Università degli studi di “Roma Tre”

Facoltà di Ingegneria

Corso di Laurea Magistrale in Ingegneria Informatica

Progettazione e Realizzazione di un Ambiente per la Configurazione Avanzata di Reti Virtuali

Relatore

Prof. Maurizio Pizzonia

Laureando

Alessio Di Fazio

Correlatore

Dott. Massimo Rimondini

Anno accademico 2007/2008

Alla mia famiglia che mi ha dato la possibilità di raggiungere questo importante traguardo.
A me per la mia tenacia e forza di volontà.
A tutte quelle persone che non hanno mai creduto in me! :)

Indice

Introduzione

Una *rete di calcolatori* può essere definita come un sistema che permette la condivisione di informazioni e risorse tra un insieme di calcolatori e di apparati di rete collegati, tramite un mezzo trasmissivo. Negli ultimi decenni si è assistito ad un eccezionale sviluppo di quella che può essere considerata la "rete delle reti", ovvero *Internet*. *Internet* non è altro che un enorme agglomerato di sottoreti eterogenee che sono collegate con le più svariate tecnologie e che insieme costituiscono una trama capace di connettere quasi ogni luogo del pianeta. Il concetto di "rete" è costantemente in evoluzione e modifica profondamente, giorno dopo giorno, il mondo in cui viviamo. Si pensi alle nuove tecnologie del campo *Mobile*: tutto ci conduce ad una realtà dove ogni individuo ha la possibilità di essere collegato ad *Internet* ovunque si trovi.

Parallelamente è cresciuto il numero di aziende, professionisti e persone che sfruttano la *Net* per svolgere le proprie attività. Questi, oltre che migliorare, integrare e fornire servizi sempre più evoluti, trasformano *Internet* in un sistema attivo in continua espansione. È per questo motivo che, fino ad oggi, molti sforzi sono stati volti alla creazione di strumenti capaci di studiare ed analizzare tutti gli aspetti che comportano la realizzazione e l'utilizzo di una rete. Uno di questi, impiegato per lo studio delle funzio-

nalità di una rete di calcolatori, è l'*emulatore di reti*. Questi ultimi possono essere definiti come *strumenti software* che riproducono in maniera esatta il funzionamento ed il comportamento dei componenti hardware che essi emulano. Gli emulatori permettono quindi di fare a meno di costose apparecchiature escludendo tutti i rischi di danneggiamento che potrebbero correre i sistemi reali durante il loro *tuning*, minimizzando così l'impatto economico. Per questi motivi tali strumenti vengono spesso utilizzati quando esiste la necessità di progettare una rete *ex-novo* o di studiare le funzionalità, analizzare le architetture e testare il funzionamento di protocolli e delle configurazioni esistenti su reti reali.

La gran parte degli ambienti di emulazione esistenti risentono però di gravi debolezze che ne limitano l'utilizzo in alcune particolari realtà. Prima tra queste l'intrinseca fragilità del modello logico su cui si basano. Questi sistemi infatti, concentrano i loro sforzi nella sempre più perfetta emulazione degli elementi coinvolti, senza curarsi della definizione di un modello formale per la rete. Tipicamente, il modello utilizzato ricalca e forse rappresenta un'astrazione dell'emulatore su cui poggiano. Ciò comporta l'inevitabile perdita di flessibilità dell'ambiente di emulazione e l'impossibilità di definire uno schema comune per le reti emulate.

Questo lavoro offre una soluzione a tale problema proponendo un modello formale per le reti vituali emulate. In prima istanza, è stato necessario realizzare uno studio puntuale dei vincoli introdotti dalla mancanza di uno schema adeguato. Successivamente si è proceduto alla definizione di un modello formale che fosse in grado di rappresentare le diverse reatà di interesse di una rete mantenendo intatte le doti di generalità, coerenza ed

astrattezza, consone ad un modello logico.

Un'altra grave carenza di cui soffrono tali sistemi, è la scarsa flessibilità che dimostrano. Ogni ambiente infatti, dispone di una base architetturale compasta da un motore di emulazione studiato *ad-hoc*, e resta ad esso irrevocabilmente vincolato.

L'ambiente realizzato nell'ambito di questo progetto supera tale vincolo grazie al modello logico su cui si centra ed alla provata certezza che un simile strumento debba collocarsi ad un livello d'astrazione superiore a quello dei sistemi di emulazione. Per conseguire tale obiettivo è stato necessario sciogliere lo stretto legame presente tra il modello della rete e l'emulatore, astraendo e creando uno schema logico adatto alla rappresentazione di ogni rete virtuale.

Scopo primario di quest'attività di tesi è stato la realizzazione un ambiente grafico per la configurazione avanzata di reti virtuali emulate. Tale strumento si basa su un modello comune per la realizzazione e la configurazione di reti virtuali, garantendo all'utente un'estrema usabilità e configurabilità. Molte sono state le problematiche affrontate nel realizzare uno strumento che risultasse nel contempo estremamente flessibile ed usabile.

Il punto di forza che dona al tool caratteristiche univoche è l'adozione di una struttura modulare che poggia su di un'*architettura a Plug-In*. Tale composizione ha permesso infatti, l'estensione delle funzionalità del sistema in modo assai più veloce e meno dispendioso, incrementando notevolmente caratteristiche quali flessibilità ed estensibilità. Tuttavia questa tipologia architetturale, seppur altamente flessibile, presentava una forte limitazione per quanto concerne l'espressività dei singoli *Plug-In*.

In questo lavoro si è analizzato il problema sopra citato, trovando una soluzione che estendesse la capacità espressiva dei moduli.

Nella prima sezione si trattano i sistemi di emulazione offrendo una panoramica degli ambienti esistenti e delle specifiche funzionalità. Successivamente vengono paragonati gli ambienti di configurazione messi a disposizione dai sistemi di emulazione, enfatizzando pregi e difetti dei singoli strumenti.

Nel secondo capitolo si effettua uno studio approfondito atto a estrarre un modello che accomuni le varie configurazioni avanzate dei potenziali servizi attivi su una *Virtual Machine*. Si affontano quindi due scenari reali di configurazioni avanzate: i servizi BGP e OSPF. Lo schema risultante viene adottato mostrando la reingegnerizzazione affrontata che ha portato alla definizione del nuovo *plugin framework* adottato da *VisualNetkit*.

Nella terza sezione si descrivono le fasi di progettazione e sviluppo di questo progetto, le metodologie utilizzate, le scelte realizzative e gli strumenti di supporto alle singole fasi. Si illustra in dettaglio la composizione dell'architettura, le difficoltà riscontrate e le motivazioni per le scelte effettuate.

Infine, viene proposto uno scenario reale che descrive le fasi che l'utente deve eseguire per costruire una configurazine avanzata di una rete virtuale, con l'ausilio dell'ambiente qui discusso.

Capitolo 1

Stato dell'arte

In questo capitolo di apertura si vuole fornire una breve descrizione dei sistemi di emulazione di reti di computers esistenti, mettendoli a confronto per capire le loro potenzialità; in particolar modo verrà descritto come questi si comportano davanti ad un utente che vuole creare configurazioni per reti complesse.

Nella seconda parte verranno discusse le qualità della prima release[1] di *VisualNetkit*, enfatizzando i punti deboli. Questa prima versione verrà messa a paragone con le potenzialità offerte dal nuovo e più flessibile ambiente realizzato per tale lavoro.

1.1 Panoramica sui sistemi di emulazione

Al fine di comprendere cosa rappresenta un "ambiente di emulazione" risulta necessario focalizzare l'attenzione sul significato stesso della parola

[1] Release, letteralmente "rilascio", in ambito informatico indica una particolare versione di un software resa disponibile ai suoi utenti, univocamente identificata da altre particolari versioni rese disponibili in precedenza da un particolare numero di versione.

"emulazione". Un software di emulazione, più comunemente chiamato "emulatore", è un programma scritto per un ambiente (hardware o software) diverso da quello sul quale viene eseguito che permette l'esecuzione di software.

Un'applicazione scritta e compilata per una determinata piattaforma software (ad esempio *Windows*) non viene eseguito su un computer con sistema operativo differente (come *Linux*). In questi casi si crea sulla macchina ospitante "Host" un emulatore che riproduce virtualmente l'ambiente che è stato usato per creare quel programma (nel nostro esempio un ambiente *Windows*). Il sistema virtuale che gira all'interno di quello emulato viene chiamato sistema "Guest".

I sistemi di emulazione di reti nascono con l'intento di riprodurre il funzionamento delle reti reali e di tutti i servizi che queste offrono agli utenti: configurazioni particolari, protocolli, ecc. . .

Una rete di calcolatori può essere definita come un sistema informatico costituito da due o più calcolatori che, collegati tra loro tramite un mezzo trasmissivo, possono scambiarsi informazioni di vario genere. Naturalmente nella realtà le cose risultano essere molto più complesse rispetto a quanto detto. Proprio per questo motivo un sistema di emulazione deve offrire la possibilità di gestire qualsiasi configurazione e comportamento che i calcolatori presenti in rete possono assumere, permettendo così una rappresentazione eterogenea di macchine che offrono diversi servizi e svolgono diverse attività.

I sistemi attualmente esistenti consentono la creazione di reti composte da centinaia di computers generando, di conseguenza, un proporzionale

aumento di macchine, interfacce di rete e protocolli coinvolti.

Ma perché emulare reti di calcolatori?

Lo scopo principale di un sistema di emulazione di reti di calcolatori è quello di riprodurre e sperimentare il funzionamento dei nodi[2] e dei connettori[3], al fine di testare nuovi protocolli e verificare il corretto funzionamento della rete. Tutto questo viene affrontato in maniera "virtuale" senza dover acquistare dispositivi, spesso molto costosi.

Si pensi ad esempio all'ambito didattico: lo studente (o il ricercatore, o l'individuo che intende studiare e sperimentare reti di calcolatori) dovrebbe dapprima analizzare la soluzione "su carta", acquistare le dovute apparecchiature di rete (che, come noto, hanno un impatto economico non trascurabile), assemblare la rete, configurare adeguatamente ogni singolo nodo e connettore della rete e, solamente alla fine, passare alla fase di test vera e propria.

Se lo studente volesse modificare una parte della rete, lo scenario che nella maggior parte dei casi si profilerebbe non sarebbe praticabile.Si può notare pertanto come l'uso dei sistemi di emulazione semplifichi notevolmente tali attività rendendo possibile l'intera progettazione su di un singolo calcolatore che ospita un sistema di emulazione.

[2]Per "nodo" in una rete di calcolatori si intendono computers provvisti di memoria.
[3]Per "connettore" si intendono dispositivi come ad esempio hub, bridge, ecc...

1.1.1 Classificazione dei sistemi di emulazione

Allo stato attuale esistono molteplici tipologie di soluzioni offerte nel campo dell'emulazione di sistemi. In questi ultimi anni il bisogno di prototipi ed ambienti per la realizzazione di esperimenti nel campo delle reti ha catturato grande attenzione nel mondo della ricerca e dello sviluppo, dirottando notevoli sforzi in tal senso. Questo ha portato alla distinzione di tre principali classi di metodologie e strumenti: reti *testbed*, *simulazione*, emulazione di reti e *Virtual Machine*.

Le reti *testbed* sono ambienti costruiti su un hardware reale, come routers o hosts, interconnessi e propriamente configurati atti a realizzare la tipologia di rete desiderata. Sebbene questa soluzione sia in grado di offrire un elevato grado di realismo, gli ambienti *testbed* tendono ad essere difficilmente realizzabili per via dell'elevata difficoltà di mantenimento e soprattutto per costi talvolta proibitivi e raramente accessibili.

I simulatori offrono ambienti per la realizzazione di reti concettuali. Questi sono tipicamente strumenti altamente configurabili ed estensibili, progettati per testare e valutare le dinamiche di rete in un ambiente disaccoppiato da ogni sorta di traffico o sistema esterno.

Gli emulatori di reti possono essere considerati un innesto di reti *testbed* e simulatori di reti, essendo in grado di riunire le caratteristiche proprie del traffico e dei sistemi coinvolti nelle reti reali. La maggior parte di essi infatti, è costituito da un *motore di emulazione* e di un'interfaccia che permetta di interagire con lo stesso. Il motore dell'emulatore assolve il compito di mantenere in piedi il sistema emulato agendo a basso livello, spesso con

primitive di sistema. Esso è in grado di riprodurre virtualmente tutte le componenti hardware e software, e quindi, di predisporre un ambiente standard "gradito" al sistema che si andrà ad emulare.

Le *Virtual Machine* infine, si possono considerare un "PC nel PC". Ossia, mediante una *Virtual Machine*, è possibile installare un secondo sistema operativo in una macchina virtuale ed avviare software in un ambiente considerato più "protetto" rispetto alla macchina host vera e propria. Come si può immaginare, al di là della lentezza - comunque relativa e proporzionale alla potenza della macchina host -, non vi è alcun limite. Questi sistemi a volte emulano anche parti di hardware, e altre volte si limitano a replicare l'hardware della macchina host. Una *Virtual Machine* solitamente è posizionata all'interno di un sistema di emulazione. Quest'ultimo aspetto verrà ripreso più volte durante il corso del lavoro proposto.

1.1.2 Ambienti di emulazione a confronto

Negli ultimi anni si sono affermati diversi strumenti ed ambienti di emulazione. Questi possono essere caratterizzati sulla base delle tecniche da essi adottate, dei tipi di dispositivi potenzialmente emulabili, della licenza con cui sono distribuiti, e delle diverse funzionalità offerte.

Di seguito andremo a descrivere brevemente alcuni dei più rappresentativi in materia di emulazione di reti.

Imunes [1] è un software di emulazione che si avvale di una estensione del kernel *FreeBSD* capace di eseguire più instanze indipendenti dello stack di rete su un unico kernel del sistema operativo. Un editor grafico della

topologia di rete consente di preparare rapidamente esperimenti costituiti da apparati di rete quali hub, switch, router e host.

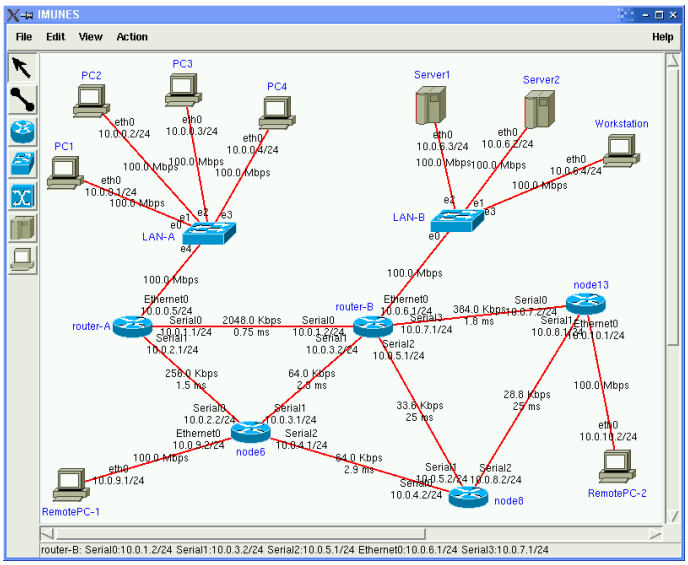


Figura 1.1: GUI per il software di emulazione IMUNES.

Marionnet [2] è anch'esso un ambiente di emulazione di reti virtuali. Permette all'utente di definire, configurare ed instanziare reti di computer anche complesse senza alcun bisogno di apparati fisici. Scritto in *OCaml* ed appoggiandosi su *User Mode Linux*[4] e *VDE*[5], offre una interfaccia grafica molto intuitiva.

NetKit[7] e *VNUML*[5] sono entrambi emulatori di medie dimensioni

[4]*User Mode Linux* (UML) dà la possibilità di avviare varie *Virtual Machine* con sistema *Linux* (sistema "Guest") per eseguire applicazioni come quello che accade con un normale sistema *Linux* (sistema "Host"). Ogni sistema guest è un normale processo avviato in *user space*.

[5]VDE è un rete virtuale compatibile con ethernet che può essere configurata su una rete di computer fisici su Internet. VDE fa parte del progetto *VirtualSquare*.

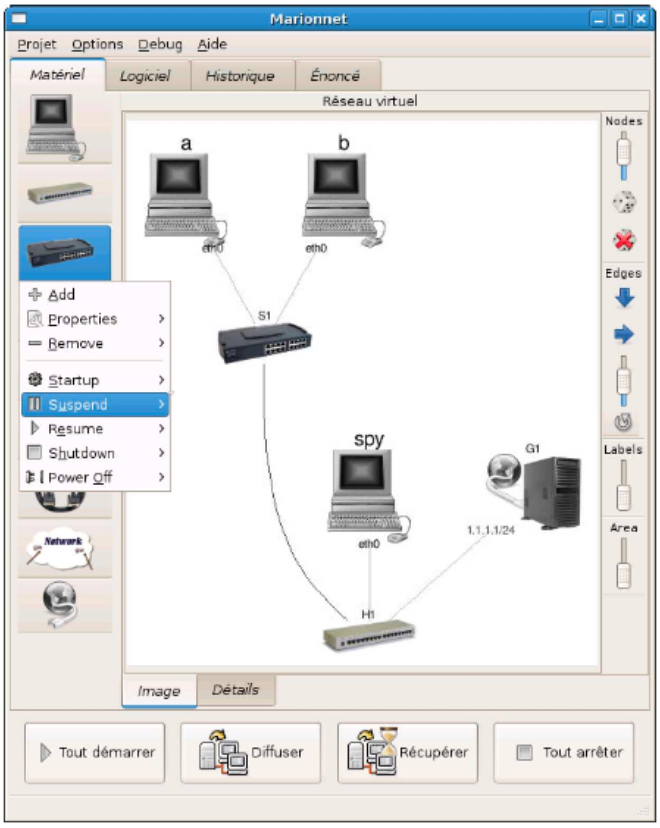


Figura 1.2: Finestra principale di Marionnet mostra una rete composta da tre computer, uno switch un hub ed un gateway.

che utilizzano un kernel *User Mode Linux* e permettono di avere importanti esperienze in ambito di reti emulate anche complesse.

Netkit In *NetKit* una rete viene modellata come la connessione di macchine virtuali. Per la creazione di un laboratorio[6] è necessario creare un file di configurazione per il Lab stesso, in cui si descrive la topologia della rete, e alcuni file e directory per ogni virtual machine che si vuole connettere

[6]Un laboratorio (o Lab) è uno scenario reale di una rete emulata.

alla rete. In figura 1.3 viene riportato un esempio di struttura di un *NetKit* laboratory.

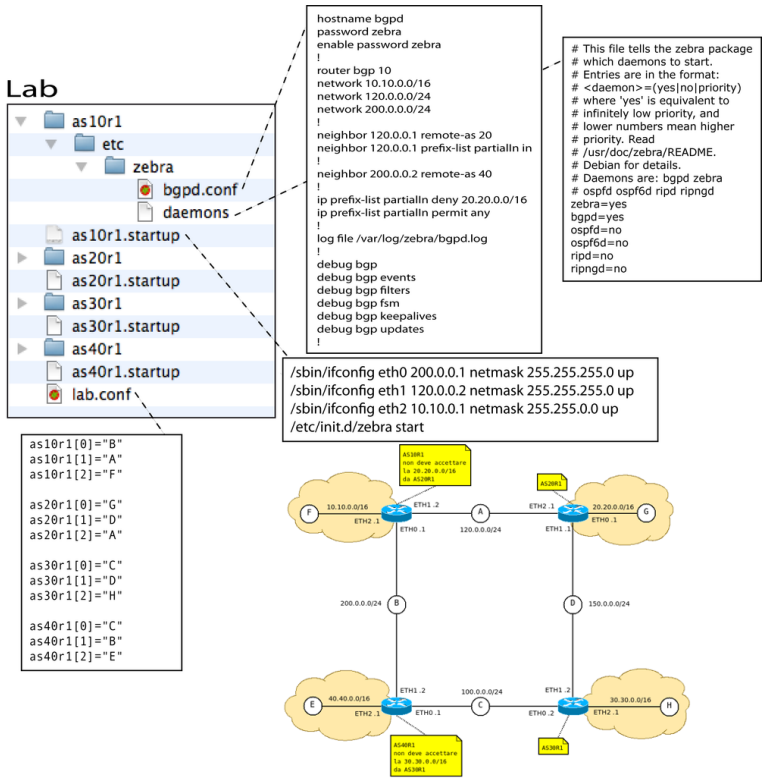


Figura 1.3: Schema di rete di un semplice lab in *NetKit* con relativa configurazione.

