
                    fs = ''
                else:
                    fs = '../' + d.filesystem.file
                tarcommand = tarcommand + d.fs_cow + " " + fs + " "
```

Fig 5.8

Caso 2) Rete esistente
Se la rete è esistente, quindi è stata caricata da un salvataggio precedente, all'inizio c'è una fase di controllo in cui si verifica se la directory in cui vogliamo salvare la nuova rete è la stessa di quella da cui è stata caricata oppure la si vuole cambiare.
Per decidere questo, prima di lanciare il comando di salvataggio basterà cambiare il nome della rete col nuovo nome (che andrà a definire anche quello della directory) così facendo si andrà a creare la nuova cartella prima di lanciare l'archiviazione vera e propria.
Verificate queste condizioni iniziali, si procede come per il caso 1), si accedere alla cartella dalla quale si sta lavorando e si archivia in quella di output tutti i file presi una sola volta.

```
if not flag_new:
    dirhome = ''
    for idx, line in enumerate(self.dir.split('/')):
        if idx == (len(self.dir.split('/')) -1):
            break
        dirhome = dirhome + line + '/'

    fs = ""

    if self.dir == dirhome:
        self.netname = self.dir.split('/').pop(-2)
        JsonNet.save_network_tofile(self, dirhome + self.netname, flag_new)
        jsonname = self.netname + ".json"
        tarcommand = "cd " + self.dir + " && tar cSjf " + dirhome + '/' + self.netname + \
                     ".tar.bz2 " + jsonname + " "
    else:
        cmd = "mkdir " + dirhome
        os.system(cmd)
        JsonNet.save_network_tofile(self, dirhome + self.netname, flag_new)
        jsonname = self.netname + ".json"
        tarcommand = "cd " + self.dir + " && tar cSjf " + dirhome + self.netname + '/' + self.netname + \
                     ".tar.bz2 " + jsonname + " "

    for h in self.lis_host:
        for d in h.disks:
            if fsflag is True:
                fslong = d.filesystem.file
                fslong = fslong.split('/')
                fsf = fslong.pop()
                fs = fslong.pop() + '/' + fsf
                if fs in tarcommand:
                    fs = ''
                tarcommand = tarcommand + d.fs_cow.split('/').pop() + " " + fs + " "

print("Executing: ", tarcommand)
os.system(tarcommand)
```

Fig5.9

Il caricamento avviene ovviamente in modo analogo ma nel verso opposto. Il metodo *load_net_archive* viene chiamato passando come parametro la directory da cui deve essere caricata la rete. Il metodo prende il percorso assoluto ricevuto e ne estrapola il nome della rete e di conseguenza del file .tar.bzip2, dopo di che entra nella cartella e lo scompatta.
Il comando tar è chiamato con l'opzione *-v (verbose)*, così da avere in output l'elenco dei file scompattati. Tale elenco viene rediretto in un file di testo temporaneo che viene aperto e letto a fine estrazione. Prima di tutto cerca se al suo interno sono presenti o meno dei file system, così da impostare il flag per il caricamento della configurazione della rete, in seguito cerca il file JSON di configurazione, chiama il metodo *load_network_fromfile* e crea di fatto il nuovo oggetto di rete Network che viene ritornato da questo metodo.

```
with open(filetmp, "r") as f:
    for line in f:
        line = line[:-1]
        for end in fs_ext:
            if line.endswith(end):
                fs_flag = True

with open(filetmp,"r") as f:
    for line in f:
        if '.json' in line:
            config = self.dir + line[0:-1]
            net = JsonNet.load_network_fromfile(config, dir, fs_flag)
            os.system(s)
            net.set_dir(dir)
            net.config_hosts()
            print(net.dir, " Fine caricamento")
            return net
```

Fig. 5.10 Frammento di codice di *load_net_archive* in cui vengono controllati i file scompattati

5.6) GESTIONE DEI FILE

Si sono voluti gestire i file in modo tale da rendere ogni salvataggio indipendente dalla propria directory e dall'utente che l'ha creata.
Prima di tutto, è importante notare che tutti i file coinvolti e creati durante tutto il processo sono dotati delle autorizzazioni necessarie per essere letti da tutti gli utenti ed è quindi possibile condividerle fra più user.
L'unico file che necessita di permessi anche di esecuzione è il kernel, che in questo framework non è stato considerato come file da salvare e quindi da poter "condividere".

I file coinvolti nell'esecuzione di questo framework sono i COW files, kernel files e file system files. Le funzioni e gli utilizzi di questi sono già stati discussi nei primi capitoli, qui mi limiterò a spiegare come ognuno di questi è stato gestito a livello di lettura e scrittura.

