UNIVERSITA' DEGLI STUDI DI NAPOLI FEDERICO II
Facoltà di Ingegneria
Corso di Studi in Ingegneria Informatica

tesi di laurea specialistica

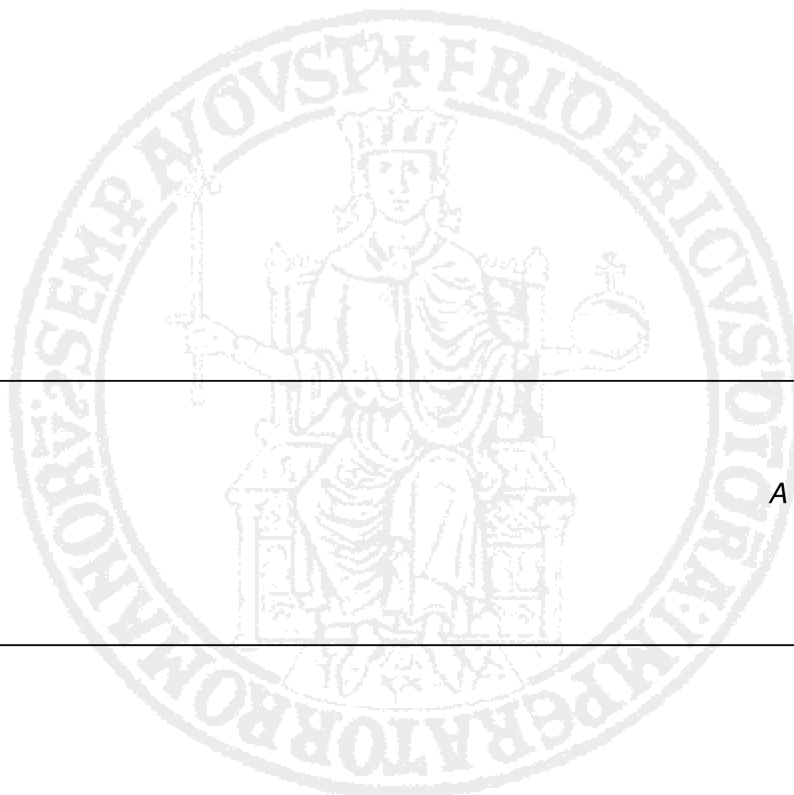
Un sistema per l'emulazione delle reti su cluster di macchine virtuali

Anno Accademico 2007/2008

relatore
Ch.mo prof. Roberto Canonico

correlatore
ing. Pasquale Di Gennaro

candidato
Roberto Bifulco
matr. 885/269



A chi lavora in silenzio
per il bene degli altri

Indice

Introduzione..6

Capitolo I
Problematiche di valutazione delle applicazioni e dei protocolli di rete......8

I.1 Tecniche di valutazione......9
I.1.1 Simulazione......9
I.1.2 Testbed......11
I.1.2.1 PlanetLab......11
I.1.3 Emulazione......12
I.2 Emulazione su cluster......13
I.2.1 Emulab......14
I.3 La gestione delle risorse hardware......15
I.4 Multiplexing delle risorse su cluster......17
I.4.1 Link multiplexing......17
I.4.2 Node multiplexing......18
I.5 Virtualizzazione......19
I.5.1 Full virtualization......21
I.5.2 Paravirtualization......22
I.5.3 Xen......23
I.5.4 Definizione di una virtual machine......24
I.5.5 Gestione di una virtual machine......25

Capitolo II
Il progetto Neptune......27

II.1 Organizzazione di Neptune......27
II.2 La gestione degli esperimenti......28
II.3 Utenti e ruoli......29
II.4 Il concetto di esperimento......29
II.5 Il concetto di topologia......33
II.6 Node multiplexing in Neptune......33
II.6.1 Fattibilità dell'approccio basato su template......35
II.7 Link multiplexing in Neptune......36
II.7.1 Virtual machine networking......38
II.7.1.1 Xen networking......39
II.7.2 Tecnologie per la simulazione dei link......40
II.7.2.1 Ebtables......40
II.7.2.2 Iptables......41
II.7.2.3 Traffic Control/Netem......41

II.7.3 Realizzazione della tecnica OLPI....42
II.7.3.1 Link parameters enforcement....43
II.7.3.2 Link creation....43
II.8 La rete di controllo....43
II.9 Architettura software....44
II.10 Problematiche di gestione delle reti degli esperimenti....46
II.11 Il modello dei dati....49
II.11.1 Modello dei dati della topologia....49
II.11.2 Modello dei dati dell'esperimento....53
II.11.3 Modello dei dati del virtual machine template....53
II.11.4 Modello dei dati del cluster....54
II.12 Validazione e completamento di una topologia....54
II.12.1 Validazione della topologia....55
II.12.2 Completamento della topologia....56

Capitolo III
Progettazione del Neptune Infrastructure Manager....57

III.1 Il workflow iniziale....58
III.2 Modello dei casi d'uso....60
III.2.1 Actors....61
III.2.2 Casi d'uso....61
III.2.2.1 Users Management....62
III.2.2.2 System Management....62
III.2.2.3 Experiments management....62
III.2.2.4 Experiment creation....62
III.2.2.5 Experiment administration....63
III.2.2.6 Experiment visualization....63
III.2.2.7 Topology definition....63
III.2.2.8 Physical Machine management....63
III.2.2.9 Virtual Nodes management....64
III.2.2.10 Virtual machine management....64
III.3 L'interfaccia utente....64
III.3.1 Log-in view....65
III.3.2 Operative View....66
III.4 Progettazione dell'architettura....68
III.5 Il sistema per la gestione delle autorizzazioni....72
III.5.1 Il command pattern....74
III.5.2 Architettura dell'IAS....75
III.5.3 Integrazione dell'IAS nell'architettura del NIM....76
III.6 La scelta della tecnologia....76
III.7 Introduzione dei vincoli tecnologici....78

Capitolo IV La realizzazione del Neptune Infrastructure Manager....80

IV.1 Neptune management library....81
IV.1.1 La definizione della struttura dei dati....82
IV.1.1.1 Struttura dati della topologia....83
IV.1.1.2 Struttura dati dell'esperimento....85
IV.1.1.3 Struttura dati del cluster....86
IV.1.2 Realizzazione del virtual cluster manager....86
IV.1.3 Realizzazione dell'emulation manager....88
IV.1.3.1 Topology enforcement....90
IV.1.3.2 Completamento della topologia....91
IV.1.3.3 Validazione della topologia....91
IV.1.4 La persistenza dei dati....93
IV.2 Identification and Authorization System....94
IV.2.1 L'interfaccia Operation....95
IV.2.2 La gestione dell'autenticazione....96
IV.2.3 Gestione delle operazioni....97
IV.2.4 Esempio d'uso....97
IV.3 Neptune Web Interface....99
IV.3.1 Lato client....100
IV.3.1.1 Widget e componenti grafici....102
IV.3.1.2 Il problema della serializzazione....103
IV.3.2 Lato Server....104

Conclusioni e sviluppi futuri....106

Bibliografia....108

Introduzione

Le reti di computer e le applicazioni realizzate tramite queste hanno assunto negli anni un'importanza crescente. Protocolli ed applicazioni di rete innovativi hanno supportato questo sviluppo in un ciclo continuo, in cui una nuova applicazione o protocollo apriva la strada a nuove prospettive che consentivano ulteriori sviluppi. La valutazione ed il *testing* di questi nuovi protocolli ed applicazioni, al crescere della complessità delle reti, è divenuto un compito sempre più arduo. Negli ultimi anni sono state presentate soluzioni diverse per fornire un sistema che rendesse possibile effettuare test in uno scenario di rete realistico, fra queste, ha riscosso particolare successo l'*emulazione di rete*, ed in particolare, l'*emulazione di rete basata su cluster*.

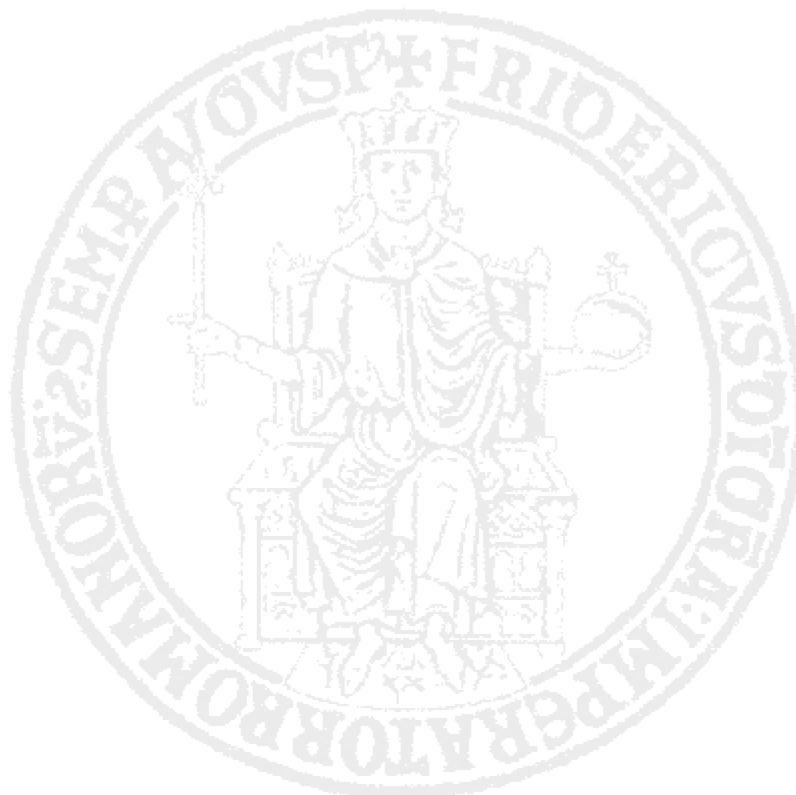
Questo lavoro di tesi ha contribuito alla progettazione e realizzazione di *Neptune*, una soluzione per l'*emulazione di rete basata su cluster* che, adoperando *hardware general purpose*, ha lo scopo di fornire un ambiente multi-utente, accessibile da remoto, per effettuare l'emulazione di diversi scenari di rete. *Neptune* offre la possibilità di *disegnare* topologie di rete che vengono poi emulate sfruttando tecnologie di *link* e *node multiplexing*, fornendo una gestione ottimale e scalabile delle risorse basata in larga parte sull'utilizzo di tecniche di *virtualizzazione*.

La tesi è costituita da quattro capitoli, così organizzati:

- **Capitolo I**: esamina le problematiche presenti nella valutazione dei protocolli e delle applicazioni di rete, per poi presentare le soluzioni attualmente proposte ai suddetti problemi. Sono presentate le tecniche di simulazione, *testbed* ed emulazione. Particolare attenzione è riservata all'emulazione basata su cluster ed

alle tecniche che permettono l'utilizzo efficiente delle risorse del cluster stesso: *link* e *node multiplexing*.

- **Capitolo II**: il secondo capitolo presenta il progetto Neptune. Sono qui trattate tutte le problematiche di gestione dell'infrastruttura di emulazione, dall'organizzazione dei nodi del cluster alle tecniche di realizzazione delle topologie di rete virtuale, alla gestione degli utenti.
- **Capitolo III**: analizza i requisiti funzionali e non del componente software *Neptune Infrastructure Manager*, che realizza gran parte delle funzionalità di *Neptune*. In questo capitolo sono inoltre trattati gli aspetti di progettazione di alto livello del componente.
- **Capitolo IV**: tratta della realizzazione del *Neptune Infrastructure Manager*, esaminando più o meno nel dettaglio aspetti caratterizzanti l'implementazione del software.



Capitolo I
Problematiche di valutazione delle applicazioni e dei protocolli di rete

Negli ultimi anni l'economia mondiale ha visto l'affermazione decisa di un prodotto rispetto agli altri, questo prodotto è la creazione e la distribuzione dell'informazione e, verosimilmente, tale andamento sarà sempre più accentuato in futuro.

I fattori di questo fenomeno sono da ricercarsi nello sviluppo della tecnologia, in particolare informatica, e quindi nella variazione della rappresentazione dell'informazione. Gli ultimi decenni hanno visto infatti una rapida "digitalizzazione" delle più disparate informazioni, con i conseguenti benefici legati alla possibilità di elaborare l'informazione tramite computer, di replicarla virtualmente all'infinito, di spostarla agevolmente e rapidamente. In questi benefici è concentrato il cuore della "Digital Revolution", infatti, l'informazione non deve essere più scambiata sotto forma di "atomi" (CD, libri, ecc.) ma è possibile rappresentare l'informazione tramite bit e poi spostare questi agevolmente tramite una rete.

Questi nuovi scenari introducono quindi nuove necessità, fra cui rientrano tecniche per rappresentare tutti i tipi di informazione come bit, e meccanismi per spostare questi bit ovunque, a basso costo, garantendo una certa qualità del servizio (*Quality of service*).

Il ruolo fondamentale in questo cambiamento è stato certamente giocato negli anni dalle tecnologie informatiche e di rete che sono cresciute notevolmente, fornendo nuovi mezzi per lo sviluppo delle reti e conseguentemente delle applicazioni che le sfruttano. Allo

stesso tempo sono sempre più diffusi dispositivi in grado di integrare più servizi complessi, come la fonia, il video, ecc. In concomitanza, i monopoli nelle telecomunicazioni sono andati svanendo, incrementando la competizione sul mercato.

Questa concomitanza di fattori economici e tecnologici ha comportato, infine, che le applicazioni ed i protocolli di rete divenissero elementi sempre più importanti, talvolta atti a realizzare anche servizi critici. Ad esempio, Internet ospita attualmente applicazioni di importanza notevole, basti pensare alle transazioni bancarie, od al ruolo vitale che svolge anche per il funzionamento di reti critiche, come quella elettrica, dei trasporti, ecc.

In questo contesto, risulta evidente come progetti che mirano a fornire servizi complessi richiedano un grande sforzo di progettazione e di valutazione delle caratteristiche prestazionali e di affidabilità, tuttavia, garantire tali caratteristiche richiede ormai infrastrutture di *testing* enormi e soprattutto costose. Di contro, queste grosse architetture di rete risultano spesso sottoutilizzate per gran parte del loro tempo nonché difficilmente gestibili e adattabili a scopi differenti.

Questa necessità di analizzare in modo adeguato nuove applicazioni e protocolli di rete ha portato alla definizione di opportune tecniche atte allo scopo. Tali tecniche differiscono per scalabilità, accuratezza e quantità di risorse impiegate: dai modelli matematici per la descrizione dei sistemi fino ai *testbed*, passando per tecniche come la simulazione e l'emulazione. La scelta di una tecnica errata può però degradare la qualità dell'esperimento e introdurre comportamenti inattesi . Nel seguito di questo capitolo è offerta una panoramica sulle tecniche più diffuse e degli sviluppi che si sono avuti negli ultimi anni.

I.1 Tecniche di valutazione

I.1.1 Simulazione

La simulazione [1] è un processo di imitazione delle operazioni eseguite nel tempo da un sistema reale. Rappresenta infatti la trasposizione, in termini logico-matematico-

procedurali, di un modello concettuale della realtà; tale modello concettuale può essere definito come l'insieme di processi che hanno luogo nel sistema valutato e il cui insieme permette di comprendere le logiche di funzionamento del sistema stesso. La simulazione consente di valutare e prevedere lo svolgersi dinamico di una serie di eventi susseguenti all'imposizione di certe condizioni da parte dell'analista. Gli scopi della simulazione sono molteplici: analisi delle prestazioni, determinazione delle criticità del sistema (attuali o future), confronto tra sistemi, progettazione di sistemi "ipotetici", ottimizzazione, *capacity planning*. Naturalmente, visti i molteplici scopi, esistono numerose aree di applicazione, che vanno dai sistemi di elaborazione e comunicazione, fino ai sistemi militari, sociali ed economici. Applicata all'ambito delle reti, la simulazione, fornisce un ambiente ripetibile e controllato per eseguire un esperimento, consentendo l'utilizzo di strumenti semplici da configurare per la progettazione e l'implementazione di protocolli a diversi livelli di astrazione. Inoltre, proprio grazie a questa facilità di configurazione, è possibile lavorare su un'ampia classe di esperimenti. In particolare, nella simulazione delle reti [2], è molto usata la tempificazione ad eventi discreti dove gli stati del modello cambiano solo in istanti discreti del tempo. L'accuratezza della simulazione varia in relazione al livello di astrazione del modello. Quando è utilizzato un alto livello di dettaglio, l'efficienza e la scalabilità della simulazione ne risentono. Bisogna quindi rinunciare a qualcosa in termini di efficienza se in un esperimento si vuole un alto livello di dettaglio e quindi un'elevata accuratezza. Al contrario, se si desidera una simulazione rapida bisogna tralasciare qualche dettaglio rinunciando alla piena accuratezza del modello da simulare. È necessario quindi effettuare un compromesso tra accuratezza ed efficienza. Lo scopo dei simulatori di reti è quello di modellare le reti del mondo reale: se un sistema può essere modellato, infatti, le caratteristiche del modello possono essere modificate ed i risultati possono essere quindi analizzati. I network simulator non sono però perfetti [3], infatti, essi non riusciranno mai a modellare in modo integrale la rete, tuttavia, maggiore sarà il grado di precisione del modello, più significante sarà la visione interna di come la rete stia lavorando e di come eventuali cambiamenti possano incidere sulla sua operatività. La

simulazione è in grado, in definitiva, di eliminare gran parte delle incognite relative a possibili inconvenienti che si possono verificare in seguito ad operazioni di modifica della rete originaria.