L'interazione tra l'utente finale e le macchine virtuali è resa possibile da una interfaccia a riga di comando tramite terminali emulati (*Linux* shell), come ci si comporta davanti ad una sessione *SSH*[7].

Vnuml [5] (Virtual Network User Mode Linux) è un ambiente di emulazione di reti che raccoglie una serie di script utilizzati per la descrizione di

[7]SSH (Secure SHell, shell sicura) è un protocollo che permette di stabilire una sessione remota cifrata ad interfaccia a linea di comando con un altro host.

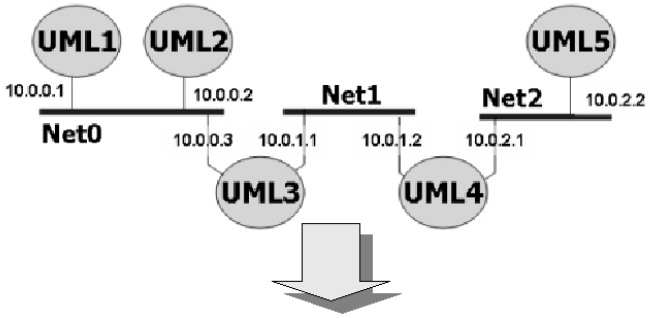
testbed in *XML*, per il test di applicazioni di rete e servizi.

VNUML consiste in un linguaggio basato su *XML*, con una specifica sintassi essendo un interprete che può essere utilizzato per descrivere ed eseguire una rete emulata di macchine virtuali. Dalla descrizione in *XML* degli scenari, VNUML instanzia le macchine virtuali e le configura nascondendo all'utente i dettagli avanzati e la configurazione virtuale delle macchine. Questo ambiente risulta molto utile per testare reti anche complesse. Comunque, i progettisti devono descrivere il completo scenario di rete utilizzando il linguaggio *XML* che, a volte, richiede dettagli molto specifici.

Questo ambiente viene affiancato da *VNUMLGUI*[6], allo scopo di introdurre una rappresentazione grafica intuitiva. *VNUMLGUI* è un'interfaccia grafica per VNUML, e soprattutto un editor di topologia con capacità grafiche. Permette infatti di creare una qualsiasi topologia di rete virtuale in VNUML senza dover editare manualmente il file *XML*, ma con la sola immissione di router e interruttori. Così come VNUML nasconde la curva di apprendimento di UML, *VNUMLGUI* nasconde la curva di apprendimento di VNUML e *XML*.

Di seguito - figura 1.4 - sono riportati a titolo di esempio uno schema di rete e la sua corrispettiva implementazione in VNUML, ed una raffigurazione dell'interfaccia grafica di *VNUMLGUI* - figura 1.5.

Sia *NetKit* che VNUML sono emulatori realizzati per offrire all'utente una rappresentazione ad alto livello della rete modellata. *NETGUI* e *VNUMLGUI* concretizzano tale scopo. Tutto ciò conferma l'intuizione posta alla base di questi progetti e l'importanza di un supporto grafico facile da usare.




```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE vnuml SYSTEM "/usr/share/xml/vnuml/vnuml.dtd">
<vnuml>
  <global>
    <version>1.7</version>
    <simulation_name>tutorial-lu</simulation_name>
    <automac/>
    <vm_mgmt type="none" />
    <vm_defaults>
       <filesystem type="cow">
          /usr/share/vnuml/filesystems/root_fs_tutorial
       </filesystem>
       <kernel>/usr/share/vnuml/kernels/linux</kernel>
       <console id="0">xterm</console>
    </vm_defaults>
  </global>
  <net name="Net0" mode="uml_switch" />
  <net name="Net1" mode="uml_switch" />
  <net name="Net2" mode="uml_switch" />
  <vm name="uml1">
    <if id="1" net="Net0">
      <ipv4>10.0.0.1</ipv4>
    </if>
    <route type="ipv4" gw="10.0.0.3">default</route>
  </vm>
  <vm name="uml2">
    <if id="1" net="Net0">
      <ipv4>10.0.0.2</ipv4>
    </if>
    <route type="ipv4" gw="10.0.0.3">default</route>
  </vm>
...
</vnuml>
```

Figura 1.4: Schema di rete di un semplice scenario in VNUML con relativa configurazione.

Qemu [4] è un emulatore e virtualizzatore di macchine che utilizza una traduzione dinamica al fine di ottenere una buona velocità di emulazione. QEMU offre due modalità operative: full system emulation ed user mode emulation, entrambe accessibili tramite interfaccia a riga di comando. Nella prima modalità QEMU emula un sistema completo (ad esempio un PC),

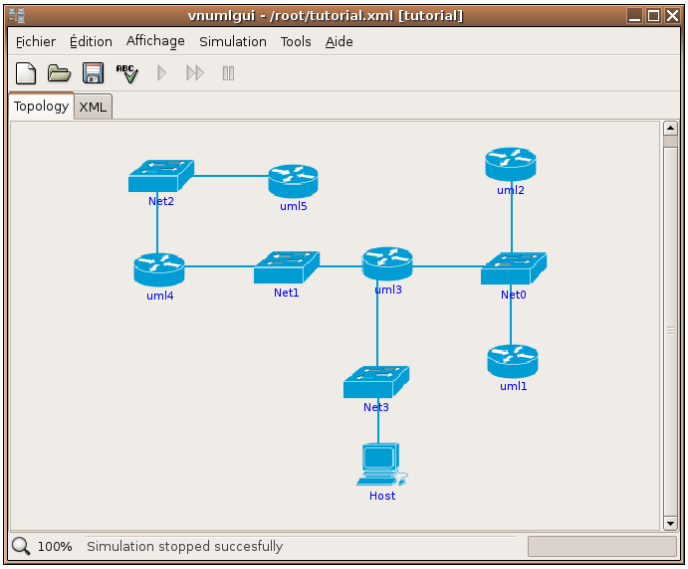


Figura 1.5: Schema di rete realizzato in *VNUMLGUI*

tra cui uno o più processori e le varie periferiche. Nella seconda modalità QEMU può avviare processi compilati per una *CPU* su un'altra *CPU*.

QEMU è capace di simulare diverse VLAN[8]. Una VLAN può essere rappresentata da un collegamento virtuale tra diversi dispositivi di rete quali ad esempio schede Ethernet virtuali QEMU o dispositivi Ethernet su host virtuali.

1.2 Ambienti per la configurazione di Reti Virtuali

In questa sezione saranno paragonate le interfacce grafiche (e quindi gli "ambienti di configurazione") che i vari sistemi di emulazione descritti pre-

[8]Il termine VLAN (Virtual LAN) indica un insieme di tecnologie che permettono di segmentare il dominio di broadcast che si crea in una rete locale (tipicamente IEEE 802.3) basata su switch, in più reti non comunicanti tra loro.

cedentemente mettono a disposizione. Quindi verranno messe a confronto le varie interfacce grafiche di: *Imunes*, *MarionNet*, *VnUMLGUI* e la versione 1.0 di *VisualNetkit*. In particolare si vuole dare al lettore una panoramica completa su come questi ambienti offrono flessibilità e configurabilità al cospetto di configurazioni complesse.

1.2.1 Configurazioni di Reti Complesse

Iniziamo con il parlare in *Imunes* e di come sia possibile creare configurazioni complesse[9]. Questo ambiente possiede una GUI piuttosto semplice scritta in *Tcl/Tk*[10], la quale offre la possibilità di disegnare una rete a livello topologico inserendo nodi virtuali come Hosts, Switches e Routers. Tale interfaccia utente è considerata più un front-end verso l'interfaccia a riga di comando che ogni nodo possiede.

Da quanto visto[3], l'interfaccia grafica che *Imunes* fornisce non offre un adeguato supporto alle configurazioni complesse; in particolare su ogni host occorre effettuare i vari settaggi con il minimo supporto grafico. Viene offerta la possibilità di impostare il tipo di Router (Statico o con software *Quagga*) senza però poter configurare le varie politiche di routing. Viene inoltre data la possibilità di settare indirizzi IPv4 nei links, rotte statiche

[9]Per configurazione "complessa" si intende una configurazione di un qualche servizio all'interno di una *Virtual Machine*, come ad esempio un firewall, un DNS, un server HTTP, un servizio di Routing Interdomain, ecc...

[10]*Tcl* (acronimo di Tool Command Language), è un linguaggio scripting creato da John Ousterhout generalmente considerato di facile apprendimento (rispetto ai linguaggi della sua generazione) ma allo stesso tempo potente. L'estensione *Tk* è un insieme di strumenti per scrivere GUI (un toolkit di widget) implementato dallo stesso autore di *Tcl*.

nei vari nodi e negli switches è possibile settare parametri quali uso della cpu, politica e dimensione delle code (Queue), ecc...

MarnionNet utilizza un'interfaccia grafica ben più potente di *Imunes*, per mezzo della quale l'utente può disegnare[2] la topologia di rete desiderata inserendo:

Computers per ognuno di questi è possibile configurare l'ammontare di memoria RAM e il numero di Ethernets/Serial ports con la possibilità (per le porte Ethernet) di settare indirizzi MAC, IPv4/6, MTU, e nome (figura 1.6);

Hubs e Switches per ogni hub o switch l'utente può specificare il numero di porte e (per gli switches) il supporto al protocollo *STP*[11];

Routers è possibile inserire routers che usano *Quagga*[12] come software per il routing statico e dinamico, non offrendo però la possibilità di gestire graficamente le varie configurazioni dei servizi e dei protocolli utilizzati;

Coulds *MarionNet* offre anche la possibilità di inserire "nuvole" (aggregatori di elementi) con due end-points;

External Socket connessioni verso il sistema Host.

Il tutto quindi risulta sviluppato staticamente all'interno del sistema grafico, e di conseguenza comporta una rigidità troppo pronunciata per poter

[11] Lo *Spanning Tree Protocol* è un protocollo che previene cicli in una topologia di rete LAN switchata.

[12] *Quagga* è un derivato del progetto *Zebra* (www.zebra.org) che permette di utilizzare protocolli quali BGP, RIP, OSF e ISIS

essere di effettivo supporto alle configurazioni avanzate dei vari elementi. Basti pensare che i servizi presenti all'interno di ogni Host virtuale vanno configurati senza nessun supporto da parte di *MarionNet*.

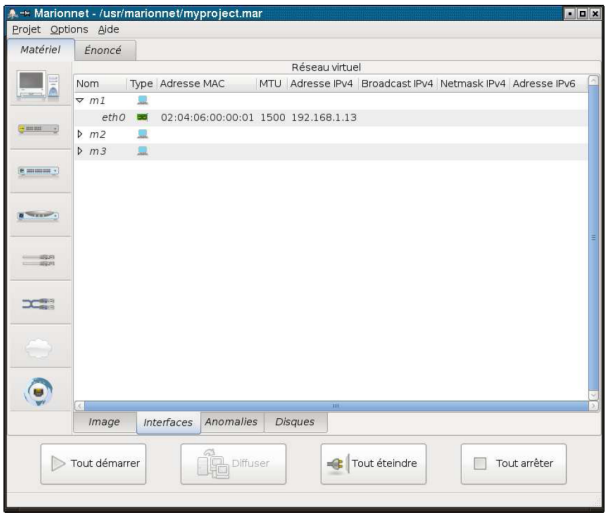


Figura 1.6: Setting dei parametri delle interfacce ethernet in *MarionNet*.

Per quanto riguarda *VnUMLGUI* si può affermare che tale interfaccia utente è la più semplice tra quelle sperimentate. Questa offre solamente la possibilità di disegnare la topologia della rete e ricavarne un file *XML* che utilizzerà in seguito VNUML per la simulazione effettiva. VNUMLGUI, composto da poche righe di codice in *Perl*, risulta tuttavia un ottimo strumento da affiancare a VNUML, poiché la scrittura della topologia tramite un file *XML* è spesso una pratica onerosa e non banale.

La prima versione di *VisualNetkit*

VisualNetkit è un ambiente grafico per la creazione di reti virtuali, inteso come un naturale punto di partenza per la progettazione di un ambiente di configurazione. *VisualNetkit*. Si tratta di un tool grafico che permette di

creare laboratori per *NetKit* disegnando la topologia della rete che si vuole emulare, e allo stesso tempo offre un buon supporto alla configurazione dei vari nodi e archi del grafo. Per nodi, in questo caso, si intendono hosts virtuali e domini di collisione e per archi, links che connettono interfacce ethernet dei singoli host con uno dei domini di collisione (figura 1.7).

Figura 1.7: Una semplice rete virtuale in *VisualNetkit*.

Questo tool, a differenza degli altri analizzati precedentemente, è stato creato attorno al concetto di flessibilità; *VisualNetkit* è stato realizzato infatti al di sopra di una struttura a *Plug-In*, i quali hanno il compito di arricchire la rete a livello di configurazioni e componenti attivi sui vari elementi che costituiscono la topologia del grafo. Questo ambiente (allo stato attuale) non consente di avviare direttamente i laboratori che produce in quanto si è voluto creare un sistema autonomo indipendente da *NetKit* stesso.

Parlando più in dettaglio della sua struttura modulare, si può dapprima osservare che *VisualNetkit* senza alcun *Plug-In* attivo offre solamente la possibilità di costruire la topologia della rete virtuale che si vuole sperimentare; è quindi possibile inserire hosts, domini di collisione e links di connessione. Tutto ciò permette a *VisualNetkit* di creare un laboratorio funzionante in tutti i suoi aspetti, ma privo di ulteriori configurazioni. Va detto inoltre che il tool in questione può essere potenzialmente utilizzato come strumento per la creazione di laboratori di altri sistemi di emulazione differenti da *NetKit*. Per avere una visione più chiara si osservi la figura 1.7.

È possibile osservare che la topologia è composta da tre elementi base: un host, un dominio di collisione e un link. Analizzando quest'ultimo si può notare che vi è un dettaglio che va oltre la semplice topologia di rete; sul link è stato attivato un *Plug-In* di tipo IPv4. Questo permette al link stesso di arricchire il proprio bagaglio informativo aggiungendo in questo caso la possibilità di settare ulteriori parametri (propriamente contenuti all'interno del *Plug-In*) quali *address*, *netmask* e *broadcast*. Chiaramente queste informazioni aggiuntive dovranno essere salvate all'interno del laboratorio che si sta creando; in questo caso il sistema prevede che ogni *Plug-In* possa fornire un numero arbitrario di "contributi", ossia porzioni di file di testo (e relativo percorso del file interessato) che costituiranno il file di configurazione per quel determinato modulo. In figura 1.8 è possibile osservare come il *Plug-In* IPv4 attivo sul link sia il diretto responsabile del contenuto del file di configurazione della macchina "Host".

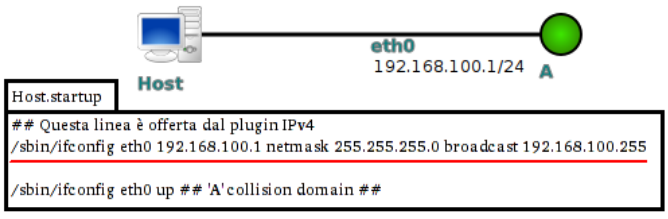


Figura 1.8: Relativo file di configurazione della rete in figura 1.7.

Come detto, ogni *Plug-In* offre informazioni aggiuntive all'elemento base a cui è stato collegato. Scendendo nel dettaglio si può affermare che i vari *Plug-In* offrono un insieme di proprietà descritte da una lista di coppie chiave-valore. Nel nostro esempio, il *Plug-In* IPv4 offre una lista di tre "property":

- **address** - l'indirizzo ipv4;
- **netmask** - la netmask associata;
- **broadcast** - l'indirizzo broadcast.

Queste proprietà sono comunque statiche all'interno dei vari *Plug-In* e l'utente può solamente limitarsi a modificarle previo controllo di errori non vincolante. È stato quindi rilevato uno dei punti deboli che la prima versione di *VisualNetkit* possiede. Anche se questa è basata su una struttura modulare e flessibile, i *Plug-In* stessi limitano tale flessibilità rimanendo rigidi per offrire un buon supporto in situazioni dove sono necessarie configurazioni avanzate.

Abbiamo appena anticipato un problema che durante il seguente lavoro tratteremo più in dettaglio, mostrando come la nuova versione del tool grafico offra maggior flessibilità e risolva in gran parte il problema appena accennato. In figura 1.9 viene mostrato l'ambiente grafico offerto da *VisualNetkit* versione 1.0.

Flessibilità delle interfacce grafiche nei sistemi esistenti

Il paragrafo precedente mostra come l'avere un ambiente grafico di supporto sia altrettanto importante del supporto offerto dal sistema di emulazione stesso. Avere infatti uno strumento potente per creare i nostri laboratori, ma che possegga specifiche di creazione complicate e poco intuitive, può indurre l'utente ad evitare di addentrarsi in particolari configurazioni avanzate poiché vi è un forte rischio di dispendio di energie e di un elevato tempo di creazione. È qui che entrano in gioco i tools grafici poc'anzi descritti.

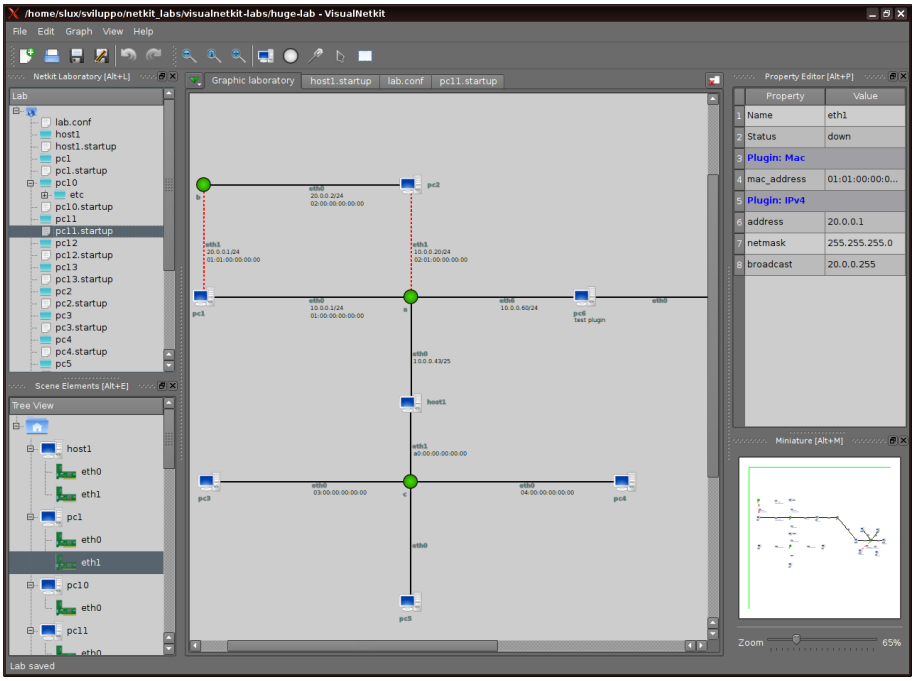


Figura 1.9: Un laboratorio creato con *VisualNetkit* versione 1.0.

Riassumendo, è possibile affermare che tra le interfacce grafiche analizzate fin'ora, nessuna riesce ad offrire un supporto totale; ognuna di esse infatti, prevede una parte di configurazione manuale. L'utente che vuole addentrarsi in configurazioni particolari deve necessariamente agire direttamente sui vari file di configurazione dei nodi della rete o, in alcuni casi, deve intervenire interattivamente tramite le interfacce a riga di comando previste per ogni *Virtual Machine*.

Di seguito (tabella 1.10) viene proposta una tabella riassuntiva che paragona i vari strumenti.

	Componenti base	**Configurazione base**	**Configurazione avanzata**	**Linguaggio**	**Interfaccia (Shell) diretta con i nodi**
VisualNetkit	Host, Collision Domain, Link, Aree	Ipv4, Mac (plugin di base)	Tramite plugins avanzati, a partire dalla versione *1.1*	C++/Qt4	No
MarionNet	Host, Hub/Switch, Router, Link, Coulds, External Socket	Ipv4, Mac, Ipv6, Spanning Tree Protocol (*STP*)	Possibilità di inserire solo la tipologia del tipo di router ES: *Quagga*.	OCaml	Si
VnUmlGui	Host, Hub/Switch, Router, Link	Ipv4, Static route	No. Solo manuale agendo sui vari nodi tramite SSH	Gtk2 - Perl	Si
Imunes	Host, Router, Switch, Link, Hardware Interface	Ipv4, Static route, Cpu usage, Bandwidth link, Switch politica delle code	Possibilità di inserire nei router il modello: *Statico* o con software *Quagga* e relative rotte statiche	Gtk2 - Perl	Si

Figura 1.10: Tabella comparativa delle interfacce grafiche analizzate.