```
k = Kernel.Kernel("/home/alex/UML/Kernel/linux")
f = FileSystem.FileSystem("FileSystem/rootfs.ext4")

d = Disk.Disk("ubdo", f, "prova1.cow")
d2 = Disk.Disk("ubdo", f, "prova2.cow")
d3 = Disk.Disk("ubdo", f, "prova3.cow")
d4 = Disk.Disk("ubdo", f, "prova4.cow")
```

Fig 5.11 Passaggio dei file come parametri di inizializzazione degli oggetti

COW*:*
Come già spiegato i file COW, se non esistenti, vengono creati in automatico durante l'esecuzione di user mode linux passando semplicemente il nome di tale file che si vuole leggere/creare.
In questo framework ho considerato tutti i file COW gestendoli con il loro nome relativo semplice: *nome.cow*.
Questi file vengono passati come parametro nella creazione dell'oggetto Disk, assieme al nome del disco e del file system. In seguito verrà differenziata la gestione se si tratta di una rete nuova, quindi questi COW verranno creati all'interno di una cartella temporanea, oppure se si tratta di una rete pre-esistente e quindi verranno caricati da un percorso assoluto.

In fase di creazione della stringa di lancio per ogni macchina, come già spiegato vengono differenziati i casi di rete nuova e caricata, infatti per ognuno di questi nella creazione della parte di stringa riferita ai dischi ci sono due metodi differenti:

Codice della classe *Disk*:

```
def make_diskstring(self, d=""):
    if len(d) < 1:
        d = ""
    str = self.diskname + "=" + d + self.fs_cow + "," + d + self.filesystem.file
    return str

def make_tmp_launch(self):
    str = self.diskname + "=" + 'tmp/' + self.fs_cow + "," + self.filesystem.file
    return str
```

Si nota che il secondo metodo è relativo alla rete nuova, quindi ritorna la configurazione del disco con un file COW all'interno di una cartella *tmp/*.
Altrimenti viene chiamato il primo metodo e aggiunge alla radice del nome il percorso assoluto della directory da cui è stato caricato.
E' importante notare che il parametro *fs_cow* dell'oggetto non viene mai cambiato, ma semplicemente in fase di configurazione gli viene aggiunto all'inizio il percorso corretto in cui si trova o deve essere creato. Così facendo è possibile gestirli lavorando da qualsiasi directory.
Data la particolarità di questi file, durante l'archiviazione di questi è importante inserire il parametro "*-S*" che serve a garantire l'integrità di questi e che non vengano archiviati e compressi con dei dati mancanti.

FILE SYSTEM:
I file system, come si vede in Fig 5.11 sono invece considerati sempre all'interno di una loro directory *FileSystem/*, di conseguenza l'attributo *file* del FileSystem viene inizializzato a *FileSystem/nome.estensione*.
Questa directory è presente all'interno di quella del framework e vi si possono aggiungere tutti i file system che si desidera.
A differenza dei file COW, quando si tratta di una rete nuova non viene creata una cartella temporanea ma viene utilizzata quella citata sopra come percorso iniziale.
Nel caso invece che si tratti di una rete caricata e che sia stato salvato il file system al suo interno, si procede in modo analogo ai file COW e in fase di creazione della stringa di configurazione di avvio si aggiunge alla radice il percorso assoluto della directory in cui è salvata la rete (vedi "Codice della classe *Disk* ").

In fase di salvataggio come spiegato nel paragrafo 5.5 è possibile decidere se salvare o meno il file system all'interno dell'archivio. Nel caso si decida di salvarlo, al suo interno verrà archiviata la cartella *FileSystem* e il relativo file (non tutti i file al suo interno, ma solo quello utilizzato dalla

rete in questione).

KERNEL:
Per quanto riguarda i kernel, nella realizzazione di questo progetto ho considerato l'utilizzo di kernel già configurati e in particolare il file è solamente l'eseguibile utilizzato per il lancio della User Mode Linux.
Come per i File System all'interno del framework è presente la cartella Kernel in cui verranno salvati tutti gli eseguibili dei kernel che sarà possibile utilizzare. E allo stesso modo viene gestito il loro utilizzo in fase di creazione della stringa di configurazione di lancio: se la rete è nuova considero come percorso quello della cartella Kernel se invece è caricata considero come percorso quello assoluto della rete caricata seguito da quello della directory Kernel.

Attualmente non sono implementati metodi per il salvataggio e il caricamento dei kernel della rete, questi vengono sempre considerati all'interno della directory Kernel e non è possibile salvarli all'interno dell'archivio. Ma in futuri miglioramenti ed implementazioni sarà un punto su cui lavorare.

5.7) SPEGNIMENTO DELLA RETE

Lo spegnimento della rete è una fase molto importante e va gestita nel modo corretto per evitare la creazione di errori all'interno dei file utilizzati.
In particolare i file COW sono molto "delicati" in questa fase, in quanto nel caso di una chiusura in maniera errata possono danneggiarsi irrimediabilmente e diventare illeggibili in futuro.
Sconsiglio caldamente di chiudere i terminali delle UML in modo brutale con la "x", dei kill o qualsiasi altro metodo che non sia quello spiegato in seguito.

Nel framework, lo spegnimento delle macchine è gestito attraverso le console di ogni signola macchina. All'interno dell'oggetto Network sono presenti due metodi per la gestione della console: *manage_all* e *manage_host.* Questi rispettivamente mandano il comando passato come parametro a tutte o solo una delle macchine in esecuzione.
Per spegnere quindi una (o più) macchina basta semplicemente invocare uno dei due metodi sopra citati passando come parametro *command* la stringa "halt". Tale comando spegne in modo corretto e sicuro la macchina associata alla console che lo esegue, facendolo quindi per tutte le console si arresta l'intera rete.