I.1.2 Testbed

Il *testbed* [4] è una piattaforma creata per condurre sperimentazioni in condizioni di lavoro reali. Permette il *testing* di applicativi in maniera rigorosa (tutte le componenti sono infatti reali), trasparente e replicabile. Sostanzialmente il *testbed* consiste in una riproduzione del sistema in laboratorio [5] con una buona fedeltà del modello, anche se meno accurato di quello descritto con la tecnica della simulazione. Soffre però di una limitata scalabilità dipendente dal costo economico dell'architettura di rete da installare, inoltre queste architetture di rete risultano spesso sottoutilizzate per gran parte del loro tempo nonché difficilmente gestibili e adattabili a scopi differenti.

I.1.2.1 PlanetLab

PlanetLab è un testbed globale di ricerca che supporta lo sviluppo di nuovi servizi di rete. L'uso di PlanetLab da parte di ricercatori o industrie è stato incentrato verso lo sviluppo di nuove tecnologie per l'archiviazione distribuita, network mapping, sistemi peer-to-peer, distributed hash tables, ecc. [6]

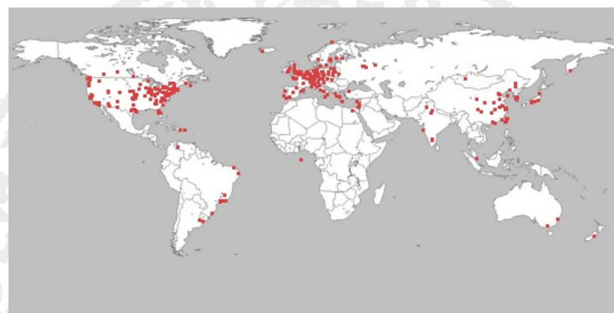


Figura I.1: Dislocazione dei nodi di Planetlab

PlanetLab, per effettuare il testing di servizi su larga scala, adopera 912 nodi dislocati in 473 siti. In ogni nodo sono istanziate delle macchine virtuali che permettono di eseguire più esperimenti in contemporanea e di ottimizzare le risorse disponibili. Ciascun esperimento viene eseguito in una "*slice*", ossia, adopera un insieme delle macchine

virtuali dislocate in un sotto-insieme dei nodi del testbed. Ciascuna di queste macchine virtuali è connessa alle altre tramite internet.

Per la realizzazione delle macchine virtuali PlanetLab adopera la *LightWeight Virtualization*, in cui viene garantito l'isolamento fra le diverse macchine virtuali, ma la virtualizzazione è ottenuta semplicemente modificando in parte il *kernel* del sistema operativo. Questo approccio costringe l'utilizzo dello stesso sistema operativo per tutte le macchine virtuali, limitando non solo la scelta dell'ambiente operativo ma anche le possibilità di modifica e personalizzazione degli aspetti di basso livello della gestione di rete, come la gestione del protocollo IP.

Per queste caratteristiche, PlanetLab è adoperato per lo più per lo sviluppo di applicazioni del livello applicazione dello *stack* protocollare di rete.

I.1.3 Emulazione

L'emulazione consente l'analisi delle dinamiche di un sistema reale attraverso un modello concettuale, permettendo l'interfacciamento del sistema con il mondo esterno. L'emulazione di una rete rappresenta dunque un approccio ibrido [7] che combina elementi reali (come hosts e implementazioni di protocolli) con elementi sintetici, astratti o simulati di una rete (come link, nodi intermedi e traffico di background). In relazione ai requisiti dell'esperimento e alle risorse disponibili vengono definiti quali elementi sono reali e quali sono parzialmente o pienamente simulati. In sostanza con il termine "emulazione" ci si riferisce alla capacità di inserire un simulatore all'interno di una rete reale. Di conseguenza essa fornisce un ambiente che è più vicino alla realtà rispetto a quello offerto dalla simulazione ed è un valido strumento per la valutazione delle prestazioni e delle funzioni di componenti reali sotto varie condizioni di funzionamento.

Poiché l'obiettivo dell'emulazione è far coesistere e cooperare elementi reali con elementi simulati, è necessario che questi ultimi siano capaci di interagire con i componenti di rete e le applicazioni reali. Questa necessità si presenta a causa della fondamentale differenza che sussiste fra simulazione ed emulazione: nella simulazione l'esperimento è condotto in

un tempo fittizio, totalmente slegato dalla realtà e dipendente soltanto dal simulatore; nell'emulazione l'esperimento è condotto nel tempo "reale", poiché gli elementi reali sono parte integrante dell'ambiente emulato.

I.2 Emulazione su cluster

Un *trend* che ha avuto molto successo negli ultimi anni nell'ambito dell'emulazione di rete è la *cluster based emulation*.

Un computer cluster [8] o più semplicemente cluster (dall'inglese grappolo) è sistema hardware modulare costituito da un insieme di computer connessi tramite una rete telematica ad alta velocità. Il cluster è generalmente dotato e di strumenti di amministrazione di base che ne semplificano la gestione di basso livello. Inoltre, è scalabile e si adatta alla crescita del business aziendale. I singoli moduli hardware sono inseriti in *rack* e possono essere gestiti e manutenuti anche da personale non super-specializzato. Utilizza componenti "standard" di mercato come gli *switch gigabit ethernet* economici e sostituibili con prodotti di altri vendor. Le unità computazionali non devono essere necessariamente dello stesso produttore, anche se questo può favorire la gestione di basso livello.

Lo scopo di un cluster è generalmente quello di distribuire una computazione molto complessa tra i vari computer componenti del cluster stesso. In sostanza un problema che richiede molte elaborazioni viene scomposto in sotto-problemi separati che poi vengono risolti in parallelo. Questo ovviamente aumenta la potenza di calcolo del sistema.

In ambito emulativo i cluster sono molto utilizzati per la realizzazione dell'emulazione di reti, poiché questi sistemi comprendono un largo insieme di componenti di rete (*link, switch, router*, ecc.) organizzati in un'unica infrastruttura, facilmente accessibile anche da remoto (con opportune interfacce). L'esempio più significativo di sistema di emulazione *cluster based* è certamente Emulab.

I.2.1 Emulab

Emulab nasce presso l'Università dello Utah e vanta svariate installazioni presso Università e altri Enti di Ricerca. Questa soluzione [9] fornisce un accesso integrato ad un vasto range di ambienti sperimentali: dalla simulazione all'emulazione fino al *testbed* su reti in larga scala, unendo tutti questi ambienti sotto una singola interfaccia utente, integrandoli in un'unica struttura, sforzandosi di preservare il controllo e la semplicità d'uso della simulazione, senza sacrificare il realismo dell'emulazione e della sperimentazione su reti reali. Tale struttura fornisce astrazioni, servizi e gestione delle etichette comuni a tutti, come l'allocazione e la designazione dei nomi da dare a nodi e link. Emulab maschera molte delle eterogeneità dei differenti approcci, mappando le astrazioni all'interno di meccanismi di *etichettamento* e gestione dei domini. L'*Emulab emulation testbed* consta di tre *sub-testbed* (interoperanti tra di loro qualora fosse necessario), e ciascuno di essi provvede ad un diverso obiettivo di ricerca:

- *Mobile wireless (robotic testbed)*
- *Fixed 802.11 wireless (per i test su reti wireless)*
- *Emulab classic*

Emulab classic è uno strumento di emulazione di reti di tipo *time-shared* e *space-shared* che raggiunge nuovi livelli in termini di semplicità di utilizzo. Diverse centinaia di PC in *rack*, organizzati in maniera opportuna dal punto di vista della sicurezza, dotati di strumenti *web user-friendly*, e guidati da script ns-compatibili [10] oppure *Java GUI*, permettono di effettuare configurazione e controllo delle macchine e dei *link* a livello *hardware*. A livello utente può essere definita la perdita dei pacchetti, il ritardo, la banda e la dimensione delle code. Anche il sistema operativo può essere completamente rimpiazzato in sicurezza da immagini personalizzate costruite opportunamente dai vari sperimentatori.

Come accennato in precedenza, in Emulab una topologia di rete può essere definita inglobando nodi reali (macchine fisiche del cluster) e nodi virtuali realizzati sul cluster. Per definire i link che collegano tali nodi Emulab adopera un'architettura basata su *Virtual*

LAN (VLAN). VLAN è una tecnologia che adopera *hardware* specifico per realizzare domini di *broadcast* separati fra i nodi, come se questi risiedessero sulla stessa LAN. Accanto alla rete virtuale vi è una specifica rete di controllo che permette agli utenti del sistema di accedere da remoto ai nodi della topologia da loro definiti. Idealmente tale rete dovrebbe essere trasparente ai nodi della topologia, per quanto, per vincoli tecnici non ancora superati, ogni nodo è di fatto a conoscenza della presenza di questa ulteriore rete.

I.3 La gestione delle risorse hardware

La necessità di ridurre il numero di risorse richieste per effettuare la valutazione delle applicazioni di rete è il motivo principale per cui nascono le tecniche di simulazione, *testbed* ed emulazione. Il problema dell'allocazione e della gestione ottimale delle risorse non ha una soluzione semplice. Generalmente nella *network emulation* bisogna realizzare un compromesso tra la quantità di risorse da utilizzare ed il realismo dell'esperimento, inteso come quantità di componenti virtuali e quantità di componenti reali utilizzati. Per affrontare questo problema è necessario:

- tener conto dei requisiti dell'esperimento e delle risorse disponibili;
- ottimizzare l'uso delle risorse hardware disponibili quando queste sono condivise.

La scelta di una tecnica errata può infatti degradare la qualità dell'esperimento e introdurre comportamenti inattesi. La simulazione, descritta nei precedenti paragrafi, permette di utilizzare la quantità minima di risorse possibile ma, per definizione, crea un sistema completamente artificiale dove la nozione di tempo è virtuale ed indipendente dal tempo reale [7][11].

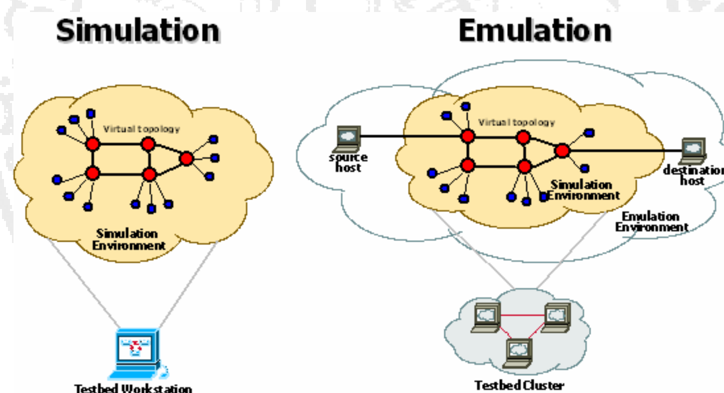


Figura I.2: Utilizzo delle risorse nella simulazione e nell'emulazione

La tecnica del *testbed*, invece, realizza le sperimentazioni in un sistema reale; il concetto di tempo di conseguenza è quello reale ma ciò comporta anche l'installazione di un'architettura di rete con componenti reali che, seppur garantendo l'affidabilità senza margini di errori, comporta un costo economico che può divenire enorme a scapito poi della scalabilità. C'è poi da evidenziare che il *testbed* comporta necessariamente un sottoutilizzo delle risorse *hardware* in quanto difficilmente in una sperimentazione verrà utilizzato tutta la memoria, la banda o la *cpu* disponibile. Il compromesso si trova, di conseguenza, tra queste due tecniche: con l'emulazione insomma. Come esposto in precedenza, l'emulazione è una soluzione ibrida che combina elementi reali con elementi simulati. La scelta dei componenti da simulare e di quali invece utilizzare come reali dipende dall'accuratezza richiesta nel *testing*. In conclusione, con l'emulazione è teoricamente possibile una migliore allocazione delle risorse. La soluzione ideale sarebbe avere un meccanismo *on demand*, in cui le risorse vengono allocate all'esperimento secondo le effettive esigenze.

Quest'ultima è la soluzione adoperata da Emulab, che gestisce le risorse del *cluster* al fine di fornire ad ogni utente l'ambiente di test desiderato. Emulab compie questa operazione adoperando VLAN per l'allocazione delle risorse di rete (intese come *link* fra nodi), mentre assegna un sottoinsieme dei nodi del cluster ad un esperimento, in base alla richiesta di risorse fatta per l'esperimento stesso. Tuttavia, anche con questo approccio il rischio è che un nodo del cluster non sia sfruttato appieno, ad esempio è possibile che un intero nodo sia impegnato in una computazione molto semplice, che, occupandolo, di fatto "spreca" le risorse non utilizzate. Per fronteggiare questa problematica. Emulab ha introdotto un sistema di virtualizzazione come tecnica di suddivisione delle risorse hardware di una macchina fisica in più nodi virtuali. Il sistema di virtualizzazione adopera il sistema operativo ***FreeBSD jail*** [9], che fornisce una basilare suddivisione ed isolamento di gruppi di processi, garantendo un più efficiente utilizzo delle risorse computazionali, per quanto, sia un sistema poco flessibile, legato in modo indissolubile ad una versione specifica del sistema operativo.

Negli ultimi anni, l'attenzione ad un utilizzo più razionale delle risorse di calcolo ha dato vita a diverse tecniche che permettono di realizzare la *multiplazione* di tali risorse. In particolare, le tecniche attraverso cui da una singola rete si ottengono più reti e da una singolo nodo più nodi prendono il nome di *link multiplexing* e *node multiplexing* e sono presentate nei prossimi paragrafi.

I.4 Multiplexing delle risorse su cluster

Il termine scalabilità [12] si riferisce, in termini generali, alla capacità di un sistema di "crescere" o "decrescere" (aumentare o diminuire di scala) in funzione delle necessità e delle disponibilità. Più precisamente, in informatica si utilizza il termine scalabilità di carico, ovvero la capacità di un sistema di incrementare le proprie prestazioni. Nell'ambito degli esperimenti di emulazione su sistemi cluster, la scalabilità dipende in massima parte dalla gestione ottimale delle risorse disponibili. Questa gestione può essere realizzata attraverso la scomposizione di ciascuna singola risorsa fisica in più unità virtuali (multiplexing delle risorse). Le due principali tecniche implementate per la multiplazione delle risorse in ambito *network emulation* sono: *link multiplexing* e *node multiplexing*.

I.4.1 Link multiplexing

Per condurre esperimenti di emulazione su cluster con un numero di link virtuali maggiore del numero dei link fisici disponibili è innanzitutto necessario implementare il link multiplexing [13], cioè permettere ai nodi del cluster, connessi tipicamente da una rete ad alta velocità, di scomporre lo stesso canale fisico condiviso in un certo numero di differenti link virtuali.

Il link multiplexing è realizzabile con differenti tecniche: abbiamo già fatto cenno alla tecnica adoperata da Emulab, che sfrutta VLAN; una alternativa è l'aliasing delle interfacce [11], con l'utilizzo dei meccanismi di traffic control propri del sistema operativo per dotare i link di caratteristiche di banda ridotta rispetto a quella fisica.

Ad esempio, nelle più recenti versioni del *kernel* di Linux (*kernel* 2.4.x in poi) viene messo a disposizione un insieme molto ricco di funzioni per il controllo del traffico utilizzabile tramite il programma utente TC (Traffic Control). Con TC il controllo del traffico comprende l'uso di vari componenti: queuing discipline (qdisc) ovvero discipline di accodamento, classi, filtri.

Le qdisc definiscono il modo in cui vengono accodati i pacchetti su quella interfaccia e sono quindi capaci di modellarne il traffico. La gestione dei diversi tipi di traffico viene effettuata tramite le classi, ognuna delle quali possiede una coda per i suoi pacchetti; a sua volta ogni classe può contenere una qdisc ed è quindi possibile creare strutture innestate.

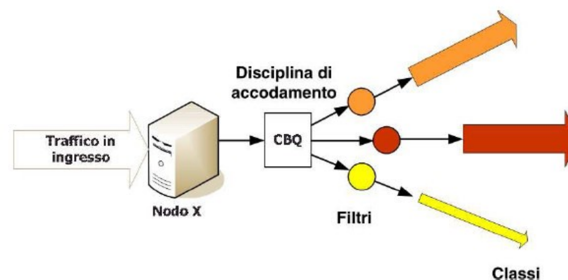


Figura I.3: Link multiplexing tramite ip aliasing

Ulteriori tecniche sono utilizzabili per ottenere il *link multiplexing* rispetto a quelle presentate sinteticamente in questo paragrafo. L'argomento sarà trattato con maggior precisione più avanti in questo stesso testo.