Capitolo 2

Configurazione Avanzata di Reti Virtuali

Per “abbracciare il cambiamento”, le strutture, il design, devono seguire le funzionalità di una applicazione in modo continuo. In un mondo in cui il cambiamento è un fattore primario e spesso violento, per seguire le funzionalità, le strutture devono essere continuamente messe in discussione e rimodellate.

Francesco Cirillo.

Come introdotto nel capitolo 1, *VisualNetkit* offre un’ottima flessibilità a livello architetturale grazie alla sua struttura modulare che si appoggia su *Plug-In*; allo stesso tempo però questa malleabilità è limitata dal potere espressivo del singolo *Plug-In*. Infatti, quest’ultimo può offrire un contributo espresso sotto forma di una lista di coppie chiave-valore rappresentanti le informazioni che il *Plug-In* andrà ad inserire all’interno dei templates[1] che produrrà.

In questo capitolo discuteremo di come *VisualNetkit* sia stato profondamente modificato per dare la possibilità ai vari *Plug-In* di poter “abbraccia-

[1] Ogni “template” rappresenta il contenuto testuale che andrà scritto sul file di configurazione indicato dal *Plug-In* stesso; se più *Plug-In* vogliono scrivere sul medesimo file di configurazione, il sistema non fa altro che accodare i vari templates.

re" praticamente la totalità delle tipologie dei file di configurazione dei vari servizi (Dns, HTTP, E-Mail, Zebra, SSH, ecc...). Si discuterà il problema dapprima analizzando alcune configurazioni avanzate di BGP e OSPF ed in secondo luogo si cercherà di estrapolare una struttura da poter descrivere all'interno dei vari *Plug-In*. Successivamente si formalizzeranno i nuovi requisiti discutendo l'impatto di tali modifiche sul sistema attuale, ed in fine ci si addentrerà in uno studio di "Analisi Architetturale" del nuovo sistema di gestione delle properties dei *Plug-In*.

2.1 Scenari che Richiedono Configurazioni Avanzate

Quando ci si misura con servizi complessi come Zebra[2], quasi sempre si ha a che fare anche con file di configurazione dall'alto potere espressivo e quindi potenzialmente complessi.

Andremo ora ad analizzare da vicino due esempi di configurazioni complesse in Zebra, in particolare nei protocolli BGP[3] e OSPF[4], e cercheremo di trovare una possibile struttura comune che possa racchiuderli.

[2]GNU Zebra è un software open-source che gestisce i protocolli di routing basati su TCP/IP. Supporta il protocollo BGP-4, come descritto nell'RFC-1771, come anche RIPv1, RIPv2 e OSPFv2.

[3]Il *Border Gateway Protocol* (BGP) è un protocollo di rete usato per connettere tra loro più router che appartengono a sistemi autonomi distinti e che vengono chiamati gateway.

[4]Il protocollo *Open Shortest Path First* (OSPF) è uno dei protocolli di instradamento più diffusi, che gestisce le tabelle di instradamento di una rete IP con il metodo del Link State.

2.1.1 Configurazione avanzata in BGP

In questa sezione si cercherà di individuare una struttura comune in uno scenario reale, prendendo come esempio la configurazione BGP proposta in figura 2.1.

```
1  ! A Zebra bgpd.conf complex file
2  !
3  hostname bgpd
4  password zebra
5  enable password zebra
6  !
7  router bgp 2
8  bgp router-id 20.0.3.2
9  network 100.0.0.1/32
10 network 20.0.3.0/24
11 network 20.0.2.0/24
12 network 20.0.4.0/24
13 network 20.0.5.0/24
14 !
15 neighbor 20.0.3.1 remote-as 1
16 neighbor 20.0.3.1 description Machine as1m2 (Peer)
17 neighbor 20.0.3.1 prefix-list acceptAny in
18 !
19 neighbor 20.0.2.4 remote-as 4
20 neighbor 20.0.2.4 description Machine as4m2 (Customer)
21 neighbor 20.0.2.4 default-originate
22 neighbor 20.0.2.4 prefix-list defaultOut out
23 neighbor 20.0.2.4 prefix-list acceptAny in
24 !
25 neighbor 20.0.4.5 remote-as 5
26 neighbor 20.0.4.5 description Machine as5m1 (Customer)
27 neighbor 20.0.4.5 default-originate
28 neighbor 20.0.4.5 prefix-list defaultOut out
29 neighbor 20.0.4.5 prefix-list acceptAny in
30 !
31 neighbor 20.0.5.6 remote-as 6
32 neighbor 20.0.5.6 description Machine as6m1 (Customer)
33 neighbor 20.0.5.6 default-originate
34 neighbor 20.0.5.6 prefix-list defaultOut out
35 neighbor 20.0.5.6 prefix-list acceptAny in
36 !
37 ip prefix-list acceptAny permit any
38 ip prefix-list defaultOut permit 0.0.0.0/0
39 !
40 log file /var/log/zebra/bgpd.log
41 !
42 debug bgp
43 debug bgp events
44 debug bgp filters
45 debug bgp fsm
46 debug bgp keepalives
47 debug bgp updates
```

Figura 2.1: Una configurazione complessa di BGP.

Osservando la struttura del file di configurazione proposto, notiamo subito che vi è una struttura comune che può essere estrapolata e classificata. Senza considerare le righe $1 \mapsto 8$, che sono riconducibili ad una semplice lista di coppie chiave-valore, è opportuno soffermarci alle righe $9 \mapsto 13$; qui si trovano le network annunciate dal router in questione e può già essere dedotto che tale struttura è una lista con cardinalità 0..n di coppie con chiave "network" e con valore uguale all'indirizzo IP - seguito dalla netmask - che si vuole annunciare. Già in questo scenario una coppia chiave-valore (usata nella versione 1.0 di *VisualNetkit*) non può essere utilizzata poiché solitamente le chiavi devono rimanere univoche.

Si osservino ora le righe $15 \mapsto 35$ analizzando in particolare le righe $19 \mapsto 23$. Come descritto nella documentazione di Zebra[8], un "peer" ha la seguente sintassi:

neighbor *peer* **remote-as** *AS-Number*
neighbor *peer* **COMMAND**

dove **COMMAND** è uno dei comandi previsti da Zebra, come ad esempio *description, default-originate, interface, version,* ecc. . . , nonché comandi atti al *Peer Filtering* quali *distribuite-list, prefix-list, route-map,* ecc. . .

Pertanto, anche in questo caso è possibile raggruppare le varie definizioni dei "vicini" (neighbor) in una struttura gerarchica in cui ogni neighbor ha una struttura composta da sotto-proprietà, eventualmente con proprietà che si riferiscono a nodi esterni, come nel caso *prefix-list* (riga $22-23$). Proprio partendo da queste due righe di codice, è quindi possibile identificare

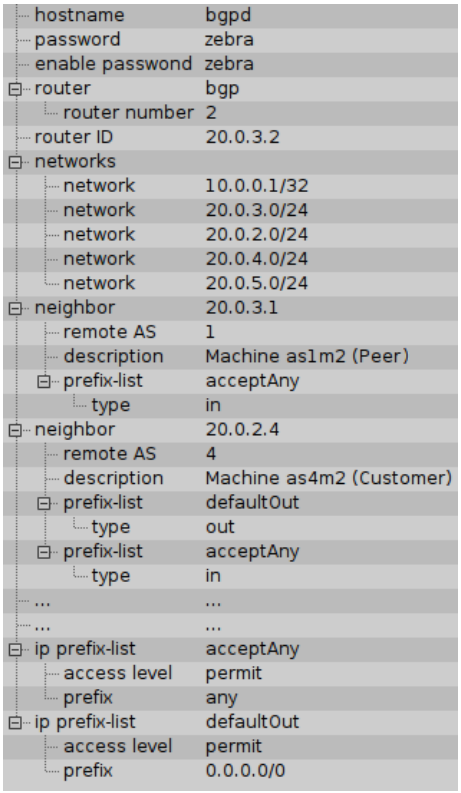


Figura 2.2: Configurazione complessa di BGP con struttura gerarchica.

proprietà correlate ad entità esterne (concetto simile alle chiavi esterne in uno schema relazionale di basi di dati); in definitiva si sta affermando che quella in questione non è nient'altro che una struttura ad albero n-ario che possiede un enorme potere descrittivo, ma allo stesso tempo una struttura complessa da gestire e manipolare. In figura 2.2 viene mostrata la mappatura del file di configurazione in esame (figura 2.1) in un albero n-ario.

Questa non è che una delle tante possibili interpretazioni di un file di configurazione in una struttura gerarchica. Solitamente ogni file di configurazione (soprattutto nei sistemi *Unix like*) possiede una struttura che

è riconducibile ad una con caratteristiche gerarchiche. L'evoluzione che *VisualNetkit* ha avuto si è mossa proprio in questa direzione, in particolare trasformando il vecchio modello chiave-valore delle proprietà dei *Plug-In*, in uno altamente dinamico (con la possibilità di inserire e/o eliminare proprietà) con struttura annidata.

2.1.2 Configurazione avanzata in OSPF

Si tenterà di applicare quanto detto poc'anzi ad un altro scenario reale che coinvolge il protocollo OSPF ed il suo file di configurazione. Prendiamo dunque in esame il file di configurazione mostrato in figura 2.3.

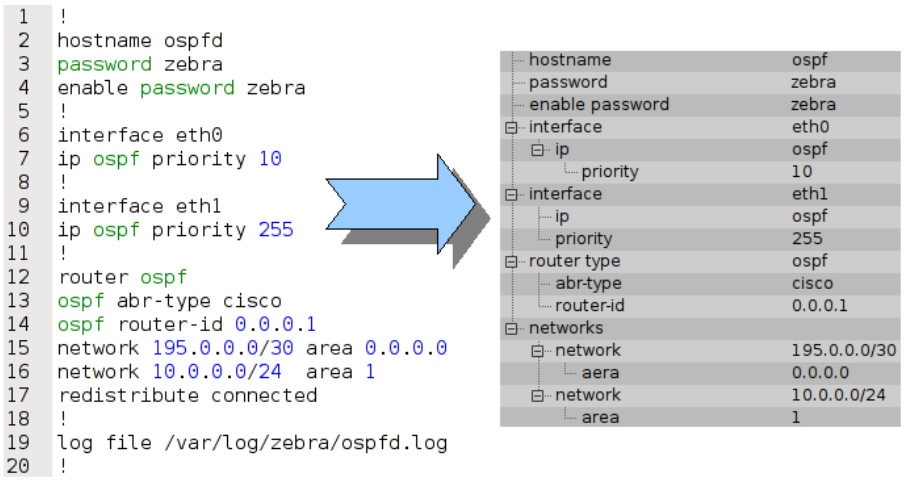


Figura 2.3: Configurazione di OSPF e relativa vista gerarchica.

Anche in questo caso si può procedere nel tentativo di trasformare il contenuto del file di configurazione proposto, in una struttura descritta da un albero n-ario. È opportuno quindi iniziare ad analizzare il testo soffermandosi sulle righe $6 \mapsto 10$; si nota subito come questa porzione abbia una struttura abbastanza uniforme - come descritto nella documentazione[8] -,

che può essere mappata all'interno di una struttura più auto-descrittiva e gerarchica (figura 2.3).

Analizzando poi le righe $15 - 16$ (tralasciando le altre) è possibile ricavare una struttura ben precisa, che nella documentazione di Zebra viene presentata nel seguente modo:

network *a.b.c.d/m* **area** *a.b.c.d*
network *a.b.c.d/m* **area** *<0-4294967295>*

Naturalmente questi scenari sono soltanto alcuni dei tanti possibili contenuti che si possono trovare all'interno delle varie configurazioni, tuttavia è stato appena dimostrato che qualunque siano le regole presenti nelle varie impostazioni dei servizi utilizzati, si riesce sempre a ricondurre questi ultimi ad una rappresentazione gerarchica, talvolta anche complessa.

2.2 Requisiti per un Ambiente di Configurazione Avanzata

Prima di trasformare il sistema in modo da essere riadattato a quanto detto fin'ora, è preferibile formalizzare i nuovi requisiti - sia quelli funzionali, che non - per avere un quadro complessivo e non ambiguo su quello che il nuovo sistema dovrà offrire.

Si è cercato di individuare gli attori principali discutendo con gli *stakeholders* per avere più punti vi vista. Gli attori individuati sono due:

- l'utente che utilizza *VisualNetkit*, in particolare uno dei suoi *Plug-In*;

- l'utente/sviluppatore che desidera realizzare un *Plug-In* dalle caratteristiche avanzate.

Sono stati quindi definiti una serie di scenari principali di successo per ciascun caso d'uso semplificato. L'insieme degli scenari ritenuto più importante verrà presentato in seguito. Con il termine "end user" si denota l'utente che utilizza *VisualNetkit*, e con il termine "plugin developer" colui che vuole realizzare un *Plug-In*.

Caso d'uso - Inizializzazione dei *Plug-In* selezionati

Scopo: applicazione *VisualNetkit*
Livello: user goal
Attore Primario: End user
Parti interessate e interessi:

- End user: desidera un interazione semplice, veloce ed intuitiva con il sistema per raggiungere i propri obiettivi con il minimo sforzo.

Prerequisiti: il sistema deve essere avviato e l'utente deve aver creato un nuovo Lab.
Goal: l'utente ha creato un elemento base (una virtual machine, un collision domain o un link) e aver scelto ed inizializzato i *Plug-In* che ha selezionato. Il sistema mostra sulla scena grafica l'elemento creato.

Scenario di successo:

1. l'utente seleziona dalla tool-bar o dal menu la tipologia dell'elemento che intende aggiungere;
2. l'utente clicca con il mouse - tasto sinistro - un punto della scena grafica e il sistema provvede a mostrare la form per l'inizializzazione dei parametri;
3. l'utente completa la form attivando inoltre i *Plug-In* che desidera attivare;
4. qualora l'utente voglia modificare i valori di default dei *Plug-In* selezionati, il sistema mostra una successiva form che offre la possibilità di mutare i vari campi delle property, nonché la possibilità di modificare la struttura delle stesse tramite l'apposito bottone "actions";
5. l'utente accetta e il sistema provvede a chiudere la form;
6. il sistema inizializza ed aggiorna il suo stato aggiungendo, l'elemento selezionato e mostrandolo all'utente.

Caso d'uso - Modifica delle proprietà di un elemento

Scopo: applicazione *VisualNetkit*
Livello: user goal
Attore Primario: End user
Parti interessate e interessi:

- End user: desidera un'interazione semplice, veloce ed intuitiva con il sistema, per raggiungere i propri obiettivi con il minimo sforzo.

Prerequisiti: il sistema necessita di essere avviato, l'utente deve aver creato un nuovo Lab e deve essere presente almeno un elemento base.
Goal: l'utente è riuscito con successo a modificare - nel contenuto o nella struttura - una delle proprietà di un elemento selezionato. Il sistema ha provveduto all'acquisizione dei cambiamenti, modificando le proprie strutture interne.
Scenario di successo:

1. l'utente clicca due volte con il mouse un elemento presente nella scena grafica, oppure clicca un elemento mostrato nell'insieme degli oggetti presenti, selezionandolo;
2. il sistema mostra nella "property dock" le proprietà dell'elemento selezionato catalogate e suddivise in base al loro ruolo: proprietà costitutive dell'elemento base e proprietà offerte dai *Plug-In* attivi;
3. l'utente modifica il contenuto di una proprietà. Il sistema provvede a validare il valore inserito e a registrare i cambiamenti effettuati, eventualmente aggiornando gli elementi grafici;
4. l'utente intende modificare la struttura delle proprietà di uno dei *Plug-In* presenti:
   (a) l'utente desidera aggiungere una proprietà o sotto-proprietà dopo averne selezionata un'altra. Tramite l'apposito bottone "actions" questi può inserire altre sotto-proprietà che automaticamente il sistema propone come possibili candidate. Dopo aver inserito una nuova proprietà, il sistema registra il cambiamento al suo interno;
   (b) l'utente desidera eliminare una proprietà dopo averla selezionata. Tramite il bottone "actions" questi seleziona "elimina proprietà" e il sistema (validando o meno l'operazione) provvede a modificare le proprie strutture interne.

Abbiamo analizzato i due principali scenari di successo che l'utente finale si dovrebbe aspettare, ora ci preoccuperemo di analizzare altri scenari di successo che riguardano le aspettative dell'utente (plugin developer). In particolare gli scenari indicano come lo sviluppatore può usufruire della possibilità di impostare configurazioni avanzate per il servizio che egli stesso intende descrive.

Caso d'uso - Creazione di *Plug-In* avanzati

Scopo: *Plug-In* per *VisualNetkit*
Livello: sub-function
Attore Primario: plugin developer
Parti interessate e interessi:

- Plugin developer: si aspetta di riuscire a creare il proprio *Plug-In*, che dovrà contenere una struttura flessibile tale da poter descrivere la maggior parte delle configurazioni avanzate di un certo servizio offerto dal *Plug-In* stesso.

Goal: lo sviluppatore realizza un *Plug-In* per un determinato servizio che al suo interno possiede flessibilità e alta adattabilità, in modo da offrire anche configurazioni complesse.
Scenario di successo:

1. lo sviluppatore costruisce il file di configurazione per il suo *Plug-In* descrivendo in modo dettagliato la struttura delle proprietà;
2. lo sviluppatore crea il proprio *Plug-In* che offrirà agli utenti finali la possibilità di una particolare estensione dell'elemento base su cui si fonda il *Plug-In*;
3. il sistema provvede a caricare il *Plug-In* durante l'avvio e ad offrire all'utente finale la possibilità di selezionarlo.

In figura 2.4 viene proposto il diagramma completo dei casi d'uso e l'interazione tra di essi.

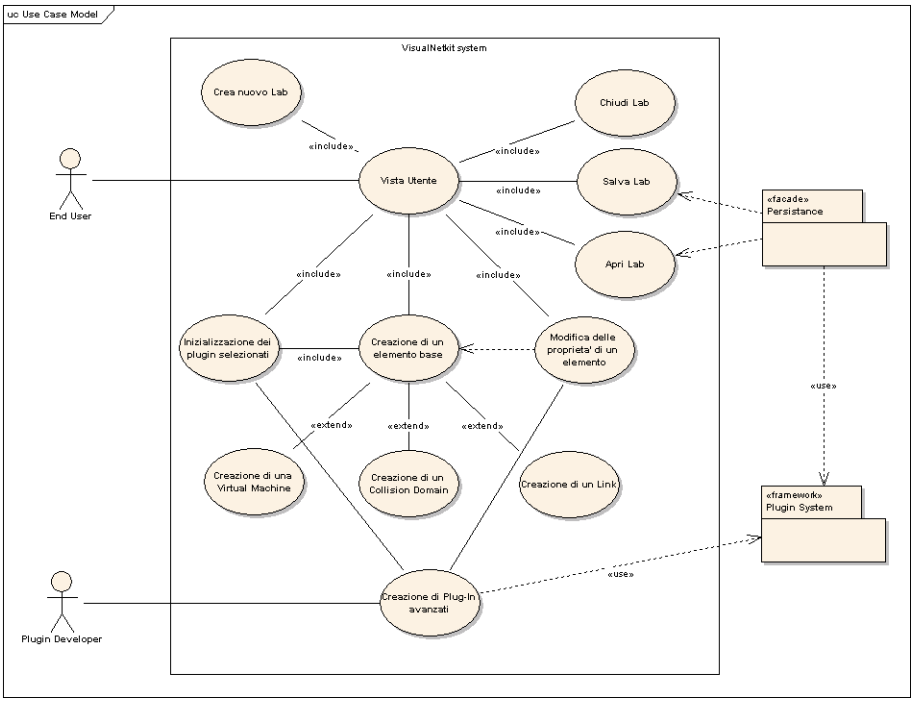


Figura 2.4: Diagramma dei casi d'uso principali in *VisualNetkit*.

L'impatto sul sistema

Verrà ora analizzato in dettaglio quale sarà l'impatto sul sistema che prevede properties unicamente descritte da una lista di coppie chiave-valore. Si sta di fatto introducendo un cambiamento piuttosto radicale che andrà a coinvolgere da una parte il core di *VisualNetkit* e dall'altra il framework che supporta l'architettura modulare. Scendendo nei particolari ci si dovrà muovere con una strategia *Bottom-Up* effettuando un restyling del *Plug-In* framework, introducendo nuove funzionalità al sistema. Queste ultime interagiranno con i moduli per interrogare le loro proprietà[5] e per validare

[5]Si ricorda che le proprietà saranno contenute in strutture ad alberi n-ari, non facilmente gestibili.

ed interpellare il loro file di configurazione *XML*.

Successivamente si andrà a rimodellare la GUI che sarà dotata di una nuova property dock. Anche il core stesso dell'applicazione dovrà essere rivisitato; i vari handler che raccolgono le modifiche effettuate dall'utente all'interno delle proprietà, andranno migliorati e correlati tra loro. In tutto questo giocherà un ruolo fondamentale il potente framework *Qt*, basato pesantemente sul pattern architetturale *MVC*[9].

2.3 Analisi Architetturale

In questa sezione si raccoglieranno tutti i requisiti descritti precedentemente e, rispettando la tassonomia dell'attuale sistema, si cercherà di formalizzare le modifiche che verranno poi applicate su *VisualNetkit*. Una prima fase di analisi è stata spesa per carpire adeguatamente il tipo di supporto offerto da *Qt*, in particolare i concetti su cui si basa il sistema *Model/View*.

L'architettura Model/View in *Qt*

Model-View-Controller (MVC) è un pattern apparso per la prima volta nel 1971 all'interno del linguaggio di programmazione *Smaltalk*, spesso applicato a scenari in cui l'interfaccia utente gioca un ruolo chiave. *Qt* separa le varie classi all'interno di tre gruppi: models, views e delegates. Ognuno di questi componenti è definito da una classe astratta comune che offre implementazioni di default di uso generico.

Tutti i modelli sono basati sulla classe *QAbstractItemModel*. Questa definisce un'interfaccia che viene usata dalla vista (View) attraverso il delegato, per accedere ai dati. I dati stessi non sono presenti nel modello, ma ven-

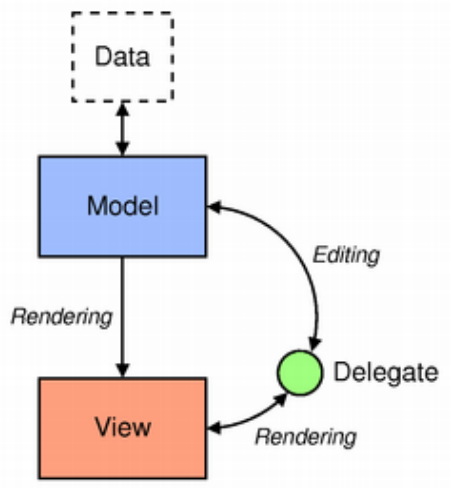


Figura 2.5: MVC all'interno di *Qt*4.

gono immagazzinati in strutture separate che possono essere classi, files, database o altri tipi di contenitori. Le viste invece implementano la parte visiva dei dati presentata all'utente finale. *Qt* offre tre tipologie di viste utilizzate, a seconda dei casi: Tabelle, Liste o Alberi. In particolare, ogni vista renderizza (tramite il delegato) le informazioni presenti nelle strutture dati gestite dal modello, e questo offre la possibilità ad ogni modello di essere visualizzato all'interno di viste differenti[6].

I delegati sono elementi di estrema importanza all'interno del framework. Questi fungono da livello di indirezione tra la vista e il modello ed offrono supporto alla renderizzazione dei dati. È infatti possibile re-implementare la classe astratta che descrive tutti i delegati (*QAbstractItemDelegate*) per poter ad esempio inserire diversi widget di input nei campi mostrati in una tabella, come mostrato in figura 2.6.

[6]Si pensi a dati anagrafici che devono essere mostrati in più tipologie di grafici: grafici a torta, a barre, istogrammi, ecc...

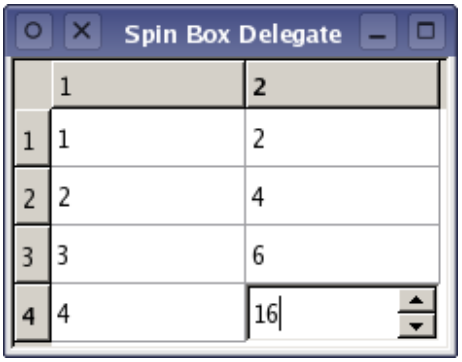


Figura 2.6: Un esempio di "delegate" in *Qt*4.

Utilizzo di MVC nel nuovo property system

Quanto appena descritto ha giocato un ruolo fondamentale nella comprensione del metodo più adatto ad effettuare le modifiche alle proprietà, sfruttando al meglio gli strumenti offerti da *Qt*. Come primo test si è partiti con un prototipo di property editor sul quale effettuare i vari esperimenti, cercando di mantenere il codice pulito per il riuso. Questa fase si affronta solitamente per fronteggiare problemi e risolverli al meglio. Il modello di MVC che *Qt* propone è stato sfruttato in tutti i suoi dettagli tranne che per i delegati in quanto, per gli scopi prefissati, non erano previste tipizzazioni nei campi valori delle proprietà.

Si è dunque partiti col creare una struttura ad albero n-ario, passando poi alla costruzione del modello, per finire quindi con la vista e l'interazione tra quest'ultima ed il modello stesso, ponendo particolare attenzione all'accoppiamento e alla coesione degli oggetti.

Durante la modellazione è stato deciso di accentrare il controllo su di un'entità astratta che fosse stata in stretto contatto con i dati e che raccogliesse le richieste di cambiamento. Uno dei primi problemi riscontrati

riguarda il fatto che il property editor sarebbe stato utilizzato non solo dagli elementi base di *VisualNetkit*, ma anche ad altri componenti quali Aree grafiche e proprietà stesse del laboratorio, prive di un interfaccia verso i *Plug-In*. Di qui la necessità di generalizzare il concetto di "property handler" tale da poter risultare abbastanza generale da essere esteso dai vari controllers concreti.

2.3.1 *Plug-In* per la Configurazione Avanzata

In questa sezione ci occuperemo di analizzare quanto sopra descritto. Si osservi quindi la figura 2.7.

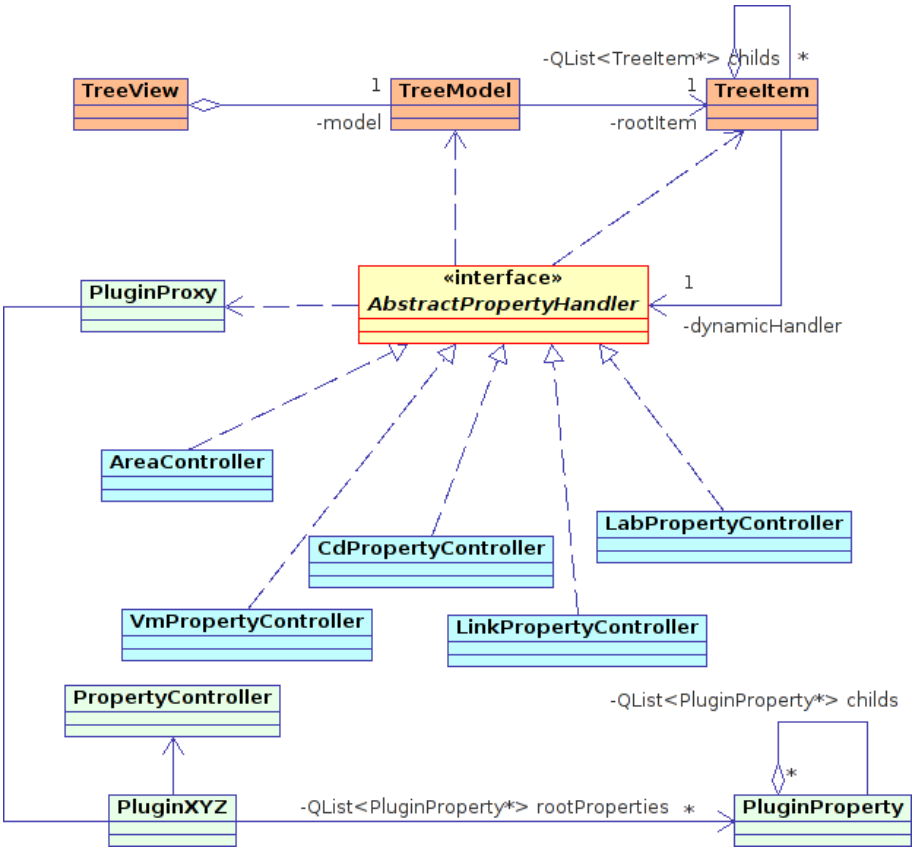


Figura 2.7: Diagramma delle classi del nuovo sistema.

Si nota subito la presenza di una generalizzazione per quanto riguarda il property handler, come precedentemente accennato. Tale interfaccia (in *C++* è chiamata una "pure abstract class") offre le seguenti funzioni:

getComposedModel() questa funzione indica che ogni classe che la estende deve necessariamente fornire un model appropriamente costruito, il quale verrà poi collegato alla vista. Il model viene creato ricavando le informazioni dall'elemento base e dalle properties offerte dai vari *Plug-In* che aggrega. Ogni "tree item" possiede quindi il riferimento al suo creatore, che è anche l'oggetto che riceve le richieste di cambiamento;

getInitMolel() questa funzione viene usata dalle sotto-classi che adempiono il ruolo di property controllers per gli elementi base[7], fornendo un modello simile al precedente ma che contiene solamente i dati dei *Plug-In* selezionati;

saveChangedProperty() viene utilizzata per il salvataggio dei cambiamenti effettuati in una determinata proprerty;

getPluginFromCurrentElement() questa funzione è puramente di convenienza e viene usata di rado;

removePluginProperty() viene utilizzata per inoltrare la richiesta di eliminazione di una property di un *Plug-In*. Il controllo di consistenza viene effettuato dal modulo stesso;

[7]Ricordiamo che per "elemento base" in *VisualNetkit* si intendono Virtual Machines, Collision Domains e Links.

addPluginProperty() come sopra, ma utilizzata per inserire una property (o sub-property) in un *Plug-In*.

Si arriva quindi ad avere un forte incremento di flessibilità ed estensibilità grazie all'aggiunta di properties dei moduli non più statiche bensì dinamiche, con struttura anche annidata (figura 2.8).

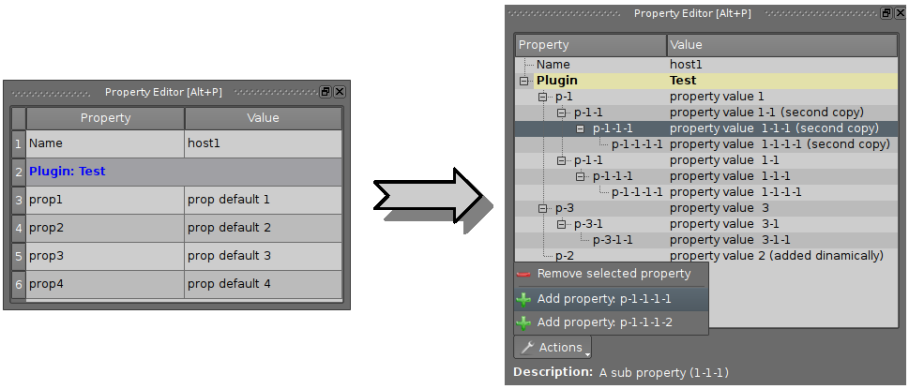


Figura 2.8: Paragone tra il vecchio ed il nuovo property editor.

2.3.2 Metodologie di sviluppo adottate

Al fine di rendere la progettazione più corretta e la fase di implementazione meno rischiosa, si è cercato di cogliere gli aspetti qualificanti di metodologie di sviluppo quali *XP*, *FDD* e le più note metodologie di sviluppo iterativo come *UP* (*Unified Process*). Seguendo le linee guida dettate da *UP*, si è optato per la definizione di cicli di sviluppo scanditi dal rilascio di prototipi funzionanti del progetto e dalla loro presentazione e discussione, così da avere una valutazione dell'avanzamento dello stesso ed una buona gestione dei fattori di rischio. Ogni ciclo è stato suddiviso in tre fasi: *inception phase*, *elaboration phase* e *construction phase*. Scopo principale della fase

iniziale è stato quello di delineare, nel modo più accurato possibile, il caso di interesse e di identificare gli elementi importanti affinché esso conduca al completamento delle funzionalità desiderate.

Nel primo ciclo di analisi e progettazione, questo stadio è stato supportato da strumenti quali un modello dei casi d'uso sufficientemente dettagliato da definire i principali obiettivi del progetto, una pianificazione dello stesso e delle prime fasi evolutive, una valutazione dei rischi ed una definizione primaria dei requisiti. Lo step che coinvolge l'elaborazione ha contribuito a definire la struttura complessiva del sistema. Questo ha compreso l'analisi di dominio ed un primo approccio alla progettazione dell'architettura a cui si è affiancato lo studio per la rappresentazione del modello logico.

Al termine di queste due è seguita una fase conclusiva di implementazione e rilascio di un prototipo e sotto-versioni dell'architettura, capaci di mostrare il completamento dei casi d'uso principali e delle funzionalità introdotte nei punti precedenti.

A dispetto delle linee guida proposte dalla metodologia *UP*, le quali prevedono una quarta *fase di transizione* in cui vengono condotte le attività di training e il *beta testing*[8] del sistema a scopo di verifica e validazione, ogni fase è stata arricchita da piccole iterazioni di analisi e test limitando considerevolmente l'overhead complessivo.

[8]Il beta testing (o beta-verifica) è un periodo di prova e collaudo di un software non ancora pubblicato, con lo scopo di trovare eventuali errori (bug). Questa operazione può essere svolta da professionisti pagati oppure, molto spesso, da semplici amatori (chiamati beta-testers).

La metodologia *XP* è stata abbracciata negli steps di verifica continua del progetto; la frequente reingegnerizzazione del software e la libertà di seguire o meno le fasi di sviluppo prefissate, ci ha permesso di raggiungere rapidi risultati. Nei punti finali dello sviluppo, per il raffinamento e perfezionamento delle funzionalità accessorie, si è optato per un tipo di sviluppo orientato verso *FDD* (*Feature Driven Development*). Quest'ultima, anch'essa parte dell'*Agile Manifesto*, cerca di spingere *XP* verso un'estrema flessibilità eliminando interamente la fase classica di progettazione e volgendo il pieno interesse verso le fasi di "definizione di una lista di funzionalità" e di uno "sviluppo per funzionalità".

Capitolo 3

Progettazione e Realizzazione di un Ambiente per la Configurazione Avanzata

Questo capitolo è dedicato alla comprensione e descrizione delle fasi di progettazione e realizzazione di *VisualNetkit*. Verrà mostrata in maniera dettagliata la nuova struttura dei *Plug-In* accennata nel precedente capitolo, e successivamente, verranno descritte le altre porzioni del sistema, analizzando i singoli moduli che lo compongono.

In questa introduzione si vuole offrire un quadro generale (figura 3.1) della composizione architetturale nello stato attuale di *VisualNetkit*, in modo da rendere più facile la localizzazione degli elementi durante la loro trattazione.

3.1 Architettura modulare basata su *Plug-In*

Si procederà dapprima all'analisi della struttura modulare del sistema che rappresenta anche il suo punto di forza per quanto riguarda la flessibilità e l'estensibilità.

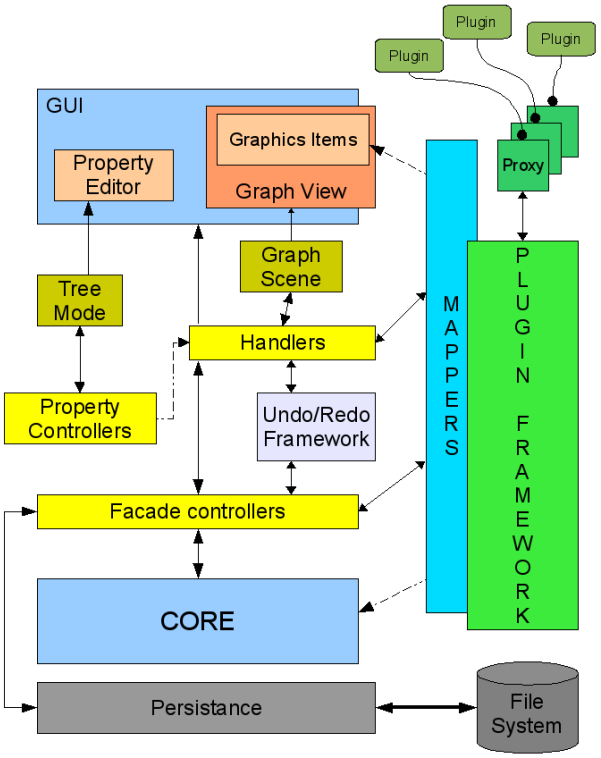


Figura 3.1: Diagramma dei componenti di *VisualNetkit*

Può essere auspicabile che un software usato in ambito scientifico sia in possesso di funzionalità opzionali, vale a dire funzionalità che possono essere aggiunte o rimosse nel corso di una simulazione senza ostacoli di rilievo. Si pensi ad esempio a quanto possa essere utile ad un progettista osservare il comportamento di una rete con o senza un determinato servizio (ad esempio *IPv6*) sulle macchine di cui è composta, o a quanto possa risultare comprensibile, per uno studente analizzare una topologia di rete malleabile che si presta a tutti i possibili test che intende effettuare.

L'utilizzo di un'architettura basata su *Plug-In* dona grande flessibilità all'intero sistema ed in particolare offre ampia libertà di utilizzo agli utenti finali.

Perché un sistema basato su *Plug-In*?

Durante le prime iterazioni e le prime fasi di testing sono emersi alcuni fattori di alto rischio. L'architettura era stata inizialmente concepita e realizzata sul concetto secondo cui tutti i requisiti dovevano essere assemblati staticamente nell'applicazione (come accade nei vari ambienti di configurazioni descritti nella sezione 1.2.1). Questa scelta prevedeva una lunghissima fase di sviluppo ed una continua ricerca dei requisiti e dei casi d'uso, che avrebbero potuto destabilizzare l'intero sistema. Inoltre, adottando questa tecnica non si sarebbe mai riusciti ad offrire una totale elasticità, ma piuttosto si sarebbe dovuti scendere a compromessi su cosa implementare e cosa no.

Analizzando nel dettaglio gli obiettivi, si è giunti alla conclusione che l'unica strada percorribile fosse quella che prevedeva la trasformazione dell'architettura monolitica in una modularizzata. Il core del sistema - senza alcun *Plug-In* attivo - avrebbe offerto all'utente solamente la possibilità di creare una rete a livello topologico, ossia priva di caratteristiche proprie.

Questa nuova tecnica avrebbe da un lato offerto una sicura controllabilità dei fattori di rischio, restringendoli alla re-ingegnerizzazione del sistema per renderlo modulare, e dall'altro avrebbe reso *VisualNetkit* estremamente scalabile e gli avrebbe conferito una connotazione del tutto unica nel suo genere.

3.1.1 Interazione tra sistema e *Plug-In*

Quando si sviluppa un'applicazione che si basa fortemente su *Plug-In*, la cosa fondamentale è definire i confini dell'uno e dell'altro sistema. Sostanzialmente, nucleo e moduli devono essere ben descritti per non rischiare di imbattersi in fenomeni di sovrapposizione: questi due mondi devono cooperare ma non intralciarsi, né tantomeno svolgere le stesse mansioni.

Dopo un'attenta ed accurata fase di studio, si è arrivati a definire la linea di demarcazione tra il sistema e le estensioni. Queste ultime possono essere attivate sugli elementi di base della rete, offrendo una sorta di caratterizzazione più specifica. Su di un componente possono essere attivi contemporaneamente più *Plug-In*, che di fatto donano una definizione ben precisa all'oggetto che li incapsula. Se per esempio su un host si attivano *Plug-In* quali DNS e HTTP, si può facilmente notare che il tipo di quell'oggetto non è altro che una semplice macchina virtuale che offre un servizio di DNS e che allo stesso tempo è un server Web.

In figura 3.2 è discritta in dettaglio la struttura del sotto sistema che permette ai moduli di colloquiare con il core dell'applicazione, e la definizione delle varie interfacce. Come si può osservare dal diagramma delle classi, tutto ruota intorno all'entità singleton "PluginRegistry". Questo controller viene invocato dal sistema centrale nel caso d'uso d'avviamento per scandire il *File System* e caricare in memoria i *Plug-In* esistenti, validarli e creare per ognuno di questi il proprio *loader factory*. Il PluginRegistry offre un servizio di directory, ossia è quel componente che mantiene traccia delle associazioni tra un *Plug-In* e un elemento base.

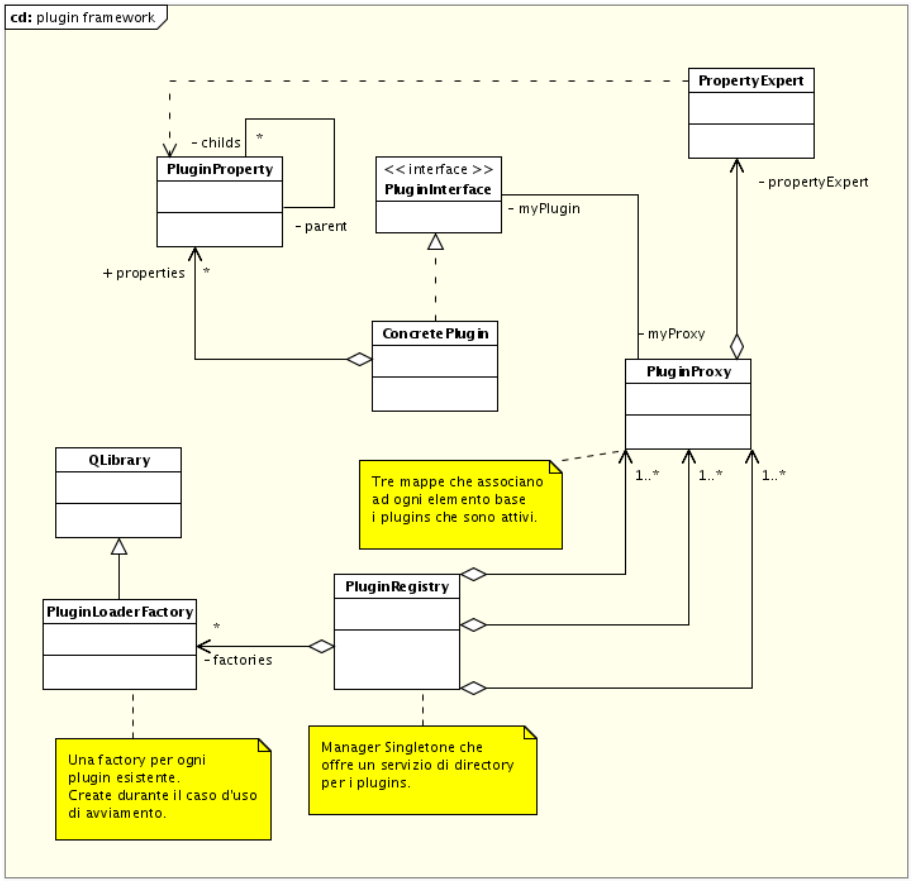


Figura 3.2: Diagramma delle classi del sotto sistema di gestione dei *Plug-In*.

In realtà il sistema non conosce direttamente l'instanza di un determinato modulo, ma conosce il suo *Proxy*, che introduce un livello di indirezione tra il core dell'applicazione e il *Plug-In* stesso. La scelta di applicare il pattern *Proxy*[13] assicura un buon controllo d'accesso all'oggetto reale di cui è responsabile, e permette inoltre un colloquio trasparente verso il sistema. Si osservi ora la classe astratta pura PluginInterface:

```
1 /**
2  * PluginInterface.h
3  */
4 class PluginInterface
5 {
```

```
6  public:
7      virtual ~PluginInterface() {};
8
9      virtual bool saveProperty(QString propUniqueId, QString propValue,
10             QString *pluginAlertMsg = NULL) = 0;
11
12     virtual QString getXMLResource() = 0;
13
14     virtual QMap<QString, QString> getTemplates() = 0;
15
16     virtual QList<PluginProperty*> getPluginProperties() = 0;
17
18     virtual PluginProxy* getProxy() = 0;
19
20     virtual void setProxy(PluginProxy* p) = 0;
21
22     virtual void setGroupID(qint32 id) { Q_UNUSED(id) };
23     virtual qint32 getGroupID() { return -1; };
24
25     virtual QString getDefaultGraphisLabel() = 0;
26
27     virtual QString getName() = 0;
28
29     virtual bool init(QString laboratoryPath) = 0;
30
31     virtual QString deleteProperty(QString propertyUniqueId) = 0;
32
33     virtual QPair<PluginProperty*, QString> addProperty(
34         QString propertyIdToAdd, QString parentPropertyUniqueId) = 0;
35
36 };
37
38 typedef PluginInterface* createPlugin_t();
39 typedef void destroyPlugin_t(PluginInterface*);
```

Trimite questa interfaccia un plugin può interagire con il sistema centrale. Ogni implementazione di un *Plug-In* deve quindi riscrivere le funzioni descritte. Il modulo deve inoltre definire le factory di creazione e distruzione - righe 38 e 39. Queste due funzioni sono usate dalla classe LoaderFactory (che risolve i due simboli **createPlugin** e **destroyPlugin**) che permettono di creare nuove instanze di un determinato modulo.

Per avere informazioni aggiuntive su come creare un *Plug-In*, si legga

l'appendice a pagina 89.

Ogni modulo è descritto da un proprio file di configurazione scritto in *XML*, che deve necessariamente essere inserito all'interno del *Qt Resource System*[1]. All'interno di questo file vi sono strutturate le credenziali del *Plug-In* come: il nome, la descrizione, gli autori, le dipendenze dagli altri *Plug-In* ecc..., ma anche un'accurata rappresentazione delle proprietà offerte all'elemento su cui verrà attivato il modulo.

Per facilitare questo processo, ogni *Proxy* è equipaggiato di una classe "expert" che si occupa della validazione e interrogazione del file *XML* sopra citato, nonché della manipolazione delle properties memorizzate all'interno di una struttura ad alberi n-ari.

Il sotto sistema appena descritto offre la possibilità di estendere le funzionalità di *VisualNetkit*. I vari *Plug-In* sono elementi passivi che vengono invocati dal sistema nei casi in cui: l'utente agisce su una property, l'utente inizializza un *Plug-In*, ecc... Tuttavia un modulo può anche interagire con il sistema stesso, ma solamente dopo che è stato invocato; questo avviene durante la fase di inizializzazione, quando il sistema invoca le funzioni di "getXmlResource()" e "getDefaultGraphisLabel()" per validare e scrivere una label nella vista del grafo della rete.

[1] Il sistema di risorse offerto da *Qt* consiste in un metodo per la memorizzazione di file binari all'interno dell'applicazione stessa (platform-indipendent). Questa è una tecnica molto utile quando l'applicazione ha sempre bisogno di determinati files e non si vuole correre il rischio di non possederli a runtime.

3.1.2 La gestione avanzata delle properties dei *Plug-In*

La creazione di un *Plug-In* in *VisualNetkit* è un'operazione che richiede uno stadio pre-implementativo che consiste nello stilare il file di configurazione in *XML*. Questo, oltre a descrivere i meta-dati - come ad esempio il nome del modulo -, assolve il compito di descrittore strutturale delle properties che il *Plug-In* offre. Si osservi l'esempio riportato qui sotto:

```
1  <plugin name="Test">
2
3   <global type="vm" version="1.0" author="Alessio Di Fazio" dependencies="">
4    <![CDATA[This is a test plugin.]]>
5   </global>
6
7   <properties>
8
9    <property id="uniqueID-0" name="1" default_value="" min="1" max="1000">
10    <![CDATA[Property 1]]>
11    <childs>
12     <property id="uniqueID-1" name="1-1" default_value="" min="1" max="2">
13      <![CDATA[A sub property (1-1)]]>
14      <childs>
15       <property id="uniqueID-2" name="1-1-1"
16                 default_value="" min="1" max="5">
17        <![CDATA[A sub property (1-1-1)]]>
18        <childs>
19         <property id="uniqueID-3" name="1-1-1-1"
20                   default_value="" min="0" max="3">
21          <![CDATA[A sub property (1-1-1-1)]]>
22          <childs />
23        </childs>
24       </property>
25      </childs>
26     </property>
27    </childs>
28   </property>
29
30   <property id="uniqueID-4" name="p-2" default_value="" min="0" max="3">
31    <![CDATA[Property 2]]>
32    <childs />
33   </property>
34
35  </properties>
36
```

```
37 </plugin>
```

è possibile osservare una prima parte che descrive alcune caratteristiche del modulo quali il tipo di elemento a cui questo può essere applicato, il numero di versione, l'autore, il nome del *Plug-In* e le dipendenze.

Il tag "properties" è il più importante e contiene tutte le possibili proprietà offerte, nonché una descrizione a livello gerarchico di queste ultime. La rappresentazione di una property deve contenere un ID univoco, un nome, un valore di default qualora necessario, un valore di minimo e massimo che rappresentano la cardinalità e una descrizione libera[2].

I valori "min" e "max" rappresentano gli attributi principali. Il primo può assumere valori pari a 0 o 1, mentre il massimo può contenere qualsiasi valore compreso tra 1 e 65536 (un intero senza segno a 16 bit). Impostando un valore minimo pari a 1, la property in questione viene creata insieme all'instanza del *Plug-In*, ed insieme alla property "padre" se quest'ultima viene in futuro aggiunta dall'utente.

Il controllo durante le azioni di inserimento e la cancellazione delle properties è a carico del modulo, il quale è tenuto a verificare il valore della cardinalità attuale di una property prima di eliminarla. Come è facilmente intuibile, un *Plug-In* non può autorizzare l'eliminazione di una proprietà - priva di altre copie - quando la cardinalità minima di quest'ultima è pari a 1. Un discorso analogo può essere applicato all'attributo "max".

[2]La descrizione di una property può contenere anche codice HTML per la formattazione del testo.

3.2 Elementi architetturali

Appurato il funzionamento e la logica che risiede dietro il framework dei *Plug-In*, è possibile iniziare ad introdurre gli altri elementi architetturali che di fatto supportano e interagiscono l'un l'altro per soddisfare tutti i requisiti (funzionali e non) che il sistema offre.

Si inizierà ad analizzare gli elementi in figura 3.1 con strategia **top-down** per quanto riguarda gli elementi che hanno una collocazione orizzontale, successivamente si analizzeranno i mappers ed in fine gli sforzi verranno convogliati nella descrizione del nuovo sistema che offre ai *Plug-In* la possibilità di possedere un'alta dinamicità nelle proprietà che questi offrono.

3.2.1 L'interfaccia utente

Uno degli obiettivi principali di questo progetto è la realizzazione di un ambiente di configurazione di reti virtuali, che offra un'interfaccia grafica capace di fornire una vista intuitiva della rete virtuale. Tale interfaccia, oltre a garantire un alto livello di ergonomia e usabilità, deve garantire buone capacità di adattamento al variare dei requisiti.

Sebbene la realizzazione di una interfaccia grafica possa dare l'impressione di essere un'attività semplice, durante tale periodo si incontrano molteplici difficoltà in quanto l'ambiente in esame si avvicina molto alla definizione di "IDE"[3] orientato - nel nostro caso - allo sviluppo di reti virtuali.

[3]Un *integrated development environment* (IDE), in italiano ambiente integrato di sviluppo, è un software che aiuta i programmatori nello sviluppo del software.

La realizzazione dell'attuale interfaccia ha richiesto diversi cicli di sviluppo e raffinamento per ottenere un risultato apprezzabile ed intuitivo. Questa è composta da varie *docks* - figura 3.3 - che al loro interno contengono vari sotto elementi come properties, zoom e miniatura del grafo, elementi attivi, struttura de Lab e un elemento centrale che rappresenta la scena che consente di disegnare la topologia di rete desiderata. L'ampio uso di grafica SVG fa si che all'utente arrivi subito un feedback positivo che incentiva lo stesso a sperimentare le caratteristiche offerte dall'ambiente in cui è inserito.

Il grafo non è altro che la vista della struttura del modello di dominio. Ogni elemento grafico (Hosts, Collision Domains, Links) è in relazione con un elemento di basso livello che incapsula le informazioni. Gli elementi grafici sono sostanzialmente degli oggetti che riflettono queste informazioni all'utente e gli consentono di interagire con gli elementi stessi.

L'esistenza del *Graphics View Framework* offerto da *Qt*, ha reso minimi gli sforzi implementativi in quanto questo è fortemente basato sul pattern MVC, che consente di collegare uno stesso modello (la scena) a più viste come avviene per la scena e la *dock* della miniatura - figura 3.3.

Dettagli implementativi

L'interazione con l'utente è stata implementata seguendo i principi dell'*event driven programming*, che rappresenta la base della programmazione delle interfacce grafiche. La programmazione guidata dagli eventi non considera i meccanismi standard di input (ad esempio la CLI - *Command Line Interface*) ma basa il suo funzionamento sulla gestione degli eventi generati

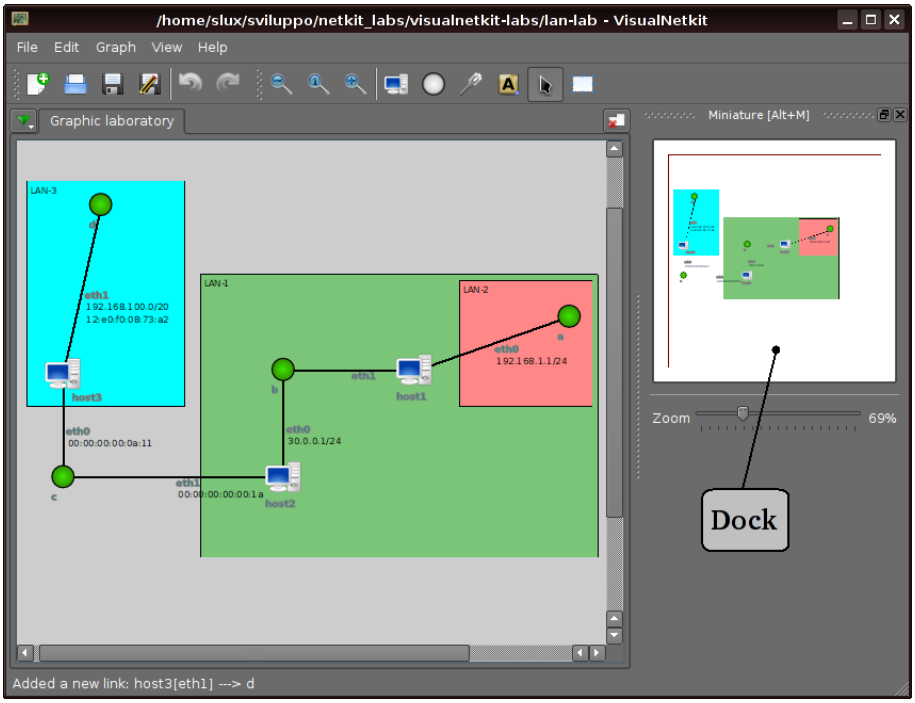


Figura 3.3: Scena del grafo inserita in due viste differenti e difinizione di "Dock".

dall'utente, coma la pressione di un bottone o un click del mouse.

L'insieme degli eventi di interesse vengono ricevuti da handlers che effettuano una prima validazione e inoltrano lo stesso evento agli strati di competenza, posti solitamente nei livelli più bassi dell'architettura.

Signals e Slots in *Qt*

In *Qt* slot e segnali sono usati per la comunicazione asincrona tra oggetti. Questo sistema fa in modo che oggetti di un certo tipo vengano avvertiti e siano in grado di comunicare con altri elementi, quando su di un oggetto si scatena un determinato evento. Per esempio, se un utente clicca un bottone **Close**, probabilmente si vorrebbe che il widget si chiuda chiamando la sua funzione "close()".

Un segnale viene emesso quando un particolare evento accade, uno slot è invece una funzione che se invocata, è responsabile di gestire un particolare segnale. Può però anche essere invocata in maniera sincrona da un qualche altro oggetto. Tale meccanismo nasconde in realtà un sistema già noto con il nome di pattern *Observer*. L'unica differenza risiede nelle chiamate asincrone che i signal effettuano allo slot di loro competenza, senza aspettare una risposta. Infatti, ogni slot è definito con un tipo di ritorno "void", e l'unico metodo che permette di effettuare modifiche sui dati è tramite effetti collaterali sull'elemento passato come puntatore. In figura 3.4 è mostrato un esempio d'uso di Signals e Slots.

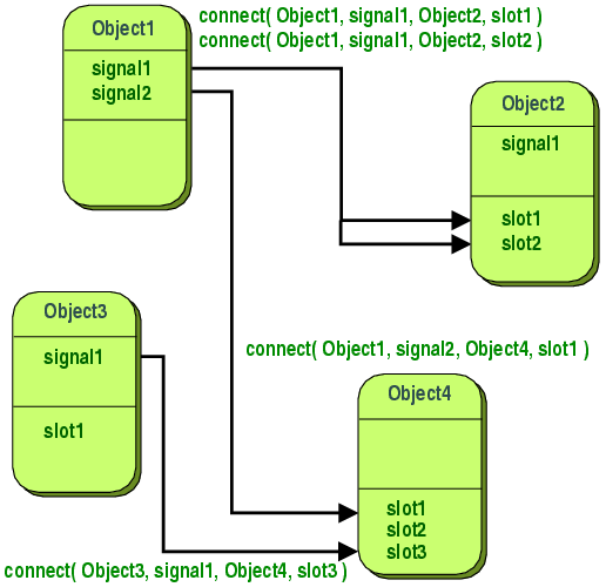


Figura 3.4: Schema d'uso di Signals e Slots.

L'editor testuale

VisualNetkit non offre solamente la possibilità di disegnare la rete di calcolatori e caratterizzarla tramite l'applicazione dei moduli. Durante la realiz-

zazione di un Lab è possibile che l'utente voglia autonomamente applicare modifiche ai files di configurazione che l'ambiente produce.

Per questo motivo *VisualNetkit* include al suo interno un editor di testo potente che offre un elevato numero di regole per il *Syntax Highlighting*, come mostrato in figura 3.5. Ogni singolo editor offre la possibilità di annullare le modifiche (Undo e Redo) e salvare il testo, dando la possibilità di creare una copia di backup.

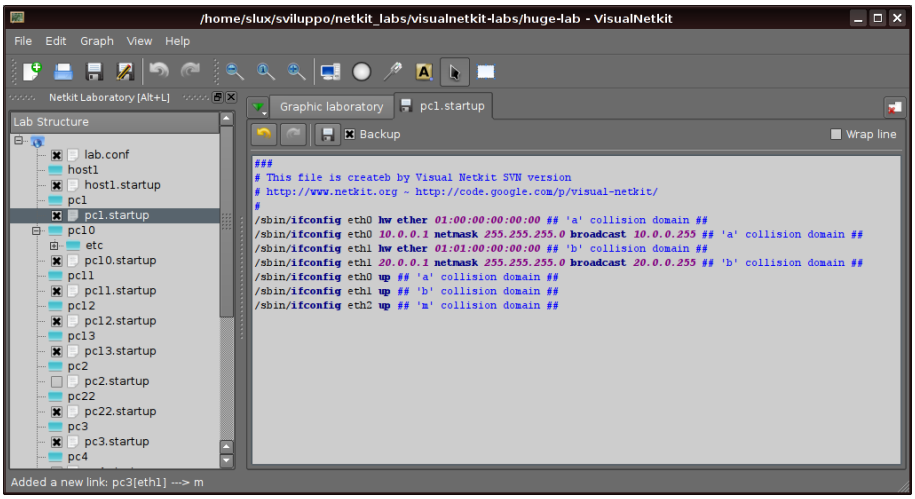


Figura 3.5: L'editor testuale di *VisualNetkit*

3.2.2 Elementi del dominio

Il kernel di questo progetto è il primo componente che è stato sviluppato ed anche il più importante. In esso sono contenuti tutti gli oggetti elementari su cui si basa la topologia di rete che l'utente vuol creare. Le classi rappresentano elementi della realtà di interesse come le virtual machine, i collision domain, le hardware interface e anche l'intero laboratorio.

Come si nota dal diagramma delle classi presente in figura 3.6, que-

sto package ha una struttura molto semplice, ma allo stesso tempo riesce a descrivere qualsiasi topologia di rete virtuale. Infatti, un laboratorio è composto da un insieme di oggetti VirtualMachine e CollisionDomain contenuti all'interno di due Mappe ordinate per nome.

La giunzione tra questi ultimi due elementi è occupata dalla classe HardwareInterface che funge da collante di riferimento tra l'host virtuale, ed i vari domini di collisione a cui è collegato. Grazie a questa semplice struttura *VisualNetkit* nasce da una parte come ambiente per la creazione di reti virtuali da emulare con *NetKit*, ma è dall'altra potenzialmente in grado di essere riadattato in modo da poter produrre configurazioni per altri sistemi di emulazione.

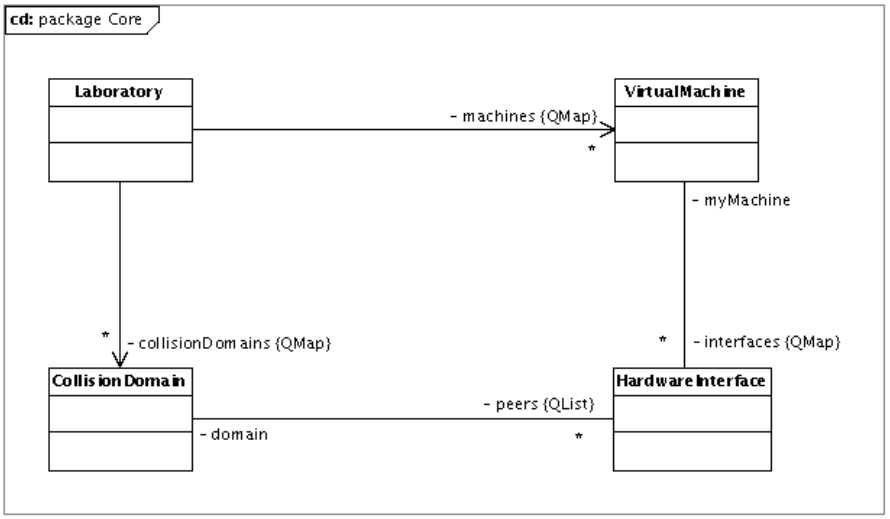


Figura 3.6: Diagramma delle classi del package Core.

Come vedremo più avanti, gli oggetti di tipo HardwareInterface saranno rappresentati da dei Links visualizzati come linee *SVG* all'interno della scena grafica.

3.2.3 La gestione degli eventi

Come in tutte le applicazioni software, ogni oggetto ha bisogno di essere "gestito" nel contesto dell'applicazione stessa. La funzione di amministrazione e controllo del singolo oggetto - in applicazioni realizzate a regola d'arte -, viene demandata ad altri oggetti tipicamente più leggeri ma non per questo meno complessi, detti *handler*.

Le funzioni di controller sono state realizzate seguendo le linee guida del pattern *Controller*[12] ed assegnando le responsabilità del controllo e compiti dell'oggetto interessato. In *VisualNetkit* esistono vari tipi di *handlers*:

- *Facade Controllers*;
- *Use Case Controllers*;
- *Mappers* - macro elementi che disaccoppiano vista e dominio;
- *Property Controllers*.

I *Facade Controllers* vengono utilizzati per raccogliere le richieste che provengono dai livelli più alti - in particolare dagli *Use Case Controllers* - quando c'è bisogno di un accecesso a basso livello che comporta un qualche tipo di cambiamento delle strutture del Dominio.

Gli *Use Case Controller*[4] sono posti in strati medio-alti dell'architettura e sono i primi oggetti che ricevono i segnali provenienti dall'interfaccia grafica. Questi effettuano una prima validazione dell'atto svolto e, se non

[4]Nella figura 3.1 sono indicati come *Handlers* generici.

vi sono errori, inoltrano la richiesta agli oggetti più bassi, in particolare ai *Facade Controllers* e, in qualche caso anche agli oggetti *Mappers*.

3.2.4 Interazione tra Utente e Proprietà degli Elementi

Ogni elemento del dominio incapsula alcune proprietà di base. Queste, oltre a caratterizzare i vari oggetti, servono anche per descrivere una certa realtà di interesse espressa dall'utente. Ciò significa che è proprio il cliente finale che vuole avere la facoltà di cambiare questi attributi in ogni momento.

In *VisualNetkit*, oltre alle proprietà offerte dagli oggetti del dominio (ad esempio il nome di una *Virtual Machine*), per la natura modulare del sistema, è possibile che per un oggetto base vi siano collegati alcuni *Plug-In* che offrono proprietà aggiuntive. La gestione di queste è demandata ad oggetti - di tipo *Use Case Controller* - che sono responsabili di renderizzare i dati e ricevere le modifiche che l'utente desidera (figura 2.7).

È possibile osservare come questi oggetti vengono dinamicamente inseriti come handlers degli elementi che rappresentano i dati delle properties, poiché la natura delle entità che l'utente può selezionare è eterogenea: ogni tipologia di elemento possiede il suo *Property Controller*.

3.2.5 Gestione degli annullamenti: l'Undo Framework

Per l'implementazione delle funzionalità di undo e di redo delle azioni svolte è stato utilizzato un potente framework offerto da *Qt*. Questo sistema ha permesso, in fase realizzativa, di tralasciare molti dei dettagli implementativi e dirigere l'attenzione verso problematiche come la consistenza dei

dati. Infatti, sebbene le operazioni di undo o redo siano atomiche, esse interagiscono con lo strato dei *Facade Controllers* e quindi indirettamente anche con gli oggetti del Dominio.

L'Undo Framework in *Qt*

Il *Qt Undo Framework* è un'implementazione del pattern *Command*[12] per l'attuazione di funzionalità di undo/redo nelle applicazioni. Il pattern *Command* è basato sul concetto che tutte le funzioni di editing in un'applicazione sono realizzate con la creazione di instanze di elementi command. Analizzando l'esempio di un editor di testo, gli oggetti command applicano le modifiche al documento e sono memorizzati in uno stack dei comandi.

Un *QUndoCommand* rappresenta un singolo atto di editing all'interno del documento come, ad esempio, l'inserimento o l'eliminazione di blocchi di testo. Un comando può apportare cambiamenti al documento con azioni di *redo()* o di *undo()*.

Inoltre, ogni comando è capace di annullare i cambiamenti da lui apportati e come riportare il documento allo stato precedente. Finché l'applicazione utilizza solo oggetti command per cambiare lo stato del documento, è possibile annullare una sequenza di comandi attraversando lo stack verso, disfacendo ciascun comando presente. È anche possibile ripetere una sequenza di comandi percorrendo la pila all'inverso.

La flessibilità offerta da questo strumento è tale da poter essere applicata ad ogni tipo di situazione. Viene ora analizzato un esempio un po' più realistico e complesso.

Nel framework ogni azione che l'utente vuole poter annullare è imple-

mentata in una classe che estende *QUndoCommand*. Per ogni azione effettuata viene creato un apposito command che andrà inserito nel *QUndoStack* di riferimento. Appena l'oggetto prende posto all'interno dello stack, il framework provvede ad invocare la sua funzione di redo per completare e/o effettuare l'azione richiesta.

L'esempio riportato implementa un semplice diagramma grafico, è pertanto possibile eliminare e aggiungere elementi e muoverli tramite *drag&drop*. Ognuna di queste azioni possiede il corrispondente undo-command, che permette eventualmente all'utente di tornare sui suoi passi. In figura 3.7 è mostrato come un *QUndoStack* può essere mostrato graficamente all'interno di un *QUndoView*, dove ogni riga presente rappresenta un singolo Command.

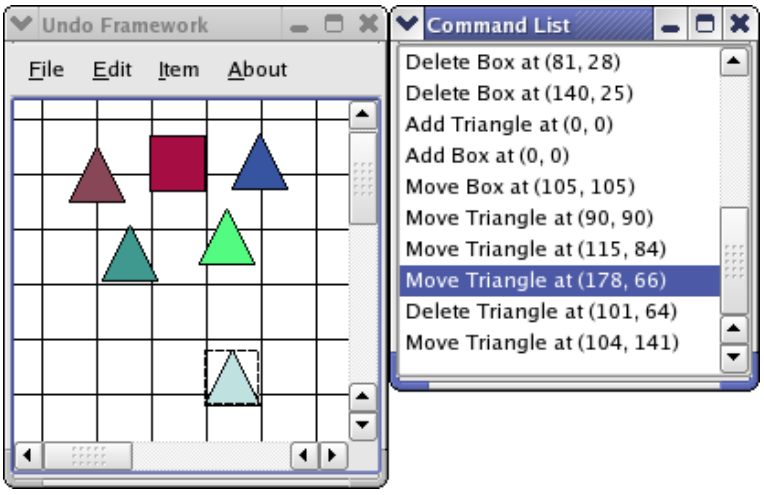


Figura 3.7: Un esempio avanzato dell'Undo Framework offerto da *Qt*.

3.2.6 L'accesso al *File System*

Lo strato di persistenza dei dati nel complesso di ogni applicazione software gioca un ruolo fondamentale. Generalmente posto nello strato più basso del sistema, questo elemento architetturale ha il duplice scopo di interagire con il *File System* per salvare i dati inerenti alle varie configurazioni dell'applicazione e dei laboratori creati, nonché esportare questi ultimi in un formato accettato e riconosciuto dall'ambiente di emulazione desiderato.

Uno dei problemi riscontrati in fase di progettazione è stato come strutturare le informazioni che dovevano essere salvate. Da una parte occorreva memorizzare lo stato del laboratorio creato - in particolare lo stato del grafo della rete -, mentre dall'altra era necessario trovare un buon sistema per poter esportare il laboratorio in modo da essere poi eseguito correttamente dal sistema di emulazione su cui quest'ambiente si basa[5].

Ad esempio, se un utente desidera salvare un Lab per poi riaprirlo, dev'essere possibile memorizzare meta-dati sugli elementi presenti nella scena grafica, che potrebbero essere la posizione degli oggetti stessi o delle etichette, ed eventuali informazioni inerenti ai *Plug-In*.

Si è deciso di procedere, quindi, con la registrazione delle meta-informazioni su di un file in formato *XML* nominato *lab.xml*, posto nella root directory del laboratorio. Il fatto di salvare i meta-dati in un file separato rende

[5]Ribadiamo il concetto che *VisualNetkit* nasce come ambiente per creazione di lab per *NetKit*, ma questo non vincola il tool ad un eventuale adattamento in favore di un qualsiasi altro sistema di emulazione.

il lab prodotto standard e auto descritto da un singolo file di configurazione esterno. Un tag principale "scene" rappresenta alcune informazioni sulla scena grafica, come ad esempio la dimensione. Internamente a questo vi sono tags di gruppo che descrivono i vari componenti grafici come links, aree, virtual machines e collision domains. Ad ognuno di questi sono associate informazioni extra che descrivono meta-dati aggiuntivi, quando previsto.

La porzione del sistema che esporta la struttura del laboratorio su *File System*, compie quattro fasi:

Fase 1 questo stadio iniziale è adibito alla creazione della struttura base delle directories. In particolare, per ogni *Virtual Machine* presente nel Lab creato viene prodotta una directory con il nome dell'host;

Fase 2 è il momento della creazione del file *lab.conf* che contiene la struttura fondamentale della rete a livello topologico, riconosciuta da *NetKit*;

Fase 3 questa fase è riservata al salvataggio dei file di "sturtup" delle macchine presenti[6];

Fase 4 l'ultimo step permette ad ogni plugin attivo di offrire porzioni (o la totalità) di file di configurazioni che descrivono i propri settaggi.

[6]In *NetKit* ogni host virtuale che viene avviato può operare azioni di post-start grazie all'esecuzione del file *.startup* che rappresenta una precisa macchina virtuale.

Template Engine in *VisualNetkit*

Nelle fasi due, tre e quattro si è parlato della creazione dei file di configurazione. Sia i *Plug-In* che l'applicazione stessa utilizzano un sistema di *mark-up* per la costruzione dei vari templates, i quali andranno poi scritti dallo strato di persistenza all'interno dei files. Questi templates possiedono una struttura interna che prevede la sostituzione localizzata di alcune parti. Si prenda in esame il templete utilizzato per creare il file "lab.conf":

```
1  ###
2  # This file  is  createb by Visual  Netkit  <VISUAL_NETKIT_VERSION> version
3  # http://www.netkit.org ~ http://code.google.com/p/visual-netkit/
4  #
5
6  LAB_DESCRIPTION="<DESCRIPTION>"
7  LAB_VERSION="<VERSION>"
8  LAB_AUTHOR="<AUTHOR>"
9  LAB_EMAIL="<EMAIL>"
10 LAB_WEB="<WEB>"
11
12 <TOPOLOGY><HOST>[<ETH_NUMBER>]="<COLLISION_DOMAIN_NAME>"</TOPOLOGY>
```

Il Template Engine interno all'applicazione viene invocato passando il riferimento (basato sul resource system di *Qt*) ad un template file. Questo viene dapprima caricato in memoria e successivamente, vengono effettuare alcune sostituzioni di base come quella del *mark-up* `<VISUAL_NETKIT_VERSION>`, che viene sostituito dalla corrente versione del tool.

L'elemento che invoca il caricamento di un template riceve quindi il contenuto del file desiderato. In seguito vengono effettuate le altre sostituzioni tramite l'ausilio di espressioni regolari talvolta complesse.

Si osservi la riga 12: l'oggetto incaricato di salvare il file "lab.conf" ha la mansione di completare la struttura del file di configurazione inserendo la

descrizione, la versione, l'autore, l'indirizzo email e il sito web del lab corrente, ed in secondo luogo andrà a costruire successivamente la topologia dell'intera rete, iterando tra i *tag* `<TOPOLOGY></TOPOLOGY>`.

In questo modo si riesce a creare un file di configurazione corretto e ben formattato come quello proposto qui sotto, che descrive la topologia di rete presente in figura 3.3.

```
1  ###
2  # This file is createb by Visual Netkit 1.1 version
3  # http://www.netkit.org ~ http://code.google.com/p/visual-netkit/
4  #
5
6  LAB_DESCRIPTION="A simple laboratory"
7  LAB_VERSION="1.0"
8  LAB_AUTHOR="Alessio Di Fazio"
9  LAB_EMAIL="slux83@gmail.com"
10 LAB_WEB="http://code.google.com/p/visual-netkit/"
11
12 host1[0]="a"
13 host1[1]="b"
14
15 host2[0]="b"
16 host2[1]="c"
17
18 host3[0]="c"
19 host3[1]="d"
```

È importante notare come il template dell'esempio appena citato risulti totalmente indipendente dal motore di emulazione utilizzato, ed è dunque facilmente rimodellizzabile al fine di essere impegnato con emulatori differenti.

Nel caso in cui un emulatore richiedesse una sintassi particolare per le varie macchine virtuali, come ad esempio VNUML, basterebbe creare un semplice *Plug-In* che offra uno specifico template ed aggiungere questo *Plug-In* ai vari host virtuali presenti nella topologia di rete.

3.2.7 Controllers tra Dominio e Vista

Un ulteriore elemento architetturale che spicca nella figura 3.1 è il componente *Mapper*. Questo package è composto da varie classi *singleton*, predisposte al ruolo di mappare gli oggetti della vista in quelli del dominio, e viceversa. Da ciò deriva la verticalità che assume tale componente all'interno del diagramma.

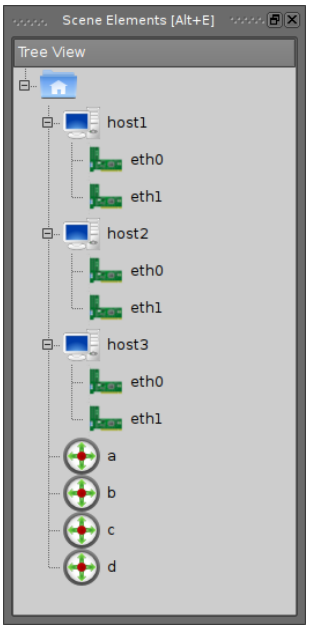


Figura 3.8: Albero degli elementi presenti sulla scena.

Ogni tipologia di oggetti del dominio possiede un proprio mapper di riferimento. Questo è ciò che accade per una *Virtual Machine* che deve essere associata al corrispettivo elemento grafico. Tale scenario è reso possibile grazie al mapper per le macchine virtuali (*VmMapper*).

Questi "controllers" inoltre, hanno il compito di disaccoppiare gli oggetti della vista con quelli del dominio, inserendo un ulteriore livello di indirezione. Tale soluzione è stata studiata per ovviare al noto problema

che il pattern MVC introduce, ossia il forte accoppiamento tra la vista e il modello.

In questo contenitore possiamo individuare diversi tipi di mappers. Oltre a quelli utilizzati come controllers per gli oggetti del dominio, vi sono elementi - *ad esempio SceneTreeMapper* - che si occupano principalmente di mantenere allineato lo stato degli elementi del dominio con una determinata porzione dell'interfaccia utente, in questo caso l'albero degli elementi grafici (figura 3.8).

3.3 Strumenti di supporto allo sviluppo

Durante tutto il lavoro sono stati utilizzati molteplici strumenti che hanno contribuito alla realizzazione del prodotto.

3.3.1 Il framework *Qt*

Qt è un *framework applicativo multi-piattaforma* (figura 3.9) per la realizzazione di software desktop ed *embedded*, sviluppato da *Nokia-Trolltech* e distribuito in versioni per uso commerciale e open source, quest'ultima sotto licenza *GPL-3*. *Qt* include una documentazione (**A**pplication **P**rogram **I**nterface) molto intuitiva ed un ricco insieme di classi *C++*, strumenti integrati per lo sviluppo e l'internazionalizzazione delle GUI, e il supporto per lo sviluppo in Java e *C++*.

Sono inoltre presenti svariati sistemi di binding per altri linguaggi di scripting quali QtPerl, QtPython, QtRuby, ecc...

La modularità delle librerie *Qt* fornisce un ricco insieme di estensioni

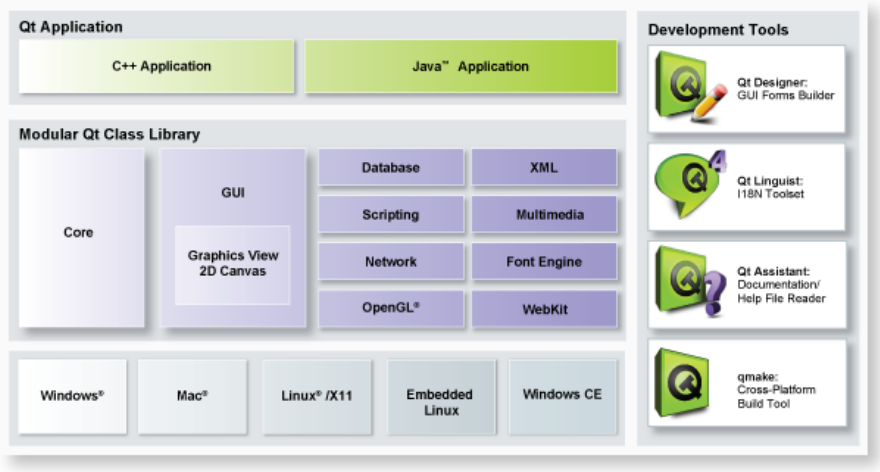


Figura 3.9: Panoramica degli elementi presenti nel framework *Qt*.

per la costruzione di applicazioni, integrando tutte le funzionalità necessarie per la creazione di software avanzati e multi-piattaforma.

Il framework equipaggia inoltre strumenti di sviluppo integrato per la realizzazione assistita di interfacce grafiche, traduzione e internazionalizzazione, documentazione e realizzazioni. Ossi i *Development Tools*.

Qt Designer

Qt Designer è un potente strumento per la costruzione di GUI e form con layout nativo e multi-piattaforma. Questo consente la rapida progettazione e realizzazione di "widgets" e finestre di dialogo, utilizzando le stesse tecniche di programmazione e mantenendo il tutto disaccoppiato dall'implementazione dal sistema. Le form create con *Qt Designer* godono di piena funzionalità e possono essere visualizzate in anteprima in modo da garantire che il loro impatto visivo rifletta esattamente le aspettative del programmatore.

Una volta creato il widget con l'ausilio del tool in questione, viene prodotto un file con estensione `.ui`, che al suo interno contiene una struttura basata su *XML*. All'interno dell'applicazione, la form dovrà essere implementata collegando questo file (**U**ser **I**nterface) all'oggetto che si vuole utilizzare nel sistema in questione.