```
def manage_host(self, host, command, console="uml_mconsole"):
    socket = host + "console"
    s = console + " " + socket + " " + command
    print(s)
    os.system(s)

def manage_all(self, hostlist, command, console="uml_mconsole"):
    for host in hostlist:
        socket = host.name + "console"
        s = console + " " + socket + " " + command
        os.system(s)
```

Fig 5.12 Metodi per la gestione delle console

Lo spegnimento degli switch invece non avviene tramite console, ma con un semplice comando da terminale che cerca la socket del vde_switch attiva tra i processi e la killa.

```
def quit_switch(self, switch = None):
    if switch is None:
        for s in self.lis_switch:
            string = "pgrep -f " + s.switchname + " | xargs kill -TERM"
            print(string)
            os.system(string)
        #os.system("pgrep -f vde_switch | xargs kill -TERM")
    else:
        string = "pgrep -f " + switch.switchname + " | xargs kill -TERM"
        os.system(string)
```

Fig 5.13

In particolare questo metodo *quit_switch* considera tutti gli switch attivi se non viene passato nessun parametro, quindi ricerca all'interno della rete tutte le socket dei relativi switch e le chiude. Mentre se viene passato un'oggetto switch, viene spento solo questo.

Nella figura a destra si vede l'output del metodo *manage_all* passando "halt" come comando. La console ritorna OK quando lo spegnimento della macchina è avvenuto in maniera corretta. In segutio sono presenti i due comandi del terminale che ricercano le socket degli switch e le "uccidono".