I.4.2 Node multiplexing

La suddivisione delle risorse computazionali fisiche in unità di risorse virtuali singolarmente gestibili e plasmabili viene chiamata *node multiplexing* [13]. Il *node multiplexing* consiste nell'emulare più di un nodo (di rete) sullo stesso nodo fisico del cluster. Ogni nodo virtuale (emulato) è perfettamente indipendente a livello di processo da un singolo nodo di rete fisico. In figura è mostrata questa operazione di scomposizione: le risorse fisiche di un elaboratore (nodo del cluster) vengono suddivise in più unità e assegnate ad entità consumatrici.

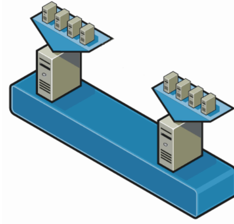


Figura I.4: Node multiplexing

Se queste entità consumatrici sono interi sistemi operativi (e tutto il software applicativo che in essi può girare) si può parlare di *hardware virtualization*: generalmente il sistema operativo delle macchine ospiti (le *virtual machine*) non percepisce il fatto di essere in un ambiente emulato ed opera come se avesse un hardware fisico completamente dedicato. Per la realizzazione del *node multiplexing* è necessario scegliere un meccanismo di virtualizzazione che si adatti alle proprie esigenze. Ovvero un meccanismo che abbia requisiti di:

- **scalabilità**: che non richieda un quantitativo eccessivo di risorse;
- **molteplicità**: che permetta l'esecuzione contemporanea e concorrente di più sistemi operativi;
- **isolamento**: che assicuri l'indipendenza tra gli ambienti di esecuzione relativi a software appartenenti a due macchine virtuali distinte multiplate sulla stessa macchina fisica;
- **trasparenza**: che non richieda modifiche all'oggetto da testare causate dall'ambiente di esecuzione introdotto dalla *node-virtualization*;
- **realizzabilità**: che consenta con gli attuali strumenti tecnologici di essere implementato senza difficoltà;
- **efficienza**: che il suo funzionamento sia stabile e privo di complicazioni;
- **semplicità**: che il suo utilizzo sia lineare.

I.5 Virtualizzazione

Le tecniche di virtualizzazione sono uno degli approcci fondamentali (se non l'unico) al *node multiplexing.* Con il termine virtualizzazione si intende la creazione di una versione virtuale di una risorsa normalmente accessibile direttamente. Qualunque risorsa hardware o software può essere virtualizzata: sistemi operativi, server, memoria, spazio disco. Un tipico esempio di virtualizzazione è la divisione di un disco fisso in partizioni logiche [14]. Ad oggi la virtualizzazione può essere effettuata a livello software ed a livello hardware.
Applicare una tecnica di virtualizzazione appropriata porta numerosi vantaggi:

- **isolation**: un processo appartenete ad una macchina virtuale è completamente isolato da processi appartenenti ad altre machine virtuali. La massima espressione dell’isolamento è la “kernel isolation”: un crash a livello kernel non porta conseguenze sulle altre macchine virtuali;
- **server consolidation**: si incrementa l’utilizzo delle risorse del server attraverso un uso più intelligente delle stesse. Si parla in questo caso di QoS isolation (quality-of-service isolation); si permette la coesistenza di sistemi operativi differenti su un’unica piattaforma;
- **lifecycle management**: strumenti avanzati di gestione delle macchine virtuali che possono essere create, accese, messe in pausa, spente e distrutte via software;
- **templating e cloning**: una macchina virtuale, una volta configurata, anche in termini di software applicativi, può diventare un *template* e clonata un numero infinito di volte;
- **legacy application support**: le macchine virtuali permettono l’abbandono dei sistemi *legacy* e la coesistenza delle applicazioni *legacy* con i nuovi sviluppi sullo stesso hardware;

Nel caso specifico del *node multiplexing* orientato alla *network emulation*, la virtualizzazione d'interesse assume il nome di ***virtualizzazione di sistema***. In questo caso lo strato software che permette la virtualizzazione viene detto Virtual Machine Monitor (VMM) oppure Hypervisor. Questo layer consente ad una singola macchina fisica di supportare l’esecuzione di ambienti multipli contemporaneamente.

Tipicamente il sistema operativo delle macchine ospiti (le virtual machine) non è al corrente del fatto che è in funzione in un ambiente emulato ed opera come se stesse girando su di un hardware fisico ad esso completamente dedicato. Il virtual machine monitor (ad esempio Xen) deve operare in maniera trasparente senza pesare con la propria attività sul funzionamento e sulle prestazioni dei sistemi operativi e svolgere attività di controllo al di sopra di ogni sistema, permettendone lo il monitoring e debugging delle attività e delle applicazioni in modo da scoprire eventuali malfunzionamenti ed intervenire repentinamente. I requisiti richiesti a questo scopo sono quelli di compatibilità, performance e semplicità.

Le tecniche di virtualizzazione si distinguono generalmente in:

- ***emulation;***
- ***full virtualization;***
- ***paravirtualization;***
- ***OS virtualization.***

L'ordine in cui sono presentate le tecniche è decrescente per livello di virtualizzazione: le prime emulano il comportamento dell'hardware grezzo, offrendo il massimo livello di virtualizzazione e di isolamento possibile, al prezzo di un maggiore consumo delle risorse computazionali e di memoria della macchina fisica sottostante, e quindi, a prezzo di una minore scalabilità. Queste tecniche differiscono nella complessità dell'implementazione, nel numero di sistemi operativi supportati e, come già accennato, nelle performance. In seguito verranno brevemente descritte le tecniche di full *virtualization* e *paravirtualization*, che sono di maggior interesse per i nostri scopi.

I.5.1 Full virtualization

Con la *native virtualization* o *full virtualization* [15], la macchina virtuale simula la maggior parte dell'hardware e permette di far girare sistemi operativi che siano stati compilati per quel tipo di cpu senza alcuna modifica. Esempi di software che implementano la *full virtualization* sono **VMware**, **VirtualBox** e **Qemu**. **Vmware** e

VirtualBox virtualizzano completamente l'hardware, ma non emulano la cpu, quindi il codice macchina del sistema operativo virtuale è eseguito in maniera nativa sulla cpu fisica. La trasparenza e l'isolamento sono complete e sullo stesso sistema operativo ospite (quello installato direttamente sulla macchina fisica) possono coesistere più sistemi operativi (Virtual OS) che girano incapsulati in macchine virtuali: i Virtual OS non necessitano di alcuna modifica. Tale approccio permette una gestione granulare delle risorse con la possibilità di implementare tecniche di QoS (Quality of Service) a livello di virtual machine. Tuttavia, l'*overhead* dovuto all'elevato livello di virtualizzazione lo rende meno scalabile rispetto ad altre soluzioni.

I.5.2 Paravirtualization

Nella *para-virtualization* la macchina virtuale non simula l'hardware ma offre una speciale API (Application programming interface) che richiede modifiche del sistema operativo. La paravirtualizzazione [16] agisce direttamente sull'hardware in modo da gestire la condivisione delle risorse destinate alle varie virtual machine. Il sistema di paravirtualizzazione si differenzia per il differente approccio utilizzato. In questo caso non c'è l'emulazione del processore e l'*overhead* è molto basso, ma un eventuale *crash* del sistema di paravirtualizzazione porterebbe in crash anche tutte le virtual machine. La paravirtualizzazione prevede che il "sistema di virtualizzazione" esponga ad ogni macchina virtuale interfacce hardware simulate funzionalmente simili, ma non identiche, alle corrispondenti interfacce fisiche: "il sistema di virtualizzazione" fornisce una libreria di chiamate (*Virtual Hardware API*) che implementa una semplice astrazione delle periferiche.

Occorre necessariamente modificare il *kernel* ed i driver dei sistemi operativi ospite per renderli compatibili con la *Virtual Hardware API* del sistema di virtualizzazione utilizzato. La complicazione di dovere modificare il kernel dei sistemi ospite viene ripagata con un maggiore semplicità del sistema di virtualizzazione che permette un incremento della velocità di elaborazione. Uno dei monitor di macchine virtuali che sfrutta

lo strumento della paravirtualizzazione è **Xen**, di cui si tratterà con maggior dettaglio nel seguito.

I.5.3 Xen

Xen [17] è un monitor di macchine virtuali open source, rilasciato sotto licenza GPL per piattaforma IA-32, x86, x86-64, IA-64 e PowerPC 970, sviluppato presso il Computer Laboratory dell'Università di Cambridge. Tra le sue caratteristiche peculiari sono da evidenziare:

- macchine virtuali con performance quasi native;
- pieno supporto architetture x86 (32 bit) con PAE e senza, x86 (64 bit);
- supporto per tutto l'hardware compatibile con i driver Linux;
- vCPU multiple supportate in ogni macchina guest (Windows, Linux);
- allocazione dinamica risorse tramite hot-plug vCPU (se supportato dal SO guest);
- migrazione delle macchine virtuali a caldo con zero-downtime;
- supporto istruzioni specifiche per la virtualizzazione dei processori Intel (Intel-VT) e AMD (AMD-V);

Xen è sostanzialmente un sottile strato *software* posto sopra l'*hardware*. Questo strato è chiamato *hypervisor* e si interpone tra le macchine virtuali e l'*hardware*.

L'*hypervisor* schedula gli accessi alle risorse fisiche della macchina ospite, da parte delle varie istanze delle macchine virtuali. Il primo sistema operativo *guest*, chiamato nella terminologia Xen "Domain-0", è caricato automaticamente quando l'*hypervisor* si avvia e ottiene speciali privilegi di gestione e l'accesso diretto all'*hardware* della macchina reale. L'amministratore di sistema effettua il login al Domain-0 per avviare qualunque operazione sui sistemi operativi guest, chiamati "Domain-U". Versioni modificate di Linux, NetBSD e Solaris possono essere usate come Domain-0 [fig. 2.3.3_b]. Infatti la tecnologia Xen richiede delle modifiche al kernel del sistema ospite e di conseguenza alcuni sistemi operativi proprietari che non rendono disponibile il codice sorgente non sono utilizzabili come sistemi virtuali. Recentemente, grazie ad uno sforzo congiunto con

Intel e AMD, sono state sviluppate tecnologie per abilitare la paravirtualizzazione a livello hardware attraverso una serie di estensioni integrate nei processori di ultima generazione (Intel-VT e AMD-V) che permettono, da una parte, di beneficiare della paravirtualizzazione ed incrementare le performance, mentre dall'altra non richiedono alcuna modifica del sistema operativo virtuale.

Nei successivi paragrafi verranno brevemente descritti i processi di creazione e gestione di una macchina virtuale in ambiente Xen, è bene precisare che tali processi, per quanto in questo contesto facciano riferimento soltanto a Xen, sono pressoché comuni a tutte le tecnologie di virtualizzazione.

I.5.4 Definizione di una virtual machine

Il processo di creazione di una *virtual machine* richiede che siano definite le informazioni sulle caratteristiche *hardware* della *virtual machine* e che sia fornito il file, o un qualsiasi altro dispositivo di memoria, che contiene la configurazione *software*.

La configurazione *hardware* è in genere specificata attraverso un opportuno file di configurazione. Tale file contiene informazioni quali la quantità di memoria RAM assegnata alla virtual machine, il numero e le caratteristiche delle interfacce di rete, ed ulteriori parametri tipicamente legati all'*hypervisor*. Ad esempio, in Xen è possibile specificare in che modo le interfacce di rete virtuali possono accedere alla rete esterna.

Come detto, la configurazione *software* può invece essere specificata in diversi modi. Di nostro interesse è la definizione tramite *file immagine*. Un file immagine non è altro che un file che contiene l'intera configurazione di un *file system*, dai file di installazione del sistema operativo a quelli delle applicazioni. Ad esempio, per un sistema *Windows*, un file immagine potrebbe contenere la partizione "C" con il suo intero contenuto.

Tipicamente, nel file di configurazione *hardware* è anche indicata quale immagine adoperare per la *virtual machine*.

Una volta specificate entrambe queste informazioni, è possibile allocare la *virtual machine* sull'*hypervisor* adoperando gli opportuni sistemi messi a disposizione da

quest'ultimo. In Xen, ad esempio, si può adoperare il tool *xm*, la cui sintassi prevede semplicemente:

```
xm create –c configFileName
```

Eseguito il precedente comando, l'*hypervisor* avvierà la macchina virtuale e ne gestirà l'esecuzione.

I.5.5 Gestione di una virtual machine

In qualsiasi momento l'utente può intervenire attraverso l'*hypervisor* per monitorare e gestire le *virtual machine.*

Durante il monitoraggio è possibile ottenere informazioni quali la quantità di memoria RAM adoperata dalla *virtual machine*, l'utilizzo del processore, ecc. Inoltre, è possibile visualizzarne l'attuale stato. A tal proposito, è bene specificare quali sono i possibili stati di una *virtual machine*:

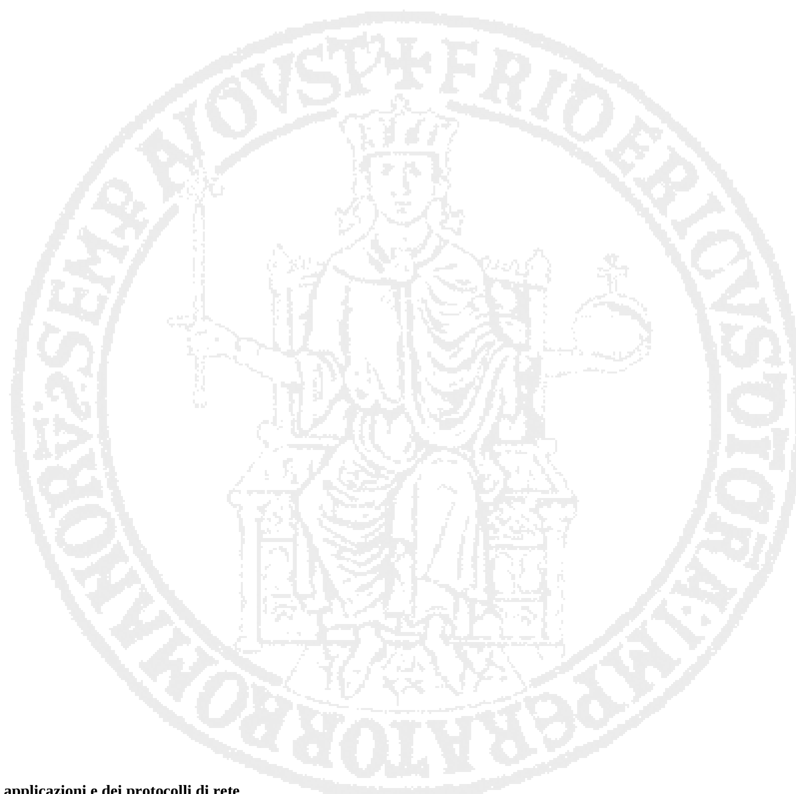
- ***Running***: la *virtual machine* è in esecuzione. In questo stato viene consumata sia la memoria RAM che la *cpu* della macchina fisica *hoster*;
- ***Paused***: la *virtual machine* e tutte le sue strutture dati sono caricate nella memoria centrale dell'*hoster* ma non c'è consumo di *cpu*. E' come se la *virtual machine* si trovasse in una fase di "pausa", ossia, tutte le elaborazioni sono sospese, ma tutte le informazioni conservate fino a quel momento nella memoria centrale continuano a persistere. Quando la *virtual machine* abbandona lo stato di *paused* per tornare *running*, tutte le elaborazioni riprendono dal punto in cui erano state interrotte;
- ***Shutdown***: la *virtual machine* è spenta, ossia, non occupa né memoria RAM né cpu. L'immagine è salvata su disco, la macchina può quindi essere ripristinata sempre in un secondo momento.

L'ordine in cui sono presentati gli stati non è casuale, infatti, *running* è lo stato che consuma il maggior numero di risorse, *shutdown* il minor numero. Si potrebbe anche definire uno stato *destroyed*, in cui la *virtual machine* è di fatto non più esistente, ossia, è

stata eliminata la sua immagine dal disco. Tale stato è stato omesso perché non è propriamente uno stato della *virtual machine*, in quanto la stessa non è più esistente.

E' bene precisare che quelli presentati sono gli stati comuni alla maggior parte degli *hypervisor*, che possono tuttavia adottare nomenclature differenti, o introdurre stati particolari come *crash, blocked*, ecc.

L'*hypervisor* fornisce un insieme di funzioni per gestire gli stati delle *virtual machine*, in particolare, tramite l'*hypervisor* si può mettere in pausa una macchina, avviarla, spegnerla, ecc.



Capitolo II
Il progetto Neptune

Neptune (Network Emulation for Protocol TUNing and Evaluation) è un progetto che ha l'obiettivo di realizzare un emulatore di rete flessibile, orientato al *multiplexing* delle risorse di calcolo e di memorizzazione, sviluppato all'Universita' di Napoli, che adopera l'approccio *cluster based* [11].

Il progetto prende molte delle assunzioni di base da Emulab [9], tuttavia, ha usato queste assunzioni solo come punto di partenza per lo sviluppo di un sistema che si differenzia sostanzialmente da quest'ultimo. Fin da principio, infatti, Neptune ha assunto la virtualizzazione come tecnologia chiave per la realizzazione di complessi sistemi di rete e, anziché basarsi su *hardware* specializzato per riprodurre il funzionamento dei sistemi reali, persegue lo scopo di evitare l'adozione di una qualsiasi soluzione basata su hardware specifico, realizzando, di fatto, un sistema capace di essere integrato in una qualsiasi infrastruttura *general purpose*.

In questo capitolo sono esposte le problematiche affrontate da Neptune e le soluzioni proposte.

II.1 Organizzazione di Neptune

L'organizzazione di Neptune segue un approccio centralizzato alla gestione dell'intero sistema. Uno dei nodi del cluster, che definiremo ***Neptune Manager*** (a cui ci riferiremo anche con il solo *Manager*), fornisce i servizi fondamentali (ad esempio *dhcp, dns, nfs*, ecc.) ed ospita il software di gestione della configurazione del sistema.

Da un punto di vista logico si possono immaginare due livelli all'interno del cluster: un livello su cui risiede il Manager, responsabile della realizzazione dei servizi di gestione del cluster e della comunicazione con l'esterno; ed un livello su cui risiedono tutti gli altri nodi, che rappresenta l'infrastruttura di emulazione vera e propria (Figura II.1).

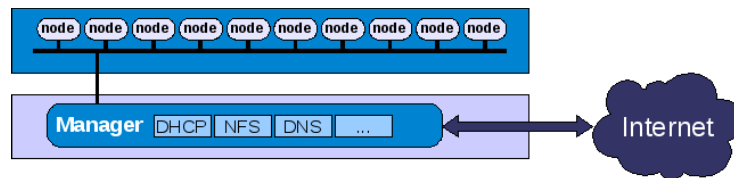


Figura II.1: Architettura di Neptune

Sia il Manager che gli altri nodi risiedono su di una rete comune, che viene definita *rete di controllo*, adoperata dal Manager per accedere a tutti gli altri nodi del cluster. L'accesso all'esterno da parte dei nodi non è garantito direttamente, ma è necessario adoperare sempre il Manager come tramite. Questo assicura la presenza di un "filtro" configurabile per l'accesso al cluster.

II.2 La gestione degli esperimenti

Poiché lo scopo di Neptune è fornire un ambiente multiutente per l'emulazione di rete, come esposto in precedenza, è necessario confrontarsi con le problematiche legate alla definizione, assegnazione e gestione delle risorse di tale ambiente.

Il requisito che si vuole soddisfare in questo caso è che ciascun utente possa richiedere ed ottenere le risorse necessarie al suo esperimento (se disponibili) senza che eventuali altri esperimenti in esecuzione ne siano influenzati o viceversa. In altre parole, si vogliono realizzare quelle caratteristiche di isolamento, molteplicità e scalabilità identificate per il *node multiplexing*, applicate però al contesto più ampio di esperimento in esecuzione sul *cluster*.

L'approccio adoperato da Neptune riprende il concetto di *Virtual Workspace*, diffuso nei sistemi di *Grid computing*. Un *virtual workspace* non e' altro che un ambiente di

esecuzione astratto che può essere reso disponibile dinamicamente per gli utenti autorizzati [18]. L'approccio con *virtual workspace* risolve i problemi di assegnazione delle risorse elaborative agli utenti, permettendo una gestione estremamente flessibile delle risorse del cluster. Nel contesto di Neptune non esistono propriamente dei *virtual workspace*, ma degli "*emulation experiment*", che, richiesti dagli utenti, vengono opportunamente allocati sfruttando le risorse del cluster.

La creazione di un *emulation experiment* passa dunque attraverso i seguenti passi:

- L'utente formula una richiesta di risorse;
- Il sistema (seguendo opportune *policy*) verifica la disponibilità delle risorse richieste e le assegna all'utente;
- L'utente accede alle risorse realizzando il suo esperimento.

II.3 Utenti e ruoli

In Neptune sono identificabili due categorie di utenti:

- **Amministratore del sistema**: è colui che gestisce gli aspetti di configurazione del sistema e che regolamenta l'accesso alle risorse del cluster, definendo opportune *policy* o intervenendo manualmente. Ad esempio, un amministratore di sistema è responsabile per la concessione delle risorse ad un esperimento;
- **Utente**: è colui che crea e/o adopera gli esperimenti.

L'amministratore di sistema può di fatto eseguire qualsiasi operazione definita in Neptune, operando attraverso l'interfaccia esposta dal ***Manager***. Fra le mansioni dell'amministratore di sistema rientrano la definizione e gestione degli utenti, l'autorizzazione all'utilizzo di determinate risorse sul cluster, la gestione globale degli esperimenti.

L'utente, invece, può eseguire le operazioni riguardanti l'esperimento, nel caso ne sia il proprietario (*Experiment owner*) o semplicemente osservarne lo stato in caso contrario (*Experiment user*).

II.4 Il concetto di esperimento

L'*emulation experiment* (a cui ci riferiremo semplicemente col nome di "esperimento") è un concetto fondamentale nella definizione di Neptune. L'esperimento mira infatti a fornire un ambiente separato e gestibile in cui eseguire l'emulazione di rete.

Tramite l'esperimento si gestiscono le risorse in relazione agli utenti del sistema, sia in termini di richiesta che di allocazione. Un esperimento contiene quattro informazioni fondamentali:

- **identificativo**: un nome univoco che permetta di identificare con certezza l'esperimento;
- **proprietario**: colui che ha la responsabilità di gestire l'esperimento e che, di fatto, è chi ha l'intenzione di realizzarlo;
- **risorse**: le risorse del cluster richieste o possedute dall'esperimento;
- **stato**: lo stato dell'esperimento è un insieme di informazioni che indica la "storia" dell'esperimento.

Adoperando questo approccio, l'esperimento diviene un contenitore concettuale di un sotto-insieme delle risorse del cluster, che punta alla realizzazione di un solo obiettivo di emulazione. Un utente, coerentemente con questa definizione, può essere proprietario di uno o più esperimenti.

Poiché si opera in un ambiente multiutente, la presenza dell'indicazione sul proprietario dell'esperimento è necessaria in modo che l'esperimento stesso sia riconducibile ad un utente, responsabile della gestione dell'esperimento, che risulta l'effettivo utilizzatore delle risorse.

Per quel che riguarda le risorse del cluster collegate ad un esperimento, osserviamo che la definizione delle risorse richieste da un esperimento è strettamente legata all'attività di emulazione che l'esperimento deve compiere. Ad esempio, l'emulazione di un sistema composto da due *host* richiede certamente meno risorse dell'emulazione di un sistema composto da decine di *host*. La richiesta in termini di risorse è quindi derivante direttamente dal sistema che l'esperimento vuole emulare, per cui, non e' altro che la definizione di tale sistema.

Facendo riferimento all'emulazione di rete, i sistemi da emulare sono identificabili da una *topologia*, ossia, da un insieme di nodi (*host*, *router*, ecc.) connessi fra loro da un insieme di *link*. Definendo la topologia da emulare si definisce di fatto la richiesta di risorse.

L'indicazione sulle risorse contenuta nell'esperimento è quindi la definizione della topologia che l'esperimento si propone di emulare. Questo informa il sistema delle risorse di cui l'esperimento necessita, ed allo stesso tempo, poiché ciascun esperimento è associato ad un unico utente proprietario, il sistema conosce esattamente le risorse richieste o adoperate da ciascun utente.

Risulta a questo punto evidente la necessità di un modo per distinguere un esperimento che richiede delle risorse da uno che invece tali risorse le ha acquisite. Sostanzialmente, è necessario definire il ciclo di vita di un esperimento e gli stati che l'esperimento assume durante tale ciclo.

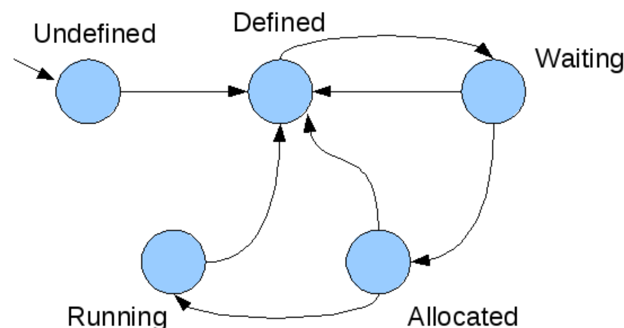


Figura II.2: ciclo di vita dell'esperimento

In Figura II.2 è riportato un diagramma a stati che rappresenta il ciclo di vita di un esperimento. Gli stati individuati sono cinque:

- **Undefined**: l'esperimento è stato creato e ad esso sono stati associati un nome identificativo ed un amministratore, ma non le informazioni sulle risorse richieste;
- **Defined**: l'esperimento è stato definito in tutte le sue parti, ossia, contiene anche le informazioni sulle risorse richieste. Le informazioni sulle risorse, come detto in precedenza, non sono altro che la topologia che si intende realizzare. Quando si è in questo stato la topologia associata è una topologia pronta ad essere realizzata sul cluster, in altre parole, è definita in ogni suo punto;

- **Waiting**: l'esperimento è in attesa di allocazione sul cluster. L'attesa è un momento necessario nella vita dell'esperimento poiché le risorse del cluster sono finite, ed è quindi inevitabile gestire un meccanismo di allocazione delle stesse. In particolare, è lecito supporre che siano due i motivi principali di attesa di un esperimento:
  - L'esperimento attende la liberazione di sufficienti risorse sul cluster per essere allocato;
  - L'esperimento attende l'autorizzazione ad occupare le risorse sul cluster;
- **Allocated**: l'esperimento ha acquisito tutte le risorse necessarie ad essere operativo. Un esperimento in questo stato sta materialmente occupando le risorse sul cluster. Tuttavia, non è ancora un esperimento operativo, infatti, l'esperimento oltre ad ottenere le risorse, per poter realizzare l'emulazione desiderata, deve realizzare la configurazione della topologia. In altre parole, l'aver ottenuto le risorse dal cluster non corrisponde ad aver realizzato la topologia emulata, che invece deve essere opportunamente configurata adoperando tali risorse;
- **Running**: l'esperimento è pronto per l'utilizzo. L'ambiente di emulazione è totalmente configurato e funzionante. Tutte le risorse che erano stato ottenute a seguito della transizione nello stato *allocated* sono ora anche configurate per supportare la topologia dell'esperimento.

Dal diagramma a stati si evidenziano, oltre agli stati appena esposti, anche le possibili transizioni fra questi. Le transizioni sono tipicamente il frutto degli interventi degli utenti del sistema, esaminiamo quando avvengono:

- **undefined → defined**: viene definita una topologia associata all'esperimento;
- **defined → waiting**: il proprietario dell'esperimento richiede l'allocazione delle risorse dell'esperimento;
- **waiting → allocated**: l'esperimento ottiene le risorse richieste;
- **allocated → running**: la topologia associata all'esperimento viene configurata adoperando le risorse ottenute dall'esperimento;

- **running → defined, allocated → defined, waiting → defined**: vengono rilasciate le risorse ottenute dall'esperimento, o annullata la richiesta di allocazione delle risorse precedentemente fatta.

II.5 Il concetto di topologia

Una *topologia* è un insieme di nodi e di link che connettono tali nodi. Sia i nodi che i link hanno identificativi univoci all'interno di una stessa topologia**.** Ad ogni topologia è inoltre associato un identificativo univoco a livello globale che la identifica con certezza.
I nodi, dati gli scopi di Neptune, sono rappresentati come entità con caratteristiche generali del tutto analoghe a quelle di un *computer* sia dal punto di vista *hardware* (memoria centrale, *cpu, hard disk,* ecc.) che *software.*
Le topologie sono accomunabili, a livello logico, ad un grafo, in cui però vanno tenute in considerazione le seguenti informazioni:

- I *nodi* portano informazioni sulla loro configurazione *hardware* e *software*;
- I *link* hanno associate informazioni riguardanti le proprietà che li caratterizzano.

Va infine tenuto in conto che l'analogia con un grafo termina nel momento in cui si prendono in considerazione oltre ai *link* punto-punto anche delle LAN per connettere i nodi fra loro. Quest'ultima caratteristica, tuttavia, verrà ignorata in questo contesto per semplificare la definizione del concetto di topologia.
Gli elementi costitutivi della topologia (nodi, link) sono quindi a loro volta strutture dati complesse, che vanno a realizzare in questo modo un "albero descrittivo della topologia".
Il metodo adoperato per descrivere la topologia sarà riportato successivamente in questo testo, dopo aver trattato ulteriori questioni riguardanti il funzionamento di Neptune.

II.6 Node multiplexing in Neptune

Come accennato in precedenza, Neptune adopera la virtualizzazione come tecnologia chiave per la realizzazione di un ambiente emulato. Volendo essere più precisi, sarebbe corretto affermare che Neptune adopera la virtualizzazione per realizzare il *node*

multiplexing. Nel capitolo I si è discusso di come il *node multiplexing* sia una tecnica per emulare più di un nodo di rete sullo stesso nodo fisico del cluster e di come la virtualizzazione sia adoperata al fine di realizzare il *node multiplexing*. Sempre in quella sede, si è introdotto l'*hypervisor* Xen, un prodotto molto diffuso nell'ambito della virtualizzazione in ambiente GNU/Linux.

Neptune, pur mantenendo un la massima generalità possibile, ha assunto la tecnologia Xen come riferimento per la definizione del suo sistema di *node multiplexing*.

Per garantire un ambiente facilmente configurabile, Neptune ha affrontato il *node multiplexing* immaginando ogni macchina fisica del cluster come un contenitore, al cui interno possono essere posizionate le macchine virtuali. Tali macchine vengono conservate in un *repository*, da cui, quando necessario, sono prelevate per essere distribuite negli opportuni "contenitori". In Figura II.3 è presentato il meccanismo adoperato.

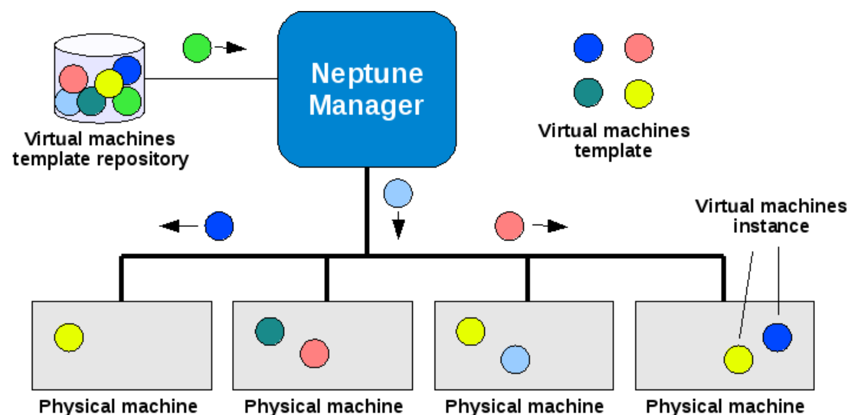


Figura II.3: modello di node multiplexing, basato su template, adoperato in Neptune

In realtà, nel *repository* non sono effettivamente conservate le macchine virtuali che poi sono distribuite sul cluster, ma dei *template* di macchina virtuale, ossia dei modelli che rappresentano una macchina virtuale, che sono poi istanziati, quando necessario, creando una nuova macchina virtuale in tutto e per tutto uguale al *template* [19]. Concettualmente, volendo fare un'analogia con la programmazione orientata agli oggetti, il *template* è accomunabile ad una *classe*, mentre l'istanza della *virtual machine* ad un *oggetto*.

L'approccio basato su *templating* facilita il processo di creazione di una macchina virtuale, che, come abbiamo visto nel capitolo I, richiede altrimenti la creazione da zero di un *file system* con installata un'istanza di un sistema operativo e relativi software.

Perché il sistema sia effettivamente efficace, è necessario definire un insieme di *template* che soddisfi le esigenze degli utenti del sistema. Esempi di *template* possono includere semplici installazioni di sistemi operativi di varie tipologie e versioni, particolari configurazioni comprendenti DBMS server, Web Server, oppure ancora, *template* che realizzano nodi con funzionalità specifiche, come ad esempio un *template* che realizza un *router*.

L'utilizzo del *repository* è vantaggioso anche per la flessibilità che fornisce, nonostante si adoperi uno strumento come quello del *templating* che solitamente risulta invece limitante. Infatti, il *repository* può dinamicamente essere esteso o integrato con ulteriori *template*, inoltre, nel caso gli amministratori del sistema lo ritenessero necessario, garantisce anche un modo per limitare la tipologia di macchine virtuali in esecuzione sul cluster.

II.6.1 Fattibilità dell'approccio basato su template

L'attuazione del *templating* nella realizzazione delle macchine virtuali deve confrontarsi con la risoluzione di alcuni problemi di carattere tecnico, derivanti dalla necessità di configurare ciascun *template* in modo che rispecchi il nodo di una topologia virtuale. Abbiamo precedentemente visto come un nodo sia definito in base alle caratteristiche *hardware* oltre che in base a quelle *software*.

Perché quindi un nodo di una topologia sia correttamente realizzato è necessario definire una macchina virtuale che rispecchi la configurazione *hardware* e *software* del nodo. Per la configurazione *software* il problema viene risolto adoperando un file immagine precostruito (tipicamente una copia del file immagine che rappresenta il *template*). Per la configurazione *hardware* vengono invece adoperate le caratteristiche degli *hypervisor*, che consentono di definire nel dettaglio l'*hardware* di una macchina virtuale.

I *template* definiti nel *repository* sono dunque adoperati soltanto per specificare la configurazione *software* di un nodo. Infatti, è proprio quest'ultima che richiede il maggior impegno in fase di definizione di una macchina virtuale. Inoltre, è possibile rendere la configurazione software pressoché indipendente dalla configurazione *hardware*, il che consente di realizzare i *template* definendo semplicemente le immagini delle macchine virtuali, mentre la configurazione *hardware* viene specificata di volta in volta in base alla macchina virtuale richiesta dalla topologia, sfruttando le funzioni di definizione dell'hardware messe a disposizione dell'*hypervisor*.

Quest'ultima caratteristica risulta di importanza vitale per il funzionamento del modello di *node multiplexing* di Neptune, poiché nodi che adoperano la stessa configurazione software possono avere una configurazione hardware molto differente fra loro, quanto meno nel numero e nel tipo delle interfacce di rete, che come vedremo più avanti, sono elementi fondanti per supportare il *link multiplexing*.

II.7 Link multiplexing in Neptune

Abbiamo visto che una topologia è caratterizzata da un insieme di nodi ed un insieme di link nella sua definizione più generale. Attenendoci a tale definizione, il problema della realizzazione della rete virtuale consiste nella realizzazione di un certo numero di link punto-punto fra determinati nodi.

I nodi in questione sono macchine virtuali, gestite da *hypervisor*, anche di diverse tecnologie, che risiedono su *computer* organizzati in un *cluster*, in cui la connettività è garantita da una rete LAN ad alta velocità condivisa fra tutte le macchine fisiche.

Su tale rete devono essere simulati i link fra i nodi, dove per simulazione si intende:

- Realizzazione della connessione bidirezionale con caratteristiche indipendenti dalle altre connessioni presenti sulla LAN (*link multiplexing*);
- Simulazione dei parametri caratteristici del link (ad esempio: bandwidth, delay, loss rate, ecc.).

Le tecniche di *link multiplexing* ideate per il funzionamento su tale struttura sono sostanzialmente due [11]:

- *IP aliasing*: prevede che la definizione del link avvenga interamente al livello IP. Sulla stessa interfaccia vengono gestiti più indirizzi IP, ciascuno rappresentante di un *end-point* di un link. L'indipendenza del link dagli altri link presenti sull'interfaccia e' garantita da sistemi di *traffic shaping* che regolano l'occupazione di risorse del link ad una frazione delle risorse totali messe a disposizione dal link stesso;
- *One Link Per Interface*: ogni *end-point* di un link richiede un'interfaccia dedicata. Ogni link, quindi, è modellato come una connessione diretta fra due interfacce di rete. Questo approccio è possibile poiché si opera con nodi realizzati con *virtual machine*, è quindi possibile installare facilmente nuove interfacce di rete virtuali per supportare i link. Anche se questa tecnica di *multiplexing* prevede di intervenire al livello *datalink*, in realtà la realizzazione del link viene completata ancora una volta al livello IP. Infatti, perché il link risulti correttamente utilizzato, è necessario intervenire sulle tabelle di *forwarding*, in modo da inoltrare tutto il traffico riguardante il link sull'interfaccia in cui questo è realizzato.

La seguente figura esemplifica il funzionamento delle due tecniche:

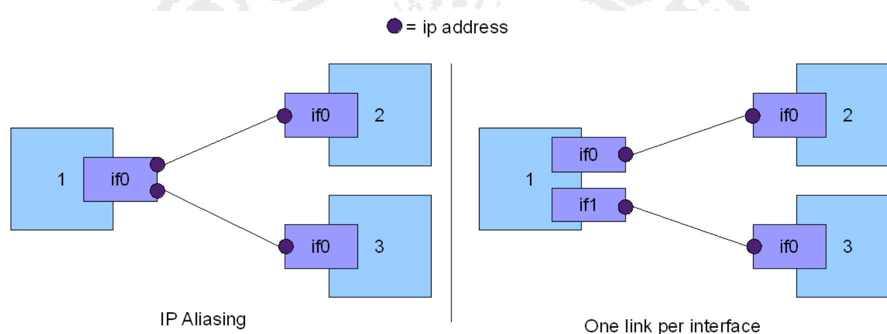


Figura II.4: Tecniche di link multiplexing in Neptune

Allo stato attuale Neptune adopera in prevalenza la seconda tecnica, che risulta di più semplice applicazione dal punto di vista tecnico, in un ambiente composto da macchine virtuali. L'IP *aliasing* richiederebbe infatti la definizione di complesse politiche di *traffic shaping*, oltre a rendere decisamente più difficoltosa l'identificazione di un link rispetto a quanto avviene con OLPI (*One Link Per Inferface*), che richiede la conoscenza soltanto delle interfacce che vengono unite dal link, perché quest'ultimo sia univocamente individuato.

Il supporto all'IP *Aliasing* viene comunque contemplato, ma se ne rimanda la realizzazione a fasi successive.

II.7.1 Virtual machine networking

Affinché sia possibile definire con precisione come realizzare la tecnica OLPI per la realizzazione di un *link* virtuale, è necessario comprendere come i nodi virtuali siano gestiti dall'*hypervisor* dal punto di vista del collegamento alla rete. Si è già detto che le macchine fisiche componenti il cluster sono connesse con una rete LAN ad alta velocità, esaminiamo dunque se e come i nodi virtuali, ospitati su tali macchine fisiche, sono visti da tale rete.

L'assunzione come modello di riferimento dell'*hypervisor* XEN [20], anticipata in precedenza, è adoperata anche in questo caso.

Anticipiamo in questo paragrafo, lasciando l'approfondimento sui dettagli di funzionamento a quello successivo, che ciascun nodo virtuale risulta, dal punto di vista della rete LAN, come una comune macchina connessa a tale rete attraverso uno *switch*. In altre parole, la rete non distingue un nodo fisico da un nodo virtuale ospitato da uno dei nodi fisici.

L'unica differenza nella comunicazione fra nodo fisico e nodo virtuale è che in quest'ultimo caso è presente un *overhead* dovuto alla realizzazione via software dello *switch* che connette il nodo virtuale alla rete.

II.7.1.1 Xen networking

Ricordiamo che il nome delle macchine virtuali in *Xen* e' *dom*. Ogni vm (*virtual machine*) e' quindi individuata da uno specifico nome *domU*, dove "U" rappresenta un numero progressivo che viene assegnato in base all'ordine di creazione delle macchine stesse (es. dom1, dom2, ecc.).

La vm dom0 e' una speciale macchina, che si occupa della gestione dell'intera infrastruttura *Xen*. In seno a questa macchina e' gestito anche il *networking* per le macchine virtuali.

Quando viene creata *dom0*, l'interfaccia reale del computer (eth0) viene rinominata in *peth0* e collegata ad uno *switch software* realizzato da Xen, che prende il nome di *xenbr0.*

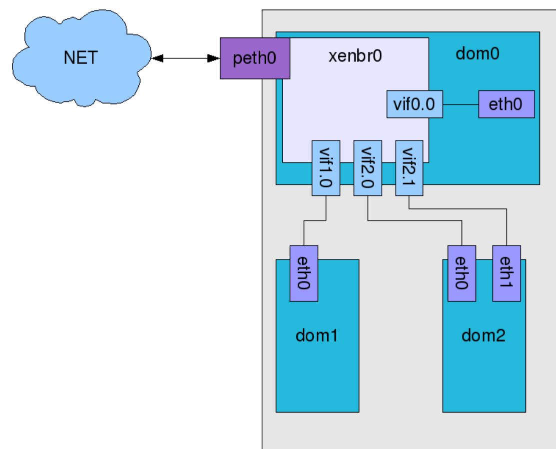


Figura II.5: Xen networking in modalità bridge

Lo *switch xenbr0* ha tante "porte" (sono in realta' delle interfacce virtuali) quante sono le interfacce virtuali dei vari *domU* che devono essere collegate. Ognuna delle interfacce di *xenbr0* prende il nome di *vifX.Y*, dove "X" corrisponde al numero di *dom* (*domX*) ed "Y" e' il numero dell'interfaccia interna a *domX* collegata a *vifX.Y*.

Quando un pacchetto arriva dalla rete all'interfaccia *peth0*, questo viene gestito innanzitutto da *xenbr0*. Poiché *xenbr0* non e' altro che un componente di *dom0*, il pacchetto viene gestito, a livello *data link*, tramite i meccanismi del *kernel* di *dom0* (in particolare ad *ebtables*, di cui si discuterà in seguito). Dopo aver passato questo stadio, *xenbr0* distribuisce il pacchetto in base al suo indirizzo MAC destinazione, consegnandolo

all'opportuna interfaccia di uno dei *domU* collegati. Giunto al *domU*, il pacchetto viene gestito nel modo convenzionale dal sistema operativo della vm.

A quanto detto si deve in realtà aggiungere qualcosa riguardo la gestione dei pacchetti in *dom0*. Se infatti in *dom0* si fa uso di *iptables* (il tool di gestione delle funzionalità *netfilter,* anch'esso trattato più avanti), si potrebbero incorrere in problemi, o meglio, i *domU* vedrebbero il loro traffico filtrato dalle impostazioni *iptables* di *dom0*. Infatti, i pacchetti che passano per *xenbr0* sono comunque affetti dalle *chains iptables* PREROUTING, FORWARD, POSTROUTING di *dom0*.

Per maggiori e piu' dettagliate informazioni si rimanda alla documentazione di Xen [riferimento bibliografia da inserire].

II.7.2 Tecnologie per la simulazione dei link

Abbiamo fin qui esaminato come i nodi virtuali sono inseriti in un contesto reale di rete. Rimane da stabilire come si possano realizzare le caratteristiche presenti nella definizione di un link virtuale, ossia come sia possibile realizzare un link punto-punto fra due arbitrari nodi, che presenti delle caratteristiche in termini di qualità del collegamento (delay, jitter, loss rate, ecc.) definite anch'esse arbitrariamente.

Nel seguito, sono descritte alcune funzionalità disponibili per *kernel linux*, che risultano utili a tale scopo.

II.7.2.1 Ebtables

Ebtables è un *tool* inserito fra i servizi standard del *kernel linux* a partire dalla versione 2.6. Il *tool* realizza un filtraggio dei dati in ingresso e uscita verso le interfacce di rete, frapponendosi fra l'elaborazione *ethernet* fatta dall'interfaccia di rete e i successivi livelli dello *stack* protocollare.

Il *tool* opera su *bridge software* di tipo *ethernet*, perché sia adoperabile è quindi necessario realizzare innanzitutto un *bridge software*, su cui è poi possibile definire, fra le altre, le seguenti operazioni [21]:

- Ethernet protocol filtering;

- MAC address filtering;
- Simple IP header filtering;
- ARP header filtering;
- 802.1Q VLAN filtering;
- In/Out interface filtering (logical and physical device);
- MAC address nat;
- Logging;
- Frame counters;
- Possibilità di aggiungere, cancellare ed inserire regole, flush chains, zero counters;
- Supporto per chains definite dall'utente;
- Supporto per il marking dei frames e per il “matching marked frames”.

Da quanto riportato, risulta evidente come nel *networking* attuato da XEN, questo *tool* possa essere agevolmente adoperato nell'impostare le politiche riguardanti il *bridge software* realizzato per la connessione in rete dei nodi virtuali.

II.7.2.2 Iptables

Netfilter [22] è un componente del *kernel* del sistema operativo GNU/Linux, che permette l'intercettazione e manipolazione dei pacchetti che attraversano il calcolatore. *Netfilter* permette di realizzare alcune funzionalità di rete avanzate come la realizzazione di un *firewall* basato sul filtraggio *stateful* dei pacchetti o configurazioni anche complesse di NAT, o ancora la manipolazione dei pacchetti in transito.

Iptables è il programma che permette agli amministratori di sistema di configurare *netfilter*, definendo le regole per i filtri di rete e il re-indirizzamento NAT. Spesso con il termine *iptables* ci si riferisce all'intera infrastruttura, incluso *netfilter*.

Iptables è un componente standard di tutte le moderne distribuzioni di GNU/Linux.

II.7.2.3 Traffic Control/Netem

Le utilità *traffic control (Tc)* [23] sono un ulteriore servizio messo a disposizione dal *kernel* Linux per la gestione del traffico di rete. *Tc* consente di definire delle politiche di schedulazione in uscita ed in ingresso ad ogni interfaccia di rete riconosciuta dal sistema.

Più precisamente, è possibile organizzare le politiche di schedulazione in una struttura ad albero [24], che consente di attuare complessi meccanismi di controllo. Quale ramo dell'albero è preso da un particolare pacchetto del flusso dati di rete è definito in base ad opportuni filtri, che possono basarsi su praticamente ogni campo degli *header* dei protocolli di rete.

A seconda di come viene percorso l'albero si possono applicare *queuing discipline* (discipline di accodamento) differenti, fra queste sono disponibili: *FIFO queuing, Token Bucket Filter, priority queuing, ecc.*

Netem [25] è un *tool* che estende le funzionalità di *Tc*, fornendo la possibilità di simulare i comportamenti di un link reale, introducendo ritardi, perdite, *jitter*, duplicazione nella definizione di una *queuing discipline*.

Sostanzialmente, il funzionamento di questi *tool* prevede che un'applicazione affidi al sistema operativo un suo pacchetto dati, che viene poi processato da un'opportuna *queuing discipline,* scelta in base a quanto definito con i filtri *Tc*. Una *queuing discipline* potrebbe forzare un flusso dati, ad esempio, ad avere una massima velocità di 50 Mbits in uscita, con un ritardo variabile fra i 50 e i 100 ms.

II.7.3 Realizzazione della tecnica OLPI

Alcune delle tecnologie precedentemente presentate sono state adoperate per l'effettiva realizzazione della tecnica OLPI. Ricordiamo che tale tecnica prevede che ogni link virtuale sia una connessione diretta fra due interfacce di rete. Nella nomenclatura adoperata, ciascuna interfaccia viene definita *end-point*, quindi, ciascun link ha esattamente due *end-point*.

Ogni *end-point* viene facilmente identificato tramite il MAC *address* dell'interfaccia che lo realizza, o attraverso la coppia di valori nome dell'interfaccia, nome del nodo cui appartiene l'interfaccia. Quest'ultima possibilità garantisce l'indipendenza di una definizione logica dell'*end-point*, dall'effettiva realizzazione fisica, che richiede la

conoscenza di dettagli, come un MAC address, che potrebbero essere non noti al momento della definizione logica.

Sfruttando la corrispondenza fra interfaccia e link, la tecnica si articola in due fasi:

1. Configurazione dei parametri di qualità del link;
2. Configurazione dei sistemi operativi per creare la connessione fra i due *end-point*.

Al termine del processo descritto si ottiene un link fra i due *end-point*, che collega esclusivamente i due *end-point* e che presenta le caratteristiche di qualità definite dall'utente.

II.7.3.1 Link parameters enforcement

La definizione dei parametri di qualità di un link avviene adoperando le funzionalità messe a disposizione da *Traffic Control/Netem*. Per ogni interfaccia che invia dati (entrambe se il link è di tipo *full-duplex*) viene definita una *queuing discipline HTB* (si prevede la possibilità di selezionare il tipo di *queuing discipline* nel breve termine) per impostare i valori di *bandwidth*, mentre le altre caratteristiche (*delay*, *loss rate*, *duplication*, *reordering*) sono impostate attraverso la definizione degli omonimi parametri tramite *Netem*.

II.7.3.2 Link creation

L'effettiva creazione del link avviene configurando la tabella di *fowarding* del sistema operativo. Perché la configurazione del link avvenga correttamente è dunque necessario che alle interfacce che sono *end-point* di un link siano associati gli indirizzi IP e che questi appartengano alla stessa *subnet*.

Posto che il link “link1” abbia come *end-point* le interfacce A e B e che a queste siano rispettivamente associate gli indirizzi IP *IpA* e *IpB*, le tabelle di *fowarding* dei sistemi operativi OS-A ed OS-B, che gestiscono le omonime interfacce, saranno:

OS-A		
Destination	**Gateway**	**Interface**
IpB	**-**	***A***

OS-B		
Destination	**Gateway**	**Interface**
IpA	**-**	***B***

II.8 La rete di controllo

In questo paragrafo puntualizziamo il ruolo della rete di controllo, che è stato solo accennato fino a questo punto. I nodi realizzati da Neptune per supportare gli esperimenti sono macchine virtuali, residenti su macchine fisiche del cluster. E' chiaramente necessario un modo per accedere a queste macchine virtuali al fine di controllarle e monitorarle. Abbiamo visto come in un esperimento, le macchine virtuali (i nodi) vadano a realizzare una particolare topologia di rete, che risulta isolata dal mondo esterno. Per accedere a queste macchine è dunque necessario un ulteriore *media* costituito appunto dalla rete di controllo. Questa rete ha *range* di indirizzi IP ben definiti, in modo che non possano essere confusi con indirizzi appartenenti a *link* di topologie degli esperimenti, e mette in comunicazione tutte le macchine virtuali con il *neptune manager*.

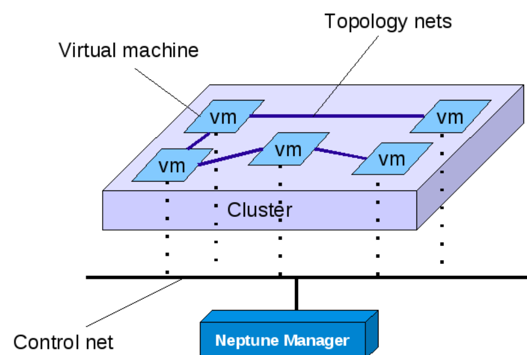


Figura II.6: La rete di controllo

La rete di controllo è un punto molto delicato dell'infrastruttura di Neptune, poiché, per quanto dovrebbe risultare trasparente per l'esperimento di emulazione, a conti fatti non lo è, in quanto ciascuna macchina virtuale è a conoscenza della sua esistenza.

II.9 Architettura software

Nei precedenti paragrafi si è introdotta l'organizzazione di Neptune dal punto di vista del cluster, con la definizione del *Neptune Manager*. In tale sede non si è fatto riferimento alla configurazione software che deve essere realizzata sul cluster perché Neptune possa funzionare.

Per sua natura Neptune nasce per poter funzionare su di un *hardware general purpose*. Questa sua caratteristica ne garantisce una forte flessibilità. Fino ad ora, si è dato per scontato che Neptune adoperasse un intero cluster per il suo funzionamento, in realtà, nulla vieta di adoperare solo porzioni del cluster, scelte anche in modo arbitrario. Arrivando a casi estremi, Neptune potrebbe funzionare anche su di un singolo *personal computer*, per quanto questo non avrebbe quasi alcuna utilità dal punto di vista degli esperimenti di emulazione.

Le macchine fisiche del cluster che si intende adoperare per Neptune devono però avere un'opportuna configurazione software, in modo che possano gestire le funzionalità di virtualizzazione richieste dal sistema. Ogni macchina deve quindi avere installato un *hypervisor* supportato da Neptune (allo stato attuale solo Xen è supportato). Il Neptune Manager ospita invece il software di gestione dell'intero sistema, che prende il nome di *Neptune Infrastructure Manager* (NIM).

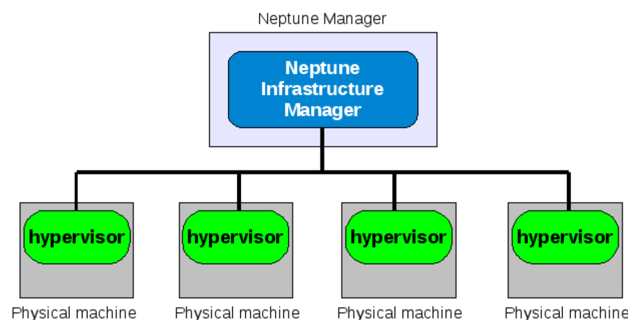


Figura II.7: architettura software di Neptune

Il NIM è un sistema software complesso che realizza tutti i meccanismi di Neptune, dalla gestione degli utenti a quella degli esperimenti, dall'allocazione delle risorse al monitoraggio di *virtual machine* e nodi fisici del cluster.

Il NIM, per dominare la complessità delle operazioni da compiere, è realizzato seguendo un approccio a livelli [19]. Concettualmente sono identificabili tre livelli:

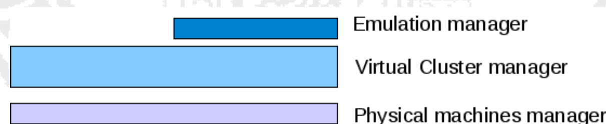


Figura II.8: l'architettura a livelli del NIM

- ***Physical machines manager (pmm)***: è il livello delle macchine fisiche. Si occupa dell'installazione e gestione delle macchine fisiche a livello di sistema operativo;
- ***Virtual cluster manager (vcm)***: comunicando con gli *hypervisor*, realizza i cluster virtuali che ospiteranno poi le topologie degli esperimenti. In questo livello sono realizzati i meccanismi di gestione e monitoraggio di una macchina virtuale. Questo livello ha coscienza del livello *pmm*, ossia, conosce il numero e la tipologie delle macchine fisiche;
- ***Emulation manager (em)***: gestisce gli esperimenti e le topologie, introducendo al contempo le caratteristiche di gestione della multi-utenza.

E' tuttavia necessario precisare che, per quanto concettualmente l'approccio a livelli sia presentato come la soluzione ottimale, nella realizzazione pratica è difficile mantenere separati i livelli *vcm* ed *em* senza rallentare o complicare eccessivamente l'implementazione del sistema. Questi livelli sono, infatti, fra loro estremamente legati ed uno sviluppo separato potrebbe comportare eccessive difficoltà di integrazione.

D'altro canto, l'approccio a livelli permette l'apertura ad un più vasto raggio di applicazioni per il progetto Neptune. Se infatti consideriamo i soli livelli *pmm* e *vcm*, è possibile realizzare un sistema per la gestione di *virtual cluster*, non più orientato all'emulazione di rete, ma che ha scopi decisamente più *general purpose*.

II.10 Problematiche di gestione delle reti degli esperimenti

Fino ad ora, parlando dei problemi di gestione delle risorse, si è sempre fatto implicitamente riferimento a risorse di tipo computazionale o di memoria. Allocare un esperimento di emulazione di rete consuma però ulteriori risorse, che sono appunto le risorse di rete. Tralasciando le problematiche di gestione della capacità della rete (e quindi la necessità di condividere un unico canale di comunicazione fra diversi link emulati), in questo frangente ci occuperemo delle problematiche legate alla gestione degli indirizzi IP. Per semplicità, tratteremo il problema considerando soltanto indirizzi Ipv4.

Dalla descrizione delle tecniche di *link multiplexing* abbiamo visto come un *link* è realizzato tramite un collegamento punto-punto fra due interfacce di rete, cui sono assegnati indirizzi IP della stessa sotto-rete. Questo implica che in un esperimento con *n link* siano definite almeno *n* sotto-reti, il che corrisponde ad assegnare 2*n* indirizzi IP. Ricordiamo, inoltre, la necessità di far convivere link appartenenti a topologie differenti sulle stesse risorse fisiche (la rete del cluster), senza che interferiscano fra loro.

Risulta chiaro a questo punto che potrebbero verificarsi problematiche legate ad una interazione di indirizzi IP appartenenti ad esperimenti differenti: un effetto indesiderato che potrebbe verificarsi ,nel caso in cui la separazione non fosse fatta correttamente, è lo scambio di dati fra nodi appartenenti a topologie differenti, ad opera del fatto che sono stati assegnati ad esperimenti differenti indirizzi IP residenti su di una comune sotto-rete.

Questi inconvenienti possono avvenire IP qualora non vengano rispettati opportuni vincoli (stabiliti via software o per convenzione). Per quanto un approccio via software sarebbe la strada più opportuna, essendo del tutto trasparente all'utente del sistema, la sua realizzazione è macchinosa e rischia di mettere a rischio la bontà dei risultati sperimentali, poiché si richiedono eccessivi strati software aggiuntivi per il solo funzionamento del sistema.

La soluzione alternativa, è l'imposizione di particolari range di indirizzi assegnabili da ciascun esperimento. Questa soluzione è praticabile poiché verosimilmente il numero di esperimenti contemporanei è nell'ordine delle decine.

Seguendo un approccio più rigoroso, possiamo definire una suddivisione gerarchica delle sotto-reti di Neptune, gestita tramite la configurazione delle *subnet mask.* :

- *neptune subnet:* è la sotto-rete generale che contiene tutte le altre. Si presuppone che questa sottorete sia definita come appartenente ad una delle sotto-reti private, come ad esempio la 10.0.0.0/8 (in realtà nulla vieterebbe l'utilizzo di una sotto-rete pubblica se è possibile farlo);

- *experiment subnet:* è una sotto-rete della *neptune subnet* assegnata ad un solo esperimento. Tutte le sotto-reti definite nell'ambito di questo esperimento devono essere sotto-reti della rispettiva *experiment subnet*;
- *link subnet:* è la sotto-rete di un link. Ciascuna *link subnet* è una sotto-rete della *experiment subnet* dell'esperimento cui appartiene.

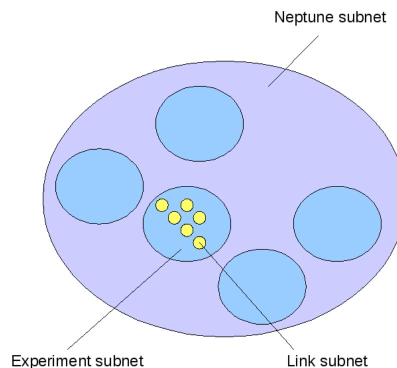


Figura II.9: Struttura delle sotto-reti

A titolo esemplificativo, si potrebbe avere una configurazione di questo tipo:

- *neptune subnet:* 10.0.0.0/8;
- *experiment subnet:* tutte le sotto reti fra 10.0.0.0/16 e 10.255.0.0/16. Queste sotto-reti sono assegnate di volta in volta ad un esperimento. Ad esempio, il primo esperimento creato avrà la sotto-rete 10.0.0.0/16, il secondo la sotto-rete 10.1.0.0/16 e così via. Quando un esperimento cessa la sua attività, la corrispondente sotto-rete ritorna libera per essere assegnata ad un nuovo esperimento;
- *link subnet:* tutte le sotto reti fra 10.0.0.0/24 e 10.255.255.0/24. Ad esempio, se abbiamo l'esperimento con sotto-rete 10.12.0.0/16, i link definiti in seno a questo esperimento avranno una *link subnet* compresa fra le subnet 10.12.0.0/24 e 10.12.255.0/24;

Neptune permette di configurare totalmente la gestione di queste *subnet*. Questo approccio consente di definire in modo personalizzato il numero massimo di esperimenti contemporanei che possono essere eseguiti (limitati dal numero massimo di *experiment*

subnet) e il numero massimo di *link* che ciascuno di questi esperimenti può avere (limitato dal numero massimo di *link subnet*). E' chiaro che queste due dimensioni sono fra loro in contrasto: per avere più esperimenti è necessario che questi siano “più piccoli”, ossia che abbiano meno link e viceversa.

II.11 Il modello dei dati

A questo punto dovrebbero essere note gran parte delle questioni riguardanti Neptune, è quindi possibile introdurre il modello dei dati a supporto del sistema, che tiene conto di tutti gli aspetti fin qui trattati.

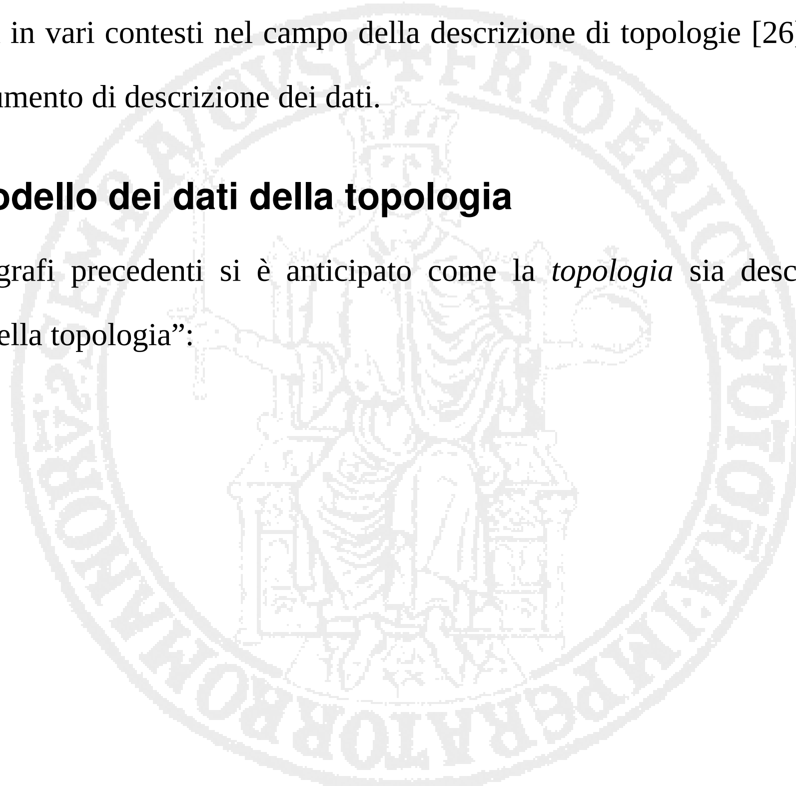
I dati di interesse, che devono essere definiti opportunamente, riguardano quattro ambiti:

- *topologia;*
- *esperimento;*
- *virtual machine template;*
- *cluster.*

Anticipiamo fin da subito che il modello dei dati si adatta perfettamente ad una descrizione strutturata, che sia agevolmente gestibile da un sistema informatico, ma che al contempo sia umanamente leggibile. Per queste ragioni, il linguaggio XML, adoperato fra l'altro già in vari contesti nel campo della descrizione di topologie [26], è stato adoperato come strumento di descrizione dei dati.

II.11.1 Modello dei dati della topologia

Nei paragrafi precedenti si è anticipato come la *topologia* sia descrivibile tramite un “albero della topologia”:



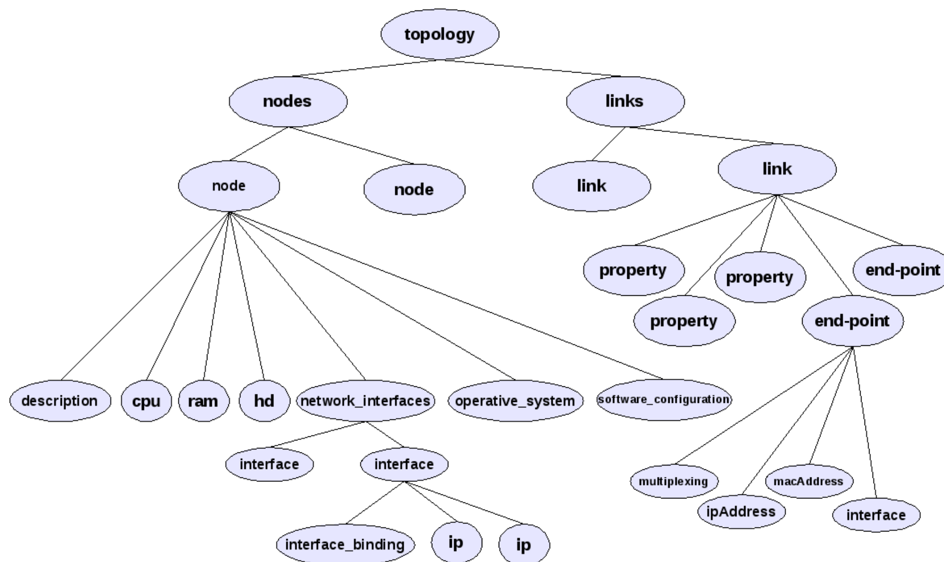


Figura II.10: Albero descrittivo della topologia

- ***topology***: elemento radice, attributi ne specificano un identificativo univoco ed il nome dell'autore;
  - ***nodes***: contenitore per gli elementi *node*. Gli elementi contenuti sono sia nodi di cui non è definito alcun genitore, che nodi che sono definiti come figli di altri nodi della topologia. Per individuare se un nodo è il figlio di un altro nodo, si esplorano tutti i nodi definiti nella topologia, ricercando se fra questi quale dichiara il nodo in questione come "hosted_VM". Il sistema garantisce l'assenza di situazioni anomale;
    - ***node***: riporta l'id di un nodo partecipante alla topologia;
      - ***description*** *(opt.)*: per scopi documentali, ha contenuto testuale leggibile;
      - ***cpu***: descrive il nome, il tipo ed il clock della cpu;
      - ***ram***: descrive la quantita' di ram della macchina;
      - ***hd***: lo spazio di storage della macchina;
      - ***network_interfaces***: contenitore per le interfacce di rete;

- ***interface***: descrive un'interfaccia di rete, specificando il nome della stessa come riconosciuto dal sistema operativo e l'indirizzo MAC. Un attributo specifica il tipo di interfaccia;
  - ***interface_binding*** *(opt.)*: descrive come è collegata l'interfaccia di rete virtuale all'interfaccia di rete della macchina che ospita la macchina virtuale. La descrizione del binding dipende dal metodo di virtualizzazione adoperato;
  - ***ip*** *(opt.)*: rappresenta un eventuale indirizzo IP associato all'interfaccia. L'attributo *type* può essere impostato a *control* o *topology* per indicare se l'indirizzo rappresenta un punto di accesso per il controllo della macchina o fa parte della topologia di un esperimento;

- ***operative_system***: specifica tipo e versione del sistema operativo;
- ***software_configuration***: è un contenitore per indicare il software installato sul sistema. Un attributo indica un'immagine che rappresenta la configurazione software della macchina;
  - ***software*** *(opt.)*: elemento generico per specificare un qualsiasi tipo di software, mediante la tripla di attributi (name, type, version);
  - ***sshServer***: specifica, tramite un attributo, la porta su cui è in ascolto il server ssh;
- ***hypervisor*** *(opt.)*: specifica l'hypervisor adoperato per la virtualizzazione;
- ***hosted_VMs*** *(opt.)*: contenitore per le macchine virtuali ospitate su questo nodo;
  - ***vm***: contenitore per l'identificazione della macchina virtuale. Specifica, tramite un attributo, un identificativo univoco della macchina, a livello della topologia;

- ***links***: contenitore per gli elementi *link*;

- ***link***: descrive un link diretto fra due nodi. Ogni link ha un identificativo univoco, un tipo (*fullduplex, halfduplex, LAN*) un indirizzo IP che identifica la *subnet* che realizza il *link* se il link è di tipo LAN. Tutte queste informazioni sono specificate come attributi dell'elemento;
  - ***property***: descrive una proprietà di un *link*. Le proprietà sono espresse come coppie (chiave, valore). Le proprietà sono: *bandwidth, delay, jitter, loss, duplication, re-ordering*;
  - ***end_point***: rappresenta uno dei punti terminali del *link*, contiene un attributo *nodeId* per specificare quale nodo è punto terminale del link. Nel caso di *link halfduplex* va specificato l'attributo *type* che può contenere i valori *source* o *destination*;
    - ***multiplexing** (opt.)**:*** rappresenta la tecnica di multiplexing adoperata per realizzare il link;
    - ***macAddress** (opt.):* riporta il mac address di questo end point. Viene specificato se lo richiede la tecnica di multiplexing;
    - ***ipAddress** (opt.):* riporta l'ip addess di questo end point. Viene specificato se lo richiede la tecnica di multiplexing;
    - ***interface**:* riporta l'identificativo dell'interfaccia su cui è attaccato questo link;

Alcune considerazioni possono essere fatte su questo modello:

- L'entità *interface_binding* ha lo scopo di definire quale modalità di *binding l'hypervisor* deve *usare* e quali sono i dati per configurare opportunamente la modalità scelta. Allo stato attuale, questa informazione rappresenta soltanto una indicazione concettuale, poiché non si prevede una realizzazione del supporto a *binding* diversi dall'*host interface binding* (il sistema presentato per Xen in precedenza), nel breve termine;
- Ogni indirizzo IP specificato per un nodo contiene un attributo *type* che serve ad identificare quell'indirizzo come punto di accesso per il controllo del nodo o come

facente parte della topologia virtuale. Questo automaticamente identifica una interfaccia di rete su cui è definito un indirizzo IP di controllo come interfaccia di controllo;

- Nella definizione degli *end_point*, è necessario specificare soltanto l'id del nodo e il nome dell'interfaccia cui l'*end point* fa riferimento. Questo consente di definire dei *link* a livello concettuale, senza dover fissare parametri “fisici” come ad esempio un MAC address;

II.11.2 Modello dei dati dell'esperimento

L'esperimento è stato già trattato in precedenza con un certo livello di dettaglio. In tale sede è stato definito come un insieme di 4 informazioni:

- **identificativo**: un nome univoco che permetta di identificare con certezza l'esperimento;
- **proprietario**: colui che ha la responsabilità di gestire l'esperimento e che, di fatto, è chi ha l'intenzione di realizzarlo;
- **topologia**: la topologia collegata all'esperimento (che rappresenta le risorse);
- **stato**: lo stato dell'esperimento è un insieme di informazioni che indica la “storia” dell'esperimento.

Lo stato non è anch'esso una struttura dati complessa, che contiene l'indicazione sul nome dello stato dell'esperimento (*allocated, running, ecc.)* e una serie di informazioni secondarie, come ad esempio la data di creazione, avvio, ecc.

Oltre a queste informazioni, l'esperimento contiene anche una lista degli utenti che possono accedere alla sua consultazione (gli *experiment user*) e l'indicazione di qual è l'*experiment subnet* ad esso collegata.

II.11.3 Modello dei dati del virtual machine template

Un *template* di macchina virtuale non è altro che una *immagine* che contiene la descrizione software della macchina. Perché il sistema possa trattare in modo opportuno queste immagini, è necessaria una descrizione che le caratterizzi e ne esponga i dettagli.

Tale descrizione non è altro che l'elencazione delle caratteristiche della configurazione software.

Abbiamo visto come nell'albero della topologia siano descritti i nodi e le loro configurazioni software. La descrizione del *virtual machine template*, quindi, non è altro che l'insieme delle entità che descrivono la configurazione software del nodo.

In particolare, tale insieme include le entità: *operative system, software_configuration, software, sshServer*.

II.11.4 Modello dei dati del cluster

Il modello dei dati del cluster serve a rappresentare l'insieme delle macchine fisiche che costituiscono il *cluster* e quindi l'infrastruttura di emulazione.

Il *cluster* non è altro che una collezione di nodi fisici, ciascuno con la sua descrizione. I nodi fisici sono descrivibili allo stesso modo di come sono descritti i nodi virtuali, definiti nel modello dei dati della topologia. Ancora una volta, quindi, la descrizione di questo modello dei dati può far riferimento all'albero della topologia, o meglio, alla descrizione dei nodi presenti nell'albero della topologia.

I nodi fisici variano in questa descrizione soltanto per la presenza aggiuntiva dell'entità *pcu* che sta a rappresentare l'eventuale presenza di una *power control unit* installata nel nodo.

II.12 Validazione e completamento di una topologia

La gestione delle informazioni riguardanti una topologia è un processo che coinvolge diverse operazioni necessarie in primo luogo a stabilire la correttezza della topologia ed, in ottica più generale, a garantire che la topologia stessa possa essere trattata dal sistema.

Il processo di descrizione di una topologia può includere la definizione di molte informazioni, non tutte essenziali ai fini della definizione della topologia stessa. In altre parole, tolto un sottoinsieme di informazioni necessarie, le restanti possono essere ricavate

o assegnate in modo automatico a seconda di quali siano le esigenze del sistema. Fatta questa premessa, definiamo le seguenti operazioni:

- ***Validation***: la validazione è un processo di verifica che varia in base allo scopo prefissato. Il processo consiste nella verifica che la topologia contenga tutte le informazioni necessarie (per lo scopo prefissato) e che tali informazioni siano corrette. A titolo esemplificativo, uno scopo potrebbe essere la verifica che siano fornite tutte le informazioni che caratterizzano a livello "logico" la topologia. (Più dettagli su questo argomento sono forniti nel seguito);
- ***Completion***: il completamento è l'operazione che definisce, per una topologia, le informazioni necessarie affinché questa possa essere allocata sul cluster.

II.12.1 Validazione della topologia

La validazione della topologia avviene in due momenti distinti, con scopi differenti. Per ogni scopo definiamo un tipo diverso di validazione:

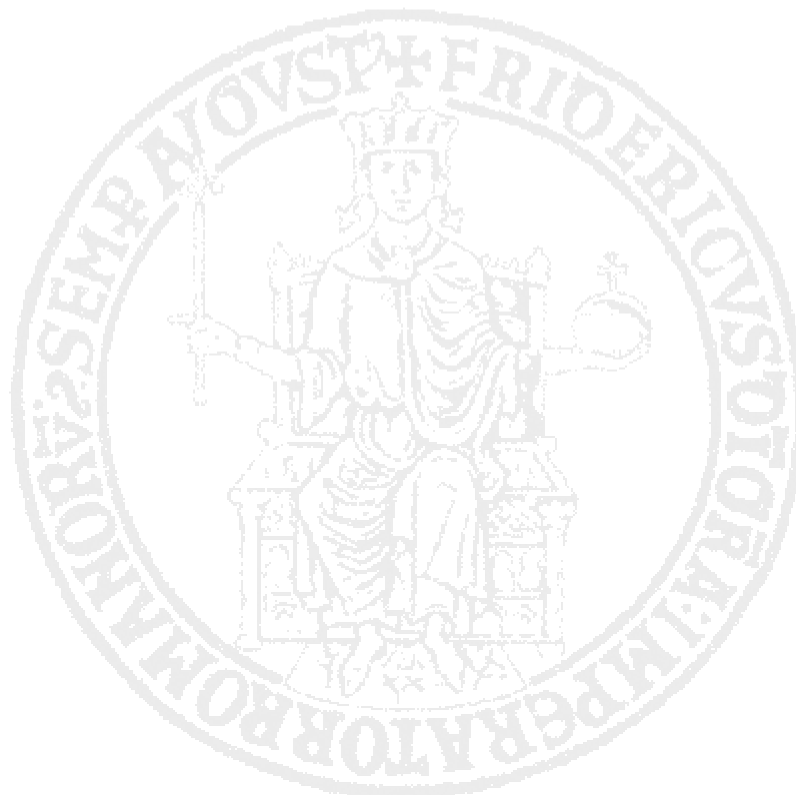
- *Core Validation*: analizza la descrizione della topologia per stabilire se le informazioni contenute sono sufficienti a definire in modo completo la topologia descritta. Le informazioni necessarie sono tutte quelle che il sistema non è in grado di generare in modo autonomo. Definiamo una topologia che passa la *core validation* semplicemente come *valid*;
- *Allocability Validation:* verifica che le informazioni contenute nel descrittore della topologia siano sufficienti a garantire la corretta allocazione della stessa sul cluster. Questa validazione è fatta immediatamente prima di richiedere l'allocazione della topologia, in modo da garantire che il sistema sia in grado di effettuare le sue operazioni in modo opportuno. Definiamo una topologia che passa l' *allocability validation* come *allocable*.

Dalla definizione dei due tipi di validazione si può affermare che la *core validation* è un sottoinsieme della *allocability validation*. In altre parole, se una topologia è *allocable*, allora sarà anche *valid*.

II.12.2 Completamento della topologia

Il completamento della topologia avviene generando in modo automatizzato le informazioni che risultano mancanti rispetto a quelle necessarie ad ottenere un sistema *allocable*.

La generazione delle informazioni non garantisce che le informazioni generate rendano la topologia *allocable*, poiché potrebbero essere state definite delle informazioni errate da parte dell'utente (il completamento non corregge le informazioni errate). In compenso, tutte le informazioni generate dal sistema sono corrette e quindi non richiedono modifiche per ottenere una topologia *allocable*.



Capitolo III
Progettazione del Neptune Infrastructure Manager

Nei precedenti capitoli si è presentato il progetto Neptune e l'infrastruttura *hardware* e *software* che lo caratterizza. Tralasciando il ruolo compiuto dagli *hypervisor*, l'elemento fondamentale del software a supporto di Neptune è il *Neptune Infrastructure Manager (NIM)*.

Come anticipato, il NIM si occupa di gestire tutti gli aspetti di Neptune, dall'interazione con i sistemi operativi delle macchine fisiche del cluster, fino alla gestione dell'interfaccia utente. E' naturale che un sistema software del genere possa risultare complesso non solo dal punto di vista progettuale, ma anche per l'usabilità dalla prospettiva dell'utente. Al contempo, è necessario che Neptune, e quindi il NIM che ne è l'interfaccia software adoperata dall'utente, offra le sue funzionalità in modo semplice ed intuitivo, poiché è destinato ad utenti che non necessariamente conoscono tutti i dettagli dell'infrastruttura di cui si compone.

Perché si possa realizzare un tale proposito, è necessario innanzitutto astrarre i concetti e fornire una visione concisa dei servizi offerti. In questo capitolo si procederà dunque a riassumere tramite un modello dei casi d'uso i requisiti software del NIM, per poi definire l'interfaccia utente del sistema. Il modello dei casi d'uso verrà sviluppato a partire da quanto specificato nei precedenti capitoli, inoltre, verrà utilizzato come riferimento l'esempio di *workflow* presentato nel prossimo paragrafo, che specifica una *story*, ossia un esempio di utilizzo del sistema, così come immaginato all'atto dell'ideazione dello stesso.

Infine, verranno trattati gli aspetti di progettazione software risultanti dalle analisi precedentemente condotte.

III.1 Il workflow iniziale

Nel seguito è descritto per sommi capi il *workflow* dell'applicazione per i processi riguardanti la definizione dell'esperimento e la sua successiva allocazione. Questo *workflow* rappresenta l'idea iniziale di interazione degli utenti con il sistema, e verrà successivamente elaborata al fine di individuare gli elementi caratterizzanti del modello dei casi d'uso (attori e casi d'uso).

Il *workflow* descritto ipotizza l'allocazione manuale degli esperimenti. Gli attori coinvolti sono:

- **Amministratore**: inteso come amministratore del sistema, ossia, colui che ha il compito di gestire la configurazione di ogni nodo fisico facente parte del cluster. Fra i compiti dell'amministratore rientrano la scelta di quale esperimento allocare e la configurazione dei nodi virtuali sui nodi fisici;
- **Utente**: colui che definisce ed esegue esperimenti.

La descrizione del *workflow* è la seguente (Figura III.2):

1. L'utente definisce un esperimento indicando un nome univoco per lo stesso. L'esperimento mantiene una duplice informazione: il suo nome identificativo ed il nome-utente del suo creatore, che è anche l'utente che ha il compito di gestire l'esperimento stesso.
2. L'utente definisce una topologia specificando i nodi ed i link che la compongono. La topologia viene quindi associata all'esperimento. Da notare come la definizione della topologia sia totalmente slegata dalla reale struttura fisica del cluster. L'utente ignora completamente la struttura fisica del cluster e non ha necessità di conoscerla per definire la topologia del suo esperimento.

A questo punto, l'esperimento risulta completamente definito. La definizione dell'esperimento, nel modello immaginato, prevede infatti la specificazione di 3 informazioni:

- Identificativo dell'esperimento;
- Nome del gestore (creatore) dell'esperimento;
- Topologia.

All'esperimento cosi' definito è possibile associare ulteriori utenti che avranno la possibilità di intervenire sullo stesso, modificandolo o monitorandolo (Vedere il documento dei casi d'uso per avere ulteriori delucidazioni). Le informazioni sullo stato dell'esperimento sono invece gestite e specificate dal sistema.

3. Quando l'utente si considera soddisfatto della definizione dell'esperimento, lo colloca in una “coda di allocazione”. Questa operazione corrisponde a dire che “l'esperimento è totalmente definito e, quindi, ci si pone in attesa perché venga collocato sui nodi fisici del cluster”. Sostanzialmente, questa è l'operazione di transito dell'esperimento nello stato *waiting*.
4. L'amministratore esamina la coda di allocazione ed estrae un esperimento dalla stessa per procedere alla sua allocazione sui nodi fisici del cluster.
5. La procedura di allocazione corrisponde a:
   - Definire quali nodi fisici del cluster ospiteranno i nodi definiti nella topologia dell'esperimento;
   - Creare i nodi virtuali, sui nodi fisici, secondo quanto previsto nel precedente punto.

   Al termine di queste operazioni, l'amministratore avvia la fase di configurazione dei link..
6. A questo punto, l'esperimento definito dall'utente è stato allocato sui nodi del cluster, quindi, è possibile accedere ad ogni nodo tramite console remota, per effettuare tutte le operazioni necessarie all'esecuzione dell'esperimento.

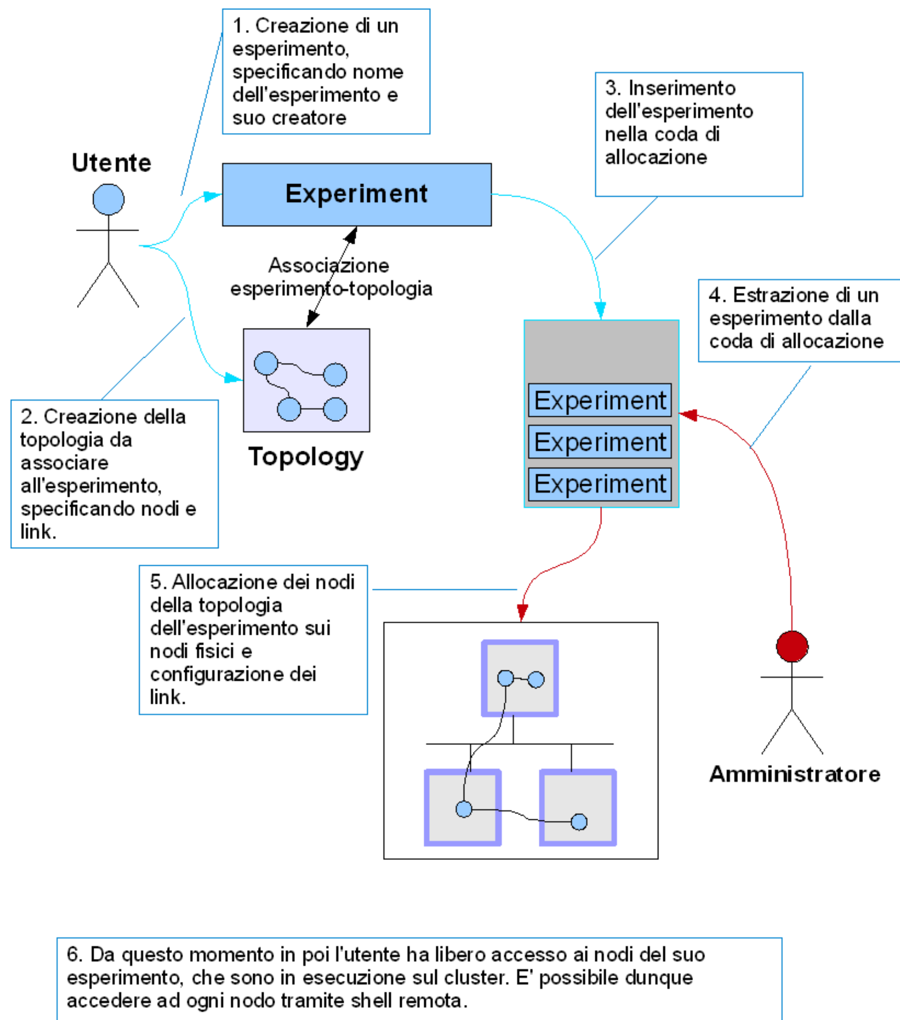


Figura III.1: Workflow di utilizzo del NIM per la creazione ed allocazione di un esperimento

III.2 Modello dei casi d'uso

La specifica dei casi d'uso è uno strumento utile alla comprensione dei requisiti funzionali dell'applicazione [27], che permette una pianificazione più efficace dello sviluppo. La descrizione dei casi d'uso sarà condotta tramite scenari. Uno scenario è una lista numerata di interazioni fra attore(i) e sistema.

Gli scenari alternativi sono presentati a seguito dello scenario principale. Il punto di inserimento dello scenario alternativo, all'interno dello scenario principale, è individuabile tramite il numero associato alla prima interazione dello scenario alternativo.

Per rappresentare l'uso di un caso d'uso all'interno di un altro, lo scenario incluso è nominato nello scenario del caso d'uso che include, al punto in cui si verifica. Tale nome viene rappresentato sottolineato, per evidenziarne il significato.

I casi d'uso sono categorizzati in 2 livelli:

- **Primary task:** sono i casi d'uso che interessano in prima persona l'utente, ossia tutte quelle operazioni legate alla business logic;
- **Summary**: sono casi d'uso che includono più casi d'uso del precedente livello, per realizzare un caso d'uso complesso.

III.2.1 Actors

- ***System Admin***: è l'amministratore di sistema. Si occupa di gestire il sistema nella sua interezza, operando azioni di monitoraggio e fornendo supporto agli altri utenti quando necessario. Ha sostanzialmente l'accesso a tutte le operazioni consentite sul sistema.
- ***Experiment Owner***: è il creatore di un esperimento e come tale ne diviene l'amministratore. Ha il compito di gestire l'esperimento di cui è amministratore;
- ***Experiment User***: è l'utente di un esperimento, può operare diverse azioni su di un esperimento per cui è stato specificato come user.

III.2.2 Casi d'uso

Nel seguito sono presentati i casi d'uso individuati per il *neptune infrastructure manager*.

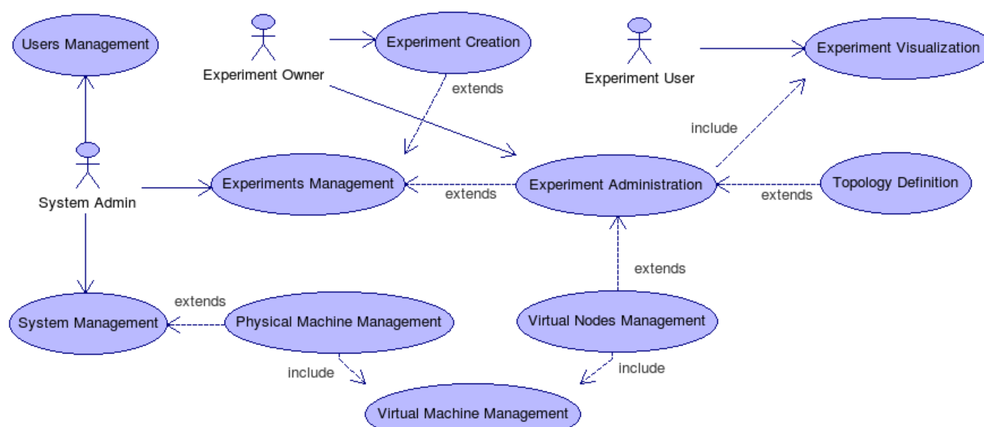


Figura III.2: Modello dei casi d'uso

III.2.2.1 Users Management

Goal: L'attore gestisce gli utenti del sistema	***Level:*** *primary task*
Trigger: l'attore richiama la funzionalità del sistema	***Actors***: System admin
Preconditions: Nessuna	
Main Success Scenario: (1) *Actor*: accede ad una schermata riassuntiva degli utenti del sistema e seleziona una fra le seguenti operazioni: aggiunta nuovo utente, modifica di un utente esistente, eliminazione di un utente esistente; (2) System: verifica la correttezza dell'operazione ed effettua le opportune modifiche ai dati sugli utenti;	
Extensions: (2a) Se il processo è in qualche sua parte errato, il sistema informa l'attore del problema riscontrato.	

III.2.2.2 System Management

Goal: L'attore gestisce la configurazione del sistema	***Level:*** *summary*
Trigger: l'attore richiama la funzionalità del sistema	***Actors***: System admin
Preconditions: Nessuna	
Main Success Scenario: (1) *Actor*: accede ad una schermata di configurazione del sistema e seleziona una fra le seguenti operazioni: aggiunta nuova macchina al sistema, modifica di macchina già esistente nel sistema, eliminazione di una macchina dal sistema, modifica dei parametri di configurazione del sistema, physical machine management; (2) System: mostra all'utente l'opportuna interfaccia per il comando selezionato, al fine di raccogliere i dati necessari a compiere l'operazione; (3) Actor: inserisce i dati richiesti e conferma l'operazione; (4) System: esegue l'operazione richiesta previa convalidazione dei dati inseriti dall'utente.	
Extensions: (4a) Se il processo è in qualche sua parte errato, il sistema informa l'attore del problema riscontrato.	

III.2.2.3 Experiments management

Goal: L'attore gestisce gli esperimenti del sistema	***Level:*** *summary*
Trigger: l'attore richiama la funzionalità del sistema	***Actors***: System admin
Preconditions: Nessuna	
Main Success Scenario: (1) *Actor*: accede ad una schermata riassuntiva degli esperimenti definiti sul sistema e seleziona una fra le seguenti operazioni: experiment creation, experiment administration; (2) System: verifica la correttezza dell'operazione e mostra l'opportuna interfaccia;	
Extensions: (2a) Se il processo è in qualche sua parte errato, il sistema informa l'attore del problema riscontrato.	

III.2.2.4 Experiment creation

Goal: L'attore crea un nuovo esperimento	***Level:*** *primary task*
Trigger: l'attore richiama la funzionalità del sistema	***Actors***: System admin/Experiment Owner
Preconditions: Nessuna	
Main Success Scenario: (1) *Actor*: fornisce il nome del nuovo esperimento; (2) System: verifica l'univocità del nome e crea il nuovo esperimento, impostandone come owner	

l'utente che ha fatto la richiesta di creazione;
Extensions: (2a) Se il processo è in qualche sua parte errato, il sistema informa l'attore del problema riscontrato.

III.2.2.5 Experiment administration

Goal: L'attore gestisce gli aspetti riguardanti un esperimento	***Level:*** *summary*
Trigger: l'attore richiama la funzionalità del sistema	***Actors***: System admin/Experiment Owner
Preconditions: L'attore ha i diritti di accesso all'esperimento	
Main Success Scenario: (1) experiment visualization; (2) Actor: accede ad una delle seguenti funzioni: virtual nodes management, topology definition. Oppure comanda una delle seguenti operazioni sull'esperimento: deploy, configure, swap-out; (3) System: mostra l'opportuna interfaccia di inserimento dati (se necessaria) per l'operazione richiesta; (4) Actor: inserisce le informazioni necessarie; (5) System: verifica la correttezza dell'operazione ed effettua le opportune modifiche ai dati sugli utenti;	
Extensions: (5a) Se il processo è in qualche sua parte errato, il sistema informa l'attore del problema riscontrato.	

III.2.2.6 Experiment visualization

Goal: L'attore visualizza lo stato dell'esperimento	***Level:*** *primary task*
Trigger: l'attore richiama la funzionalità del sistema	***Actors***: System admin/ Experiment Owner/ Experiment User
Preconditions: L'attore ha i diritti di accesso all'esperimento	
Main Success Scenario: (1) *Actor*: accede ad una schermata riassuntiva dello stato del sistema; (2) System: recupera i dati riguardanti l'esperimento e li mostra all'attore.	
Extensions: (2a) Se il processo è in qualche sua parte errato, il sistema informa l'attore del problema riscontrato.	

III.2.2.7 Topology definition

Goal: L'attore definisce una topologia per un esperimento	***Level:*** *primary task*
Trigger: l'attore richiama la funzionalità del sistema	***Actors***: Experiment Owner
Preconditions: L'attore ha i diritti di accesso all'esperimento	
Main Success Scenario: (1) *Actor*: accede ad una schermata di definizione della topologia, e seleziona il metodo di definizione prescelto; (2) System: mostra un'opportuna interfaccia di definizione della topologia; (3) Actor: definisce la topologia e la invia al sistema: (4) System: convalida la topologia e mostra il risultato all'attore. In base a tale risultato, aggiorna lo stato dell'esperimento a cui la topologia è collegata.	
Extensions: (4a) Se il processo è in qualche sua parte errato, il sistema informa l'attore del problema riscontrato e ripristina il precedente stato dell'esperimento.	

III.2.2.8 Physical Machine management

Goal: L'attore visualizza lo stato di una macchina fisica	***Level:*** *primary task*

Trigger: l'attore richiama la funzionalità del sistema	***Actors***: System admin
Preconditions: la macchina fisica è definita sul sistema	
Main Success Scenario: (1) *Actor*: accede ad una schermata riassuntiva dello stato della macchina fisica; (2) System: fornisce i dati riguardanti l'utilizzo delle risorse della macchina fisica e mostra quali macchine virtuali sono attualmente in esecuzione su di essa; (3) Actor: per ogni macchina virtuale in esecuzione sulla macchina fisica, può richiamare la funzione <u>virtual machine management</u>;	
Extensions: (2a) Se il processo è in qualche sua parte errato, il sistema informa l'attore del problema riscontrato.	

III.2.2.9 Virtual Nodes management

Goal: L'attore gestisce I nodi della topologia dell'esperimento	***Level:*** *primary task*
Trigger: l'attore richiama la funzionalità del sistema	***Actors***: Experiment Owner
Preconditions: L'attore ha i diritti di accesso all'esperimento e quest'ultimo è allocato sul sistema	
Main Success Scenario: (1) *Actor*: accede ad una schermata riassuntiva dello stato dei nodi allocati dell'esperimento; (2) System: recupera i dati riguardanti i nodi dell'esperimento e li mostra all'attore; (3) Actor: per ogni nodo mostrato può eseguire l'operazione <u>virtual machine management</u>.	
Extensions: (2a) Se il processo è in qualche sua parte errato, il sistema informa l'attore del problema riscontrato.	

III.2.2.10 Virtual machine management

Goal: L'attore gestisce lo stato della virtual machine	***Level:*** *primary task*
Trigger: l'attore richiama la funzionalità del sistema	***Actors***: System admin/ Experiment Owner
Preconditions: L'attore ha i diritti di accesso all'esperimento	
Main Success Scenario: (1) *Actor*: seleziona uno dei seguenti comandi: start, reboot, pause, shutdown; (2) System: esegue il comando sull'opportuna virtual machine.	
Extensions: (2a) Se il processo è in qualche sua parte errato, il sistema informa l'attore del problema riscontrato.	

III.3 L'interfaccia utente

Fin da principio abbiamo presentato Neptune come un sistema multi-utente, che punta alla massima generalità e semplicità possibili. Questo approccio è stato mantenuto anche nella definizione dell'interfaccia utente e nella scelta tecnologica che ne ha guidato la progettazione.

Per garantire un accesso ad utenti remoti, senza costringere gli utenti stessi a confrontarsi con problematiche di installazione e configurazione di software aggiuntivo, si è scelta la

strada dell'interfaccia web. La scelta è stata incoraggiata dallo sviluppo sostenuto che le tecnologie web hanno avuto negli ultimi anni, che hanno portato una considerevole evoluzione nella qualità delle interfacce proposte, pur mantenendo l'accessibilità e la semplicità di utilizzo caratteristiche di una pagina web. Tali tecnologie vengono spesso riassunte in un unico acronimo: *ajax*. L'acronimo sta per *Asynchronous Javascript And Xml* ma, nel tempo, la sua accezione è diventata molto più generale. Sostanzialmente, le tecniche *ajax* trasformano il browser in un ambiente in cui eseguire un'applicazione *client* evoluta (tipicamente, tali applicazioni vengono definite *rich client*, facendo riferimento alla sia ricchezza di contenuti dinamici, che alla ricchezza dell'esperienza dell'utente) [28].
Escludendo alcune problematiche tecniche legate all'utilizzo del *browser* come ambiente di esecuzione [29] e di HTTP come protocollo di trasporto, le tecniche *ajax* consentono di realizzare dei client del tutto simili ai classici *client* realizzati per ambienti *desktop*. Per queste motivazioni, la progettazione dell'interfaccia grafica ha seguito un approccio non vincolato ai classici stili delle pagine web, ma più orientato alla struttura di un'applicazione desktop vera e propria.
Le linee guida nella definizione dell'interfaccia sono state, oltre che la ricerca di semplicità ed usabilità, la realizzazione di una vista omogenea, che garantisse una semplice espandibilità delle funzionalità esposte.
L'organizzazione dell'interfaccia comprende due “viste”: *log-in view* ed *operative view*.

III.3.1 Log-in view

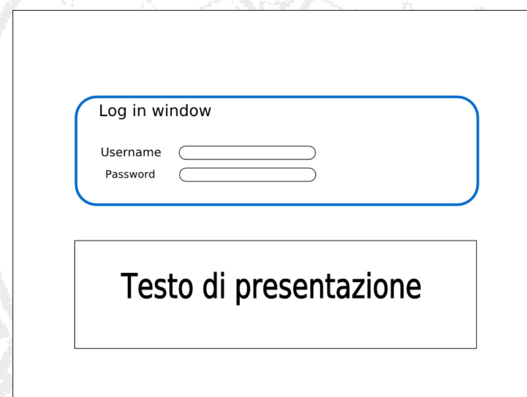


Figura III.3: Log-in view

La schermata di *log-in* presenta semplicemente i campi per l'inserimento di nome utente e password utili all'identificazione e autenticazione dell'utente. Si prevede inoltre uno spazio per riportare eventuali informazioni come notizie, indicazioni, anche variabili nel tempo.

III.3.2 Operative View

La *operative view* si occupa di gestire tutta la logica dell'applicazione. L'organizzazione di questa vista è fatta per ricordare l'interfaccia di una comune applicazione desktop, con la vista divisa in due parti: una parte fissa, che contiene comandi ed una console di stato, ed una parte variabile che contiene le informazioni in base al comando richiamato.

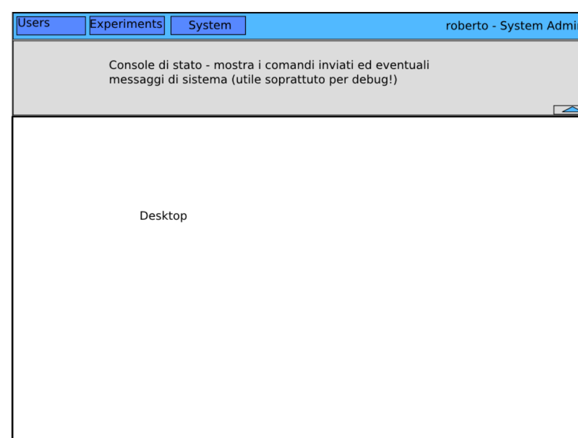


Figura III.4: Operative view

La parte indicata come “Desktop” nella figura precedente, è la parte variabile che si adatta al comando richiamato.

L'interfaccia presentata può essere facilmente estesa aggiungendo ulteriori elementi ai menù dei comandi, garantendo un certo grado di omogeneità. Ulteriori esempi di interfaccia, in cui sono presentate alcune configurazioni del campo “Desktop” sono mostrati nel seguito:

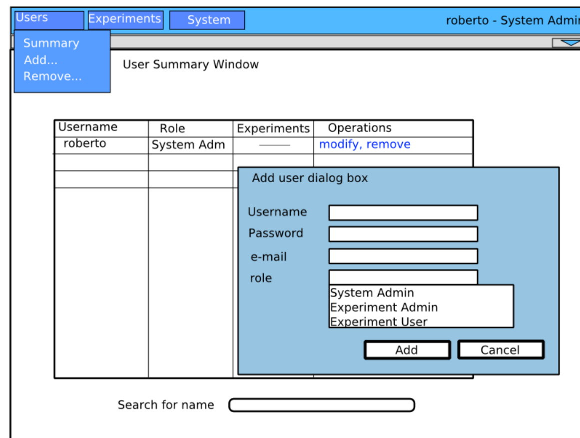


Figura III.5: Interfaccia per la gestione degli utenti