Esistono vari metodi per collegare un file UI ad una classe della nostra applicazione. Quello utilizzato durante l'intero lavoro prende il nome di "Multiple Inheritance Approach" che consiste nell'inserire la form creata tramite *Qt Designer* all'interno del file di progetto del sistema. Il precompilatore (uic - User Interface Compiler) si preoccuperà di trasformare il file *XML* in un header *C++* da poter utilizzare nella classe concreta del widget. Si osservi l'esempio riportato di seguito:

```
1  /**
2   * ui_calculatorform.h e' l'header creato
3   * tramite il pre-compilatore 'uic'
4   */
5  #include "ui_calculatorform.h"
6
7  /**
8   * Implementazione del widget creato con QtDesigner
9   */
10 class CalculatorForm : public QWidget, private Ui::CalculatorForm
11 {
12     Q_OBJECT
13 public:
14     CalculatorForm(QWidget *parent = NULL);
15
16 private slots:
17     void on_inputSpinBox1_valueChanged(int value);
18     void on_inputSpinBox2_valueChanged(int value);
19 };
```

La classe concreta *CalculatorForm* deve estendere *Ui::CalculatorForm* privatamente per assicurare che l'oggetto sia un rappresentante del widget

creato con *Qt Designer*.

A differenza di altri strumenti per la creazione di interfacce grafiche, *Qt Designer* mette a disposizione dell'utente una quantità di strumenti (figura 3.10) che permettono di muovere e scalare gli elementi nell'interfaccia in modo automatico, offrendo quindi layouts auto-adattabili. Ne consegue che le interfacce sono sia funzionali che native, e si adattano comodamente all'ambiente operativo ed alle preferenze dell'utente.

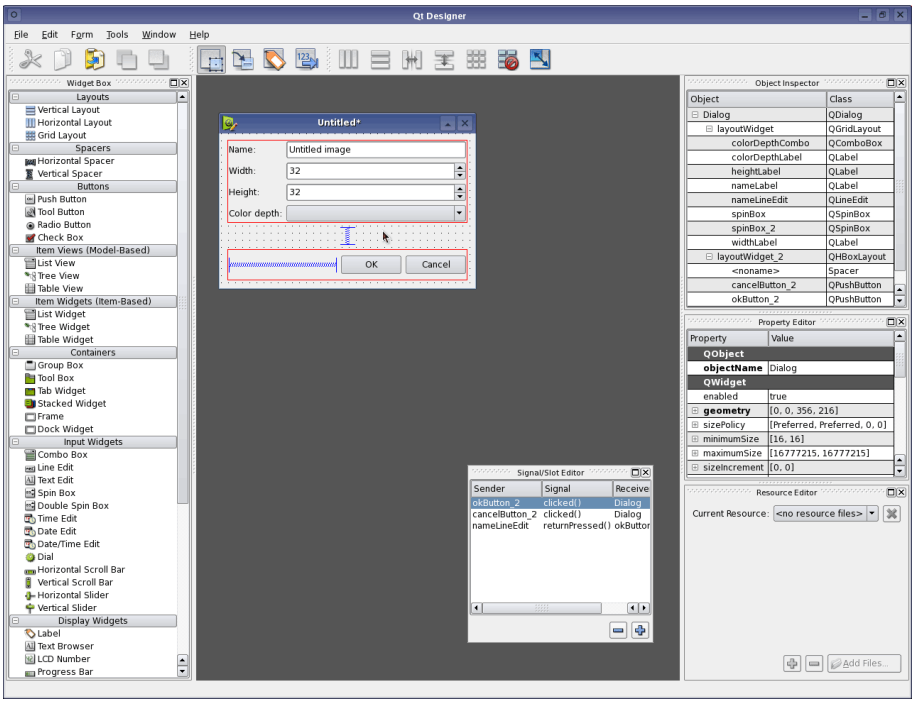


Figura 3.10: Un esempio di utilizzo di *Qt Designer*.

Qt Linguistic

Qt Linguistic fornisce una serie di strumenti che velocizzano la traduzione e l'internazionalizzazione delle applicazioni (*i18n*). *Qt* offre anche il suppor-

to simultaneo a più lingue e sistemi di scrittura, il tutto con un solo albero dei sorgenti ed un solo codice binario.

Il tool fornisce inoltre un'interfaccia che riduce i tempi del processo di traduzione della GUI, raccogliendo tutto il testo presente nelle UI che verrà mostrato all'utente finale in una finestra separata, per facilitarne la traduzione. Quando il testo è tradotto, il programma procede automaticamente alla prossima UI, fino a quando saranno completate tutte. Questo consente una più veloce e più accurata traduzione, offrendo alle applicazioni un supporto completo all'internalizzazione.

Qt Assistant

La maggiorparte delle applicazioni necessita di documentazione on-line o di un file di aiuto. *Qt Assistant* è un lettore di documentazioni configurabile (figura 3.11) che può essere facilmente personalizzato e ridistribuito con le singole applicazioni create, e che funziona in modo simile ad un browser web presentando la documentazione come una raccolta di pagine e collegamenti ipertestuali. Questo riconosce pagine con segnalibri e presenta i pulsanti standard "precedente" e "successivo" nella barra degli strumenti per la navigazione tra le pagine visitate. *Qt Assistant* utilizza il l'insieme della documentazione offerta dalle librerie *Qt* nei formati *rich text* e HTML per la visualizzazione delle pagine. Ciò significa che redattori e tecnici non hanno bisogno di software ad-hoc per creare la propria documentazione, ma possono utilizzare gli strumenti come *Doxygen* per scrivere i meta-dati direttamente nei sorgenti.

Il tool fornisce anche un veloce strumento di ricerca per accedere ra-

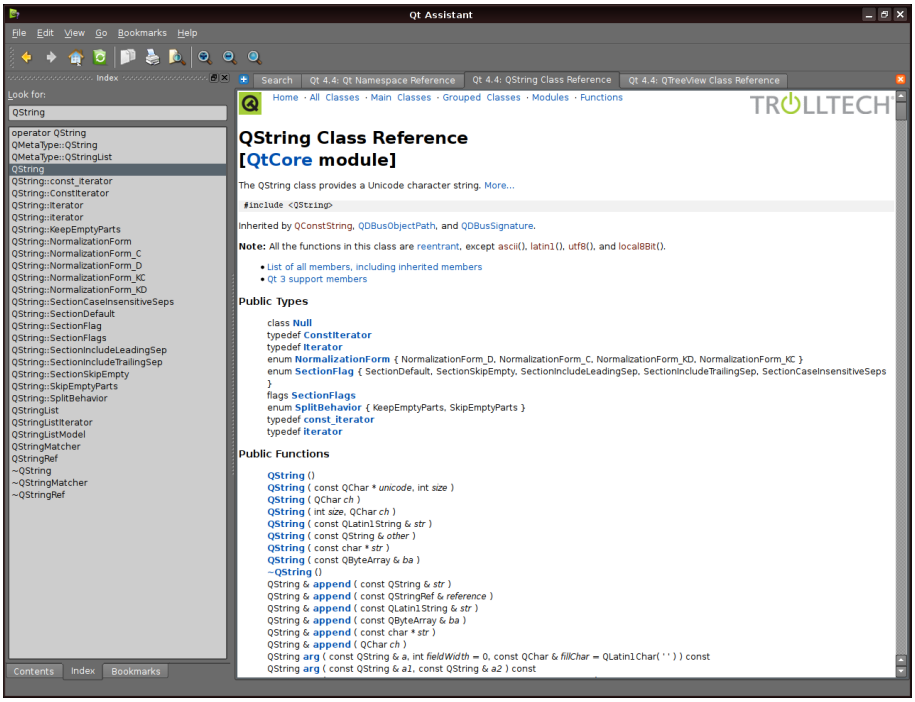


Figura 3.11: *Qt Assistant*.

pidamente all'argomento di interesse. I documenti rinvenuti nel corso di un'indagine vengono riportati in ordine di rilevanza, ed ogni occorrenza della parola chiave all'interno della documentazione viene evidenziata.

QMake

QMake è uno strumento multi-piattaforma che semplifica il processo di costruzione dei progetti *Qt* su diversi sistemi. A seconda del sistema operativo in cui ci si trovi, *QMake* genera i vari *Makefiles* in modo da essere compatibili con la piattaforma utilizzata.

QMake genera un Makefile sulla base delle informazioni presenti all'interno del file di progetto. Questi ultimi vengono creati dallo sviluppatore e sono molto semplici; un tipico esempio è riportato sotto:

```
1  ##############################################
2  #
3  # Example of .pro  file
4  #
5  ##############################################
6
7  TEMPLATE = app
8  TARGET   = test
9  LANGUAGE = C++
10 CONFIG  += console debug
11
12 QT += core \
13      gui
14
15 HEADERS  = stable.h \
16            mydialog.h \
17            myobject.h
18 SOURCES  = main.cpp \
19            mydialog.cpp \
20            myobject.cpp \
21            util.cpp
22 FORMS    = mydialog.ui
```

3.3.2 Altri strumenti secondari

Plugin Qt per Eclipse

L'integrazione di *Qt* in *Eclipse* consente ai programmatori di creare le applicazioni in modo veloce. I punti di forza di questo potente *Plug-In* sono l'integrazione con i principali strumenti *Qt* precedentemente descritti, il notevole supporto nella creazione dei file di progetto, il completamento automatico durante la digitazione, e molti altri. Il potente debugger e l'ambiente che l'IDE offre in generale rendono questo strumento uno dei migliori editor per lo sviluppo di applicazioni *Qt*. Per gli sviluppatori *Java* e *C++* questo modulo - che estende le funzionalità dei *Plug-In* base di *Eclipse* - al momento attuale risulta la soluzione migliore tra tutte quelle testate.

Umbrello UML Modeller

Umbrello UML Modeller - figura 3.12 - è uno strumento per la realizzazione di diagrammi UML che si è rivelato essere di grande aiuto nel processo di sviluppo del software, specialmente durante le fasi di analisi e progettazione, in cui la creazione di *diagrammi delle classi*, *dei casi d'uso* e dei *diagrammi di interazione* era spesso indispensabile. Umbrello ha contribuito a rendere il progetto chiaro soprattutto per quanto riguarda la documentazione delle scelte progettuali tra le parti interessate.

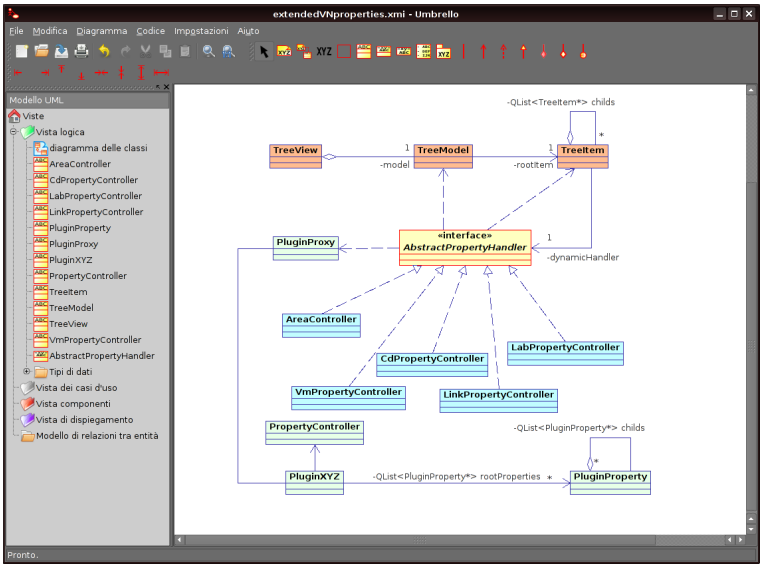


Figura 3.12: Un esempio d'uso del tool *Umbrello*.

SVN e Google Code

A supporto dell'intero sviluppo si sono utilizzati gli strumenti per la gestione del codice *SVN* e *Google Code*. In particolare, SVN è stato usato come strumento principale per gestire la continua evoluzione dei documenti relativi al progetto, come il codice sorgente del software, i disegni tecnici ed

i diagrammi, la documentazione, e altre informazioni importanti su cui si è lavorato.

Nella maggiorparte dei progetti di sviluppo software, più sviluppatori lavorano in parallelo sullo stesso codice. Se due sviluppatori tentano di modificare lo stesso file contemporaneamente, in assenza di un metodo di gestione degli accessi, essi possono facilmente sovrascrivere o perdere le modifiche effettuate contestualmente. La maggior parte dei sistemi di controllo versione possono risolvere questo problema in due diversi modi. Alcuni di questi prevengono i problemi dovuti ad accessi simultanei semplicemente bloccando i file (mediante l'utilizzo di un *lock*), cosicché solamente uno sviluppatore alla volta riceve i permessi di accesso in scrittura alla copia di quel file, contenuta nel *repository* centrale. Altri, come CVS, permettono a più sviluppatori di modificare lo stesso file contemporaneamente e forniscono gli strumenti per combinare in seguito le modifiche, tramite operazioni di *merge*.

Subversion (noto anche come SVN) è un sistema di controllo versione progettato da *CollabNet Inc.* nel 2000 con lo scopo di essere il naturale successore di CVS, oramai considerato superato.

Google Code, ed in particolare la sezione *Project Hosting*, non è stato per questo progetto solo il repository dedicato all'SVN, ma ha messo a disposizione dei programmatori una serie di strumenti tra cui un *wiki* ed una sezione *issues* - o comunemente chiamato *Bug Tracker* -, molto utili come luogo di discussione e informazione per tutti quelli interessati al progetto.

Capitolo 4

Scenaro per la configurazione avanzata di reti virtuali

Nei precedenti capitoli è stato descritto il lavoro svolto durante l'attività di tesi. In questo capitolo si vuole illustrare il funzionamento dell'ambiente sviluppato, suddividendo l'esposizione in uno scenario reale analizzando in tre fasi.

4.1 Caso di studio: realizzazione e configurazione di un Lab tramite *VisualNetkit*

La prima fase illustra le azioni da effettuare per la costruzione di un semplice laboratorio composto da una rete LAN.

Lo step successivo descrive le mansioni da eseguire sul laboratorio precedentemente cerato per riuscire a configurare i nodi presenti, secondo le esigenze dell'utente. Vengono qui pertanto mostrate le potenzialità del nuovo *Plug-In* framework, sottolineando com'è possibile applicare dettagliatamente le configurazioni. In mancanza di moduli complessi - come *Zebra* o *Quagga* -, è stato utilizzato un *Plug-In* di test che possiede una strut-

tura delle properties estesa e altamente dinamica, priva di un significato logico.

L'ultima parte espone alcuni test di funzionamento sulla rete precedentemente realizzata, tramite l'ausilio dell'ambiente di emulazione offerto da *NetKit*.

4.1.1 Realizzazione di un laboratorio

L'obiettivo di questa fase di apertura è la realizzazione di una topologia di rete semplice, composta da quattro *Virtual Machine* connesse a stella ad un unico dominio di collisione.

In primo luogo occorre creare un nuovo laboratorio scegliendo dal menu *File* la voce *New Lab*, oppure tramite l'uso della shortcut **Ctrl+N**. Da questo momento fino alla chiusura del Lab, *VisualNetkit* è pronto per la creazione della rete. In figura 4.1 è discritta la struttura delle docks che compongono l'interfaccia utente.

Le docks presenti - che comprendono anche le tool-bars -, sono tutte configurabili. L'utente può comodamente disporle a suo piacimento, nonché nascondere quelle che ritiene superflue.

Successivamente è possibile iniziare a costruire la topologia della rete in esame: utilizzando le apposite azioni presenti nel menu *Graph*, l'utente può inserire nella scena grafica elementi quali Virtual Machines, Links, Collision Domain e Aree. Dapprima è necessario inserire le quattro *Virtual Machine* nella scena, attivando per l'host "PC1" anche il *Plug-In Test*, che verrà configurato in un secondo momento.

Al termine dell'operazione è possibile inserire il dominio di collisione

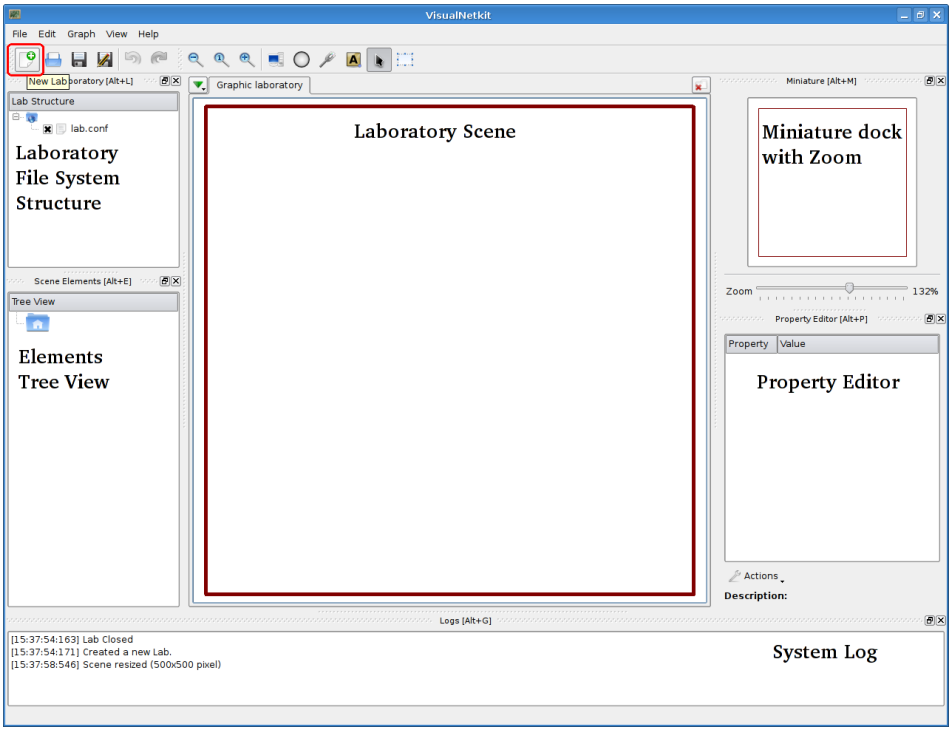


Figura 4.1: Docks in *VisualNetkit*

posto a "centro stella", che andrà a collegare tutti gli host virtuali precedentemente creati. Infine, il lavoro sarà completato dall'inserimento dei links in cui verranno attivati i *Plug-In IPv4* e *MAC*. Figura 4.2.

Una volta realizzata la topologia di rete desiderata, il tool permette l'aggiunta di aree che possono essere utilizzate come etichette, oppure come aggregatori di elementi.

4.1.2 Configurazione avanzata del laboratorio

Spesso l'utente ha la necessità di configurare i servizi presenti all'interno degli host virtuali per garantire un corretto funzionamento delle macchine, al fine di effettuare con successo i test desiderati. *VisualNetkit* offre questa possibilità mediante l'uso della dock delle proprietà. Quando si seleziona

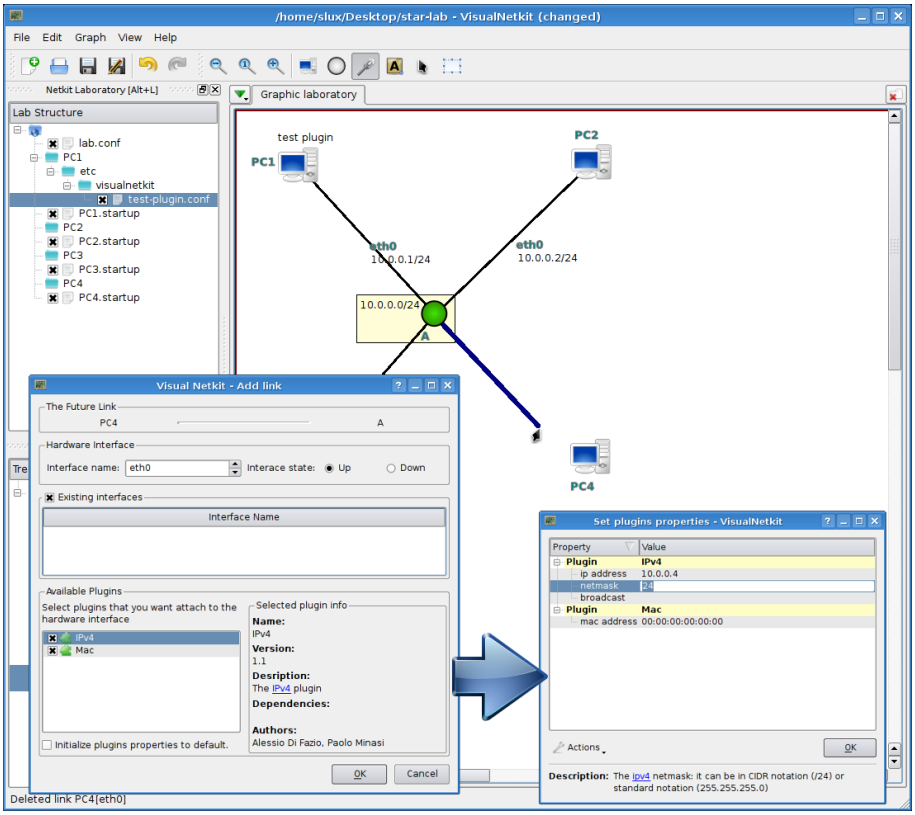


Figura 4.2: Inserimento dei links.

un elemento della scena grafica con un doppio click, il sistema provvede a renderizzare le informazioni dell'oggetto all'interno della property dock. Possiamo quindi trovare tutte le propietà presenti all'interno dei *Plug-In* attivi per l'entità selezionata.

Alcuni moduli più complessi (nel caso dell'esempio il *Plug-In Test*) prevedono la possibilità di poter manipolare la struttura delle properties, in particolare questo avviene inserendo o eliminando sotto-attributi (figura 4.3).

Si vuole quindi modificare la struttura del modulo *Test* attivo sulla macchina virtuale "PC1". Come mostrato, viene prima inserata una sotto-

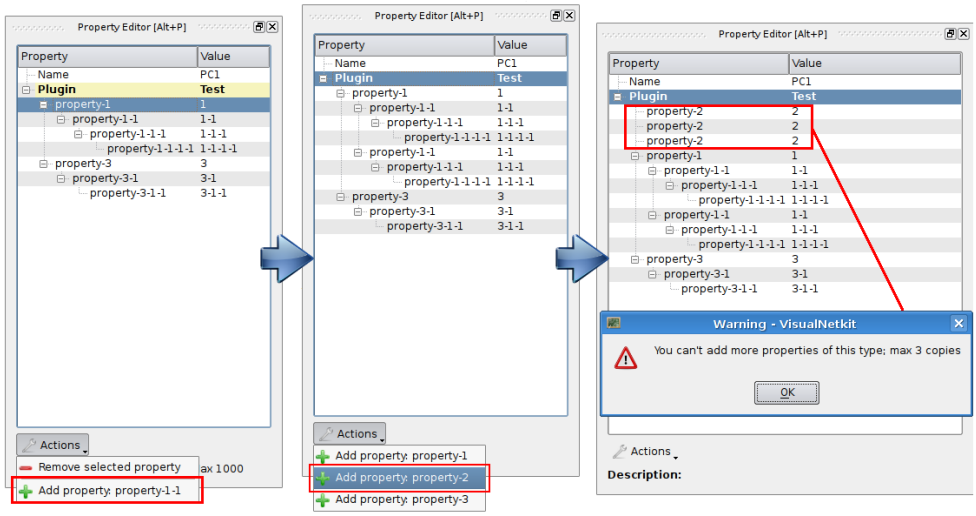


Figura 4.3: Modifica della struttura delle properties di un *Plug-In*.

proprietà dell'attributo "property-1", successivamente si tenta di inserire tre copie della proprietà "property-2" che il modulo prevede. Al tentativo di inserimento della quarta copia di quest'ultima, il *Plug-In* restituisce un errore in quanto non sono previste più di tre duplicati per la proprietà in esame.

Tramite l'apposito bottone "Actions" è possibile, per ogni proprietà di un modulo, eliminare o aggiungere le sotto-property quando consentito. Quest'ultimo controllo, come quello inerente al controllo di cardinalità di una property poc'anzi citato, è interamente affidato al *Plug-In*.

Il modulo analizzato nell'esempio è puramente utilizzato per il testing. Tuttavia, si può facilmente immaginare che un *Plug-In* complesso, come *Zebra* o *Quagga*, possa avere una struttura simile a quella mostrata, il quale contiene molteplici attributi e sotto-attributi.

4.1.3 Sperimentazioni sul laboratorio esportato

Lo stadio finale della costruzione di un laboratorio mediante l'utilizzo di *VisualNetkit* si conclude con il salvataggio dello stesso sul *File System*. Attraverso la voce *Save As...* presente nel menu *File* è possibile selezionare la directory dove il Lab creato verrà esportato.

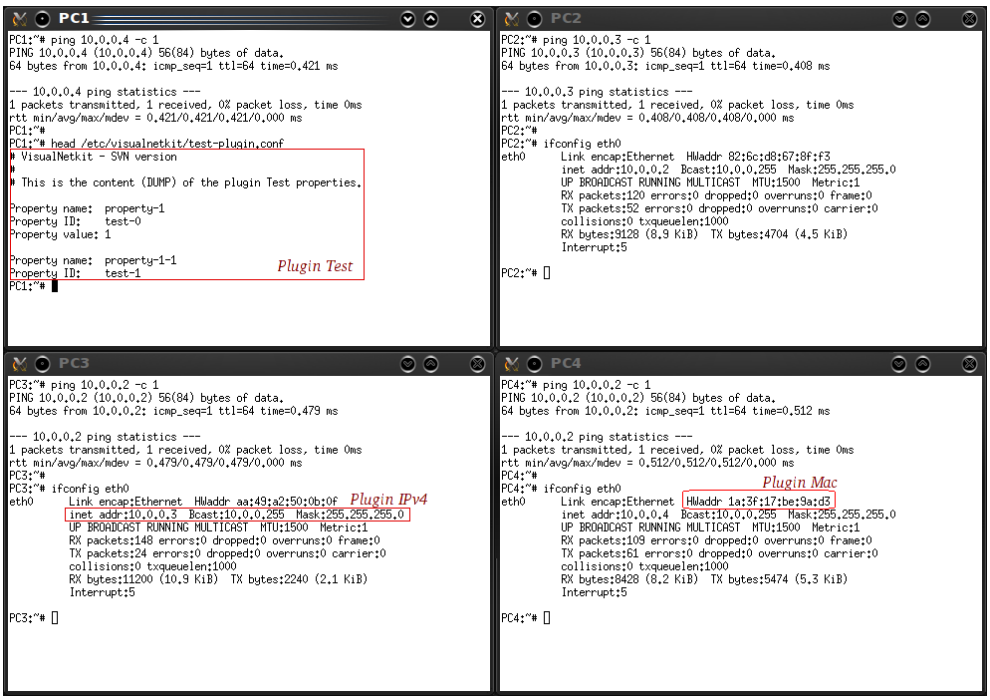


Figura 4.4: Il laboratorio avviato emulato tramite *NetKit*.

Effettuata questa operazione si può provvedere anche alla chiusura di *VisualNetkit*. Da questo momento in poi viene utilizzato *NetKit* per avviare la rete virtuale, in cui è possibile effettuare tutti i test che si desiderano.

Studiando il comportamento della rete precedentemente creata, è possibile osservare che il laboratorio esportato è completo in ogni dettaglio. Tutti gli host virtuali connessi ad un unico dominio di collisione riescono a colloquiare correttamente, come mostrato in figura 4.4. Inoltre, si può con-

statare che i *Plug-In* attivati sui links e sull'host "PC1" hanno contribuito tutti a caratterizzare la parte della rete di propria competenza.

Ancora, è possibile immaginare un'estensione dell'architettura al fine di prevedere la possibilità di avviare un laboratorio direttamente da *Visual-Netkit*, avendo la possibilità di mostrare all'utente una rappresentazione grafica dei terminali virtuali.

In ultima istanza, potrebbe risultare necessaria ed innovativa la realizzazione di un meccanismo che permetta ai vari moduli di comunicare e cooperare tra di loro; si pensi ad esempio, allo scenario in cui un modulo (attivo su una *Virtual Machine*) volesse conoscere tutti gli indirizzi IPv4 attivi sulle interfacce hardware presenti sull'host.

APPENDICE: Operazioni per la Creazione di un Nuovo Plugin

Questa parte del lavoro è dedicata a coloro che sono interessati alla creazione di un *Plug-In* per *VisualNetkit*. Viene fornita una descrizione dettagliata sull'interfaccia dei moduli e sulle singole funzioni da implementare e viene inoltre mostrato il supporto di alcune classi che possono essere utilizzate all'interno del *Plug-In* per semplificare alcune azioni particolarmente complesse.

Plugin Interface

A partire dalla versione 1.1, l'interfaccia dei moduli è stata modificata per consentire l'estensione del sistema che gestisce le properties, come precedentemente descritto. Qui di seguito è riportata la classe astratta pura che descrive l'interfaccia di un *Plug-In*:

```
1  /**
2   * PluginInterface.h
3   */
4  class PluginInterface
5  {
6  public:
7      virtual ~PluginInterface() {};
8
9      virtual bool saveProperty(QString propUniqueId, QString propValue,
10             QString *pluginAlertMsg = NULL) = 0;
```

```
11
12      virtual QString getXMLResource() = 0;
13
14      virtual QMap<QString, QString> getTemplates() = 0;
15
16      virtual QList<PluginProperty*> getPluginProperties() = 0;
17
18      virtual PluginProxy* getProxy() = 0;
19
20      virtual void setProxy(PluginProxy* p) = 0;
21
22      virtual void setGroupID(qint32 id) { Q_UNUSED(id) };
23      virtual qint32 getGroupID() { return -1; };
24
25      virtual QString getDefaultGraphisLabel() = 0;
26
27      virtual QString getName() = 0;
28
29      virtual bool init(QString laboratoryPath) = 0;
30
31      virtual QString deleteProperty(QString propertyUniqueId) = 0;
32
33      virtual QPair<PluginProperty*, QString> addProperty(
34          QString propertyIdToAdd, QString parentPropertyUniqueId) = 0;
35
36  };
37
38  typedef PluginInterface* createPlugin_t();
39  typedef void destroyPlugin_t(PluginInterface*);
```

Le righe di codice più importanti di questa classe sono gli ultimi due "typedef". Ogni *Plug-In* concreto deve necessariamente avere al suo interno l'implementazione delle factory di creazione e distruzione, definite come *extern "C"*. Queste due funzioni sono usate dalla classe *LoaderFactory* che risolve i simboli **createPlugin** e **destroyPlugin** permettendo di creare nuove instanze di un determinato modulo. Un tipico esempio di queste due funzioni è riportato qui di seguito:

```
1  /* Factory (creator) */
2  extern "C" PluginInterface* createPlugin()
3  {
4      return new YourContretePluginClass();
```

```
5  }
6
7  /* Factory (destroyer) */
8  extern "C" void destroyPlugin(PluginInterface* p)
9  {
10     delete p;
11 }
```

Il resto delle funzioni da implementare e le loro mansioni, sono qui elencate:

saveProperty() questa funzione viene invocata dal sistema centrale quando una particolare proprietà viene modificata. La property è indicizzata dal proprio ID univoco ed inoltre viene passato un puntatore ad un *QString* che, in caso di errore, deve essere riempito da un messaggio di warning che verrà mostrato all'utente. Se tale stringa è un NULL pointer, la property deve essere salvata senza effettuare alcun tipo di controllo, ritornado sempre "true";

getXMLResource() ritorna una stringa che contiene il *Qt Resource* entry del file di configurazione del *Plug-In*.
Esempio `:/pluginName/xml-config`;

getTemplates() questa funzione ritorna una *QMap<QString, QString>* che contiene come chiave il path del file di configurazione a partire dalla root del laboratorio e, come valore, contiene il testo da scrivere all'interno del file indicato dalla chiave della mappa;

getPluginProperties() restituisce la lista contenente le root properties del *Plug-In*;

setProxy() dopo la creazione del modulo, il sistema passa il proxy al *Plug-In*: in questo momento è possibile effettuare post-operazioni quali validazione del file *XML* di configurazione e il salvataggio delle properties di base offerte dal property expert;

setGroupID() e getGroupID() non sono utilizzate allo stato attuale;

getDefaultGraphisLabel() fornisce la label da mostrare all'interno della scena grafica accanto all'elemento base di propria competenza, quando il modulo viene creato;

getName() ritorna il nome del *Plug-In*;

init() questa funzione è invocata quando un laboratorio viene caricato. Normalmente l'implementazione di questa parte prevede un azione di *parsing* delle informazioni celate all'interno dei vari files di configurazione, per ricostruire la struttura delle proprietà. Il valore booleano di ritorno attualmente non è utilizzato, ma verrà impiegato durante l'importing di un laboratorio: il plugin restituirà "true" se esistono i files di sua competenza e quindi deve rimanere attivo per quel determinato elemento;

deleteProperty() la rimozione di una property viene effettuata passando l'id univoco. Il *Plug-In* è tenuto a controllare se il parent della property passata possiede un numero di occorrenze minimo accettabile. In caso di errore la funizone ritorna un *QString* contenente una stringa di errore, altrimenti ritorna un *QString* vuota;

addProperty() l'inserimento di una property prevede il passaggio di due parametri: l'id univoco del parent, e l'id (del file *XML*) che descrive la nuova property. Se non vi sono errori la funzione restituisce una coppia "pair" che contiene come primo elemento il puntatore alla nuova *PluginProperty* appena creata, e una *QString* vuota come secondo elemento. In caso di fallimento (come ad esempio un controllo di occorrenza violato), la funzione ritorna una *QPair* che contiene NULL come primo elemento e la stringa che descrive l'errore come secondo elemento.

File di configurazione di un *Plug-In*

Un elemento importante che risiede all'interno del resource file di un modulo, è il proprio file di configurazione.

```
1  <plugin name="Plugin Name">
2    <global type="vm|cd|link" version="1.0" author="" dependencies="">
3      <![CDATA[This is a plugin.]]>
4    </global>
5    <properties>
6      <property id="" name="" default_value="" min="0|1" max="">
7        <![CDATA[description (anche codice HTML)]]>
8        <childs>
9          <property id="" name="" default_value="" min="0|1" max="">
10           <![CDATA[description]]></childs>
11         </property>
12       </childs>
13     </property>
14   </properties>
15 </plugin>
```

La sezione "global" descrive i meta-dati del plugin quali: il nome, il tipo di elemento a cui può essere attivato, autori, dipendenze e descrizione estesa.

Le proprietà sono così composte: ogni property deve essere identificata da un ID univoco, deve possedere un nome e un attributo di minima occorrenza pari a 1 o 0. Se si ricade nel primo caso (per l'attributo "min"), il property expert crea la property al momento di inizializzazione, altrimenti questa viene ignorata e può essere inserita in un secondo momento dall'utente. L'attributo di massimo può assumere valiri nel range 1 - 65536, ed indica il numero massimo di occorrenze di una property.

Questa struttura può essere modellata senza limiti particolari. L'utente può inserire un numero arbitrario di properties e sub-properties, in base alle proprie esigenze.

Plugin Proxy e Property Expert

Durante lo sviluppo di un *Plug-In* è molto importante conoscere le classi che sono accoppiate con il modulo stesso. Il *Plugin Proxy* è un intermediario tra l'estensione e il sistema centrale. Alcune funzioni utili sono riportate qui sotto:

getBaseElement() ritorna il puntatore al *QObject* che rappresenta l'elemento su cui è stato attivato il modulo. in caso di errore ritorna un NULL pointer;

getPropertyExpert() ritorna il riferimento al property expert;

changeGraphicsLabel() questa funzione è invocata quando dal *Plug-In* quando esso desidera cambiare la label grafica.

Property Expert

Un elemento importante che collabora alla gestione della struttura delle proprietà di un *Plug-In*, è la classe *PropertyExpert*. Questa offre le seguenti funzioni:

buildBaseProperties() legge le informazioni dal file di configurazione del modulo e ritorna un albero n-ario delle properties, descritto da una lista di root-properties;

parseXmlGlobalInfo() scansiona e restituisce le informazioni (meta-dati) di un *Plug-In*;

isXmlConfValid() valida il file di configurazione di un modulo;

searchProperty() restituisce il puntatore ad una property identificata tramite l'id univoco; ritorna un NULL pointer in caso di errore;

searchPropertiesById() effettua una ricerca delle properties che posseggono l'id (descritto come attributo "id" nel file *XML*) passato come argomento;

getPropertyInfo() fornisce le informazioni di una property leggendo il file *XML* dell'elemento che possiede l'id passato come argomento;

newProperty() factory di costruzione di una nuova property.

Conclusioni e sviluppi futuri

Nel presente lavoro sono state descritte le fasi di analisi, progettazione e realizzazione di un ambiente per la configurazione avanzata di reti virtuali emulate. Particolare attenzione è stata concessa allo studio dei sistemi esistenti al fine di coglierne pregi e difetti.

Nell'intero documento si è cercato di far comprendere i limiti dei vari ambienti di configurazione esistenti - che comprende anche la prima release di *VisualNetkit* -, mostrando un modello altamente flessibile e dinamico che andasse ad incrementare la configurabilità delle reti create tramite *VisualNetkit*.

Basando le successive fasi sul modello logico descritto, l'attività di tesi è proseguita con la progettazione e realizzazione di nuovi requisiti. Questi ultimi hanno portato il tool ad una maturità tale da poter affermare che allo stato attuale è possibile descrivere la quasi totalità dei servizi residenti in un host, mediante la creazione di *Plug-In* ad-hoc. Proprio questa struttura altamente modulare fa sì che *VisualNetkit* sia uno strumento unico nel suo genere, nonostante sia "giovane".

Grazie alle scelte metodologiche, progettuali e realizzative adottate come *Unified Process* e *eXtreme Programming*, il sistema si presta molto bene ad ulteriori raffinamenti, offrendo al programmatore un ambiente ben or-

ganizzato e coeso. Potenzialmente un lavoro di questo tipo può evolveresi senza trovare mai un punto estremo. Uno dei prossimi sviluppi sarà la possibilità da parte del tool di importare un laboratorio creato dall'utente senza l'ausilio di alcuno strumento. Un importante ostacolo in questo requisito è la composizione grafica della rete. La posizione dei nodi e degli archi dovrà essere disegnata dall'ambiente in modo intelligente, adottando quindi algoritmi di *Graph Drawing*.

Un laboratorio creato da un utente contiene svariate regole e servizi. Durante la fase di importing sarà quindi opportuno che *VisualNetkit* disponga di un elevato numero di moduli, potenzialmente attivabili, al fine di effettuare un import del Lab più accurata possibile. Saranno infatti i *Plug-In* stessi a decidere di rimanere attivi o meno, sulla base delle informazioni *parsate* dai files di configurazione presenti nel laboratorio in esame.

Affrontare un'esperienza di questo tipo fa sì che lo sviluppatore acquisisca ulteriori competenze per quello che concerne lo sviluppo di medie e grandi applicazioni.

Ringraziamenti

A Michela per il suo impagabile sostegno e aiuto.

A tutti gli amici con i quali ho condiviso questi anni universitari.

A Dario per avermi fatto avvicinare al mondo open-source.

Al Prof. Maurizio Pizzonia e Massimo Rimondini che mi hanno dato la possibilità di svolgere questo lavoro.

Ed infine, a tutti i professori del Dipartimento di Informatica e Automazione (*D.I.A.*) dell'Università degli studi di "Roma Tre", che con estrema professionalità hanno sempre svolto il proprio lavoro egregiamente.

Bibliografia

[1] M. Zec, M. Mikuc. *Operating System Support for Integrated Network Emulation in IMUNES*, to appear in Proceedings of the 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure / ASPLOS-XI, Boston, October 2004.

[2] Jean-Vincent Loddo, Luca Saiu. *Marionnet: A Virtual Network Laboratory and Simulation Tool*, SimulationWorks, Marseille (France), 2008

[3] Gabriel Astudillo Muñoz. *Descripcíon del software IMUNES para su utilizacíon en el Laboratorio de Redes y Sistemas Operativos*, www.elo.utfsm.cl/~elo324/doc/manual_imunes.pdf.

[4] Fabrice Bellard. *QEMU, a Fast and Portable Dynamic Translator*, 2005 USENIX Annual Technical Conference.

[5] *VNUML Tutorial*, www.dit.upm.es/vnumlwiki/index.php/Tutorial.

[6] *VNUMLGUI*, pagesperso.erasme.org/michel/vnumlgui.

[7] *Netkit*, www.netkit.org.

[8] *Zebra Documentation*, www.zebra.org/zebra/index.html

[9] *Qt4 Documentation - An Introduction to Model/View Programming*, doc.trolltech.com/4.4/model-view-introduction.html

[10] Massimo Rimondini. *Interdomain Routing Policies in the Internet: Inference and Analysis*, A thesis presented by Massimo Rimondini in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science and Engineering, Roma Tre University, Dept. of Informatics and Automation, March 2007.

[11] Massimo Rimondini. *Emulation of Computer Networks with Netkit*, Tutorial held during the 4th International Workshop on Internet Performance, Simulation, Monitoring and Measurement (IPS MoMe 2006) in Salzburg, February 2006.

[12] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*, Prentice Hall PTR, October 2004.

[13] Nick Rozanski, Eoin Woods *Software System Architecture: Working With Stakeholders Using Viewpoints and Prespectives*, Addison Wesley