```
OK Linux localhost 3.16.4 #5 Tue Nov 4 14:04:09 CET
Spengo le macchine
OK
OK
OK
OK
Spengo gli switch
pgrep -f /tmp/switch2 | xargs kill -TERM
pgrep -f /tmp/switch1 | xargs kill -TERM
```

6) Conclusioni

6.1) Raggiungimento degli obiettivi prefissati

Tutti i metodi presenti all'interno del framework per la gestione della rete sono tutti scritti in maniera semplice e diretta. Questo è dovuto in gran parte alla semplicità e potenza del linguaggio Python con cui è stato scritto, mentre da un lato ho cercato di rendere tutti i metodi principali più facili ed intuitivi. Impostando tutti i parametri che non richiedono particolari attenzioni già settati di default, così da rendere molto più veloce e semplice la creazione di una rete.
Ho inserito diversi metodi di controllo, come ad esempio per verificare se gli host sono connessi e se una configurazione è stata fatta in modo corretto.
Tutto il codice del framework è integrato con dei commenti o esempi di output che si dovrebbe avere, in modo da renderlo più chiaro ad una terza persona che dovesse leggere e/o modificare.

6.2) Conoscenze necessarie per l'utilizzo del framework

Per l'utilizzo di questo framework è richiesta prima di tutto la conoscenza del linguaggio Python. In particolare le classi sulla gestione dei multiprocessi, le interazioni con il sistema unix.
Inoltre è di fontamentale importanza conoscere l'ambiente Linux, come vengono gestite le impostazioni sulla connettività e le reti, la gestione dei file e delle directory e le basi dei terminali Unix.

6.3) Future implementazioni

La prima idea sul futuro di Network Circus si basa sulla sua possibile implementazione all'interno di un software dedicato alla creazione e gestione di reti virtuali, come ad esempio Marionnet citato nell'introduzione. L'idea è quella di partire da questo software, migliorarlo dove possibile ed implementare un'interfaccia grafica intuitiva e pratica tale da rendere lo studio e la gestione di reti virtuali il più comoda e facile possibile, potenzialmente per un uso didattico ma non solo.

Tra le possibili migliorie si evidenzia prima di tutto il problema del titolo, come accennato nel capitolo 6.2. Si dovrebbe rendere più semplice, magari riportando solo il nome della macchina e qualche dato fondamentale su decisione dell'utente.
Inoltre si potrebbe implementare la possibilità di connettere fra loro due oggetti simli (host-host , switch-switch) in quanto tutt'ora non è ancora possibile.
Di grande utilità sarebbe inoltre andare a creare uno script che agisce su ogni macchina, dove, in fase di post avvio va a modificare le configurazioni di rete delle varie interfacce secondo delle modalità inserite in fase di configurazione. Da questo punto, nella fase di creazione degli oggetti nell'interfaccia grafica, inserire diverse opzioni che appunto verranno ereditate dalle macchine quando verranno avviate (es. IP, netmask, subnet, gateway ...)
Riconoscere a livello di interfaccia grafica le modifiche apportate alle macchine all'interno della user mode linux (es. Evidenziare le macchine nella stessa subnet).

Per queanto riguarda l'interfaccia grafica, sicuramente si dovrà creare un drag & drop, con la possibilità di disporre i vari oggetti in maniera libera. Inoltre rendere ogni oggetto modificabile tramite l'apertura di un menù a tendina cliccando sopra l'oggetto stesso, senza andare a cercarlo nell'elenco generale. Inoltre creare, per tutti gli attributi di tipo boolean (es. Modalità hub dello switch) dei flag comodi da poter spuntare all'occorrenza senza dover necessariamente creare un nuovo oggetto.

6.4) Bug

Il primo bug che ho riscontrato è un errore che si verifica in modo casuale nella fase di avvio. Infatti può succedere a volte che il terminale, di conseguenza anche la macchina virtuale, non venga avviata e non è possibile capire quale sia il problema in quanto non venendo aperto nessun terminale e quindi non si crei nessun processo non c'è modo di controllare il valore di ritorno e analizzarlo per capire quale sia il problema e quale sia la sua fonte.
Ho effettuato diversi test di avvio mantenendo sempre le stesse configurazioni e, con una percentuale molto bassa (circa 1 avvio su 20) succedeva che uno o più terminali non venissero aperti. L'unica soluzione che ho trovato era quella di spegnere (in maniera sicura) tutte le macchine e riavviare la rete.
Questo errore non riporta danni a lungo termine, come ad esempio danneggiamenti dei file coinvolti nella macchina non avviata, ma solo una situazione di disagio all'avvio in quanto appunto si rende necessario il riavvio della rete.

Il secondo e ultimo bug si verifica a rete già avviata, dove può verificarsi la situazione in cui anche se due host sono connessi e configurati in modo che in linea teorica dovrebbero riuscire a comunicare questo invece non avviene. Questo è dovuto probabilmente al file system che ho utilizzato per i test finali, in quanto inizalmente operavo con un altro file system (sempre di una distribuzione Debian) e questo problema non si è mai presentato. La probabilità che questo accada è molto bassa, in quanto fra tutti i test che ho fatto si è verificato solo un paio di volte, ma credo sia giusto riportarlo ugualmente. Come il bug precedente, la soluzione è spegnere e riavviare le macchine coinvolte (se si salvano le configurazioni di rete non è necessario reimpostarle).

Infine riporto un problema a livello di GUI: il titolo del terminale delle macchine e degli switch. Essendo la funzione *-t* o *–title* di *xterm* deprecata, non mi è stato possibile impostare per ogni terminale come titolo il solo nome della macchina. Per fare questo sarei dovuto andare a modificare i file di ambiente e sistema per ogni terminale, ma avrebbe comportato un eccessivo lavoro che mi portava al di fuori degli obiettivi del progetto.
Fortunatamente *xterm* setta come titolo la parte finale della stringa che gli viene passata come comando da eseguire, quindi per ogni terminale si ha il nome dell'host, la sua console e il nome della socket dell'ultimo switch a cui è connessa la macchina. Compaiono questi termini in quanto sono gli ultimi parametri inseriti nella stringa di esecuzione della UML (Fig 6.1).

Fig 6.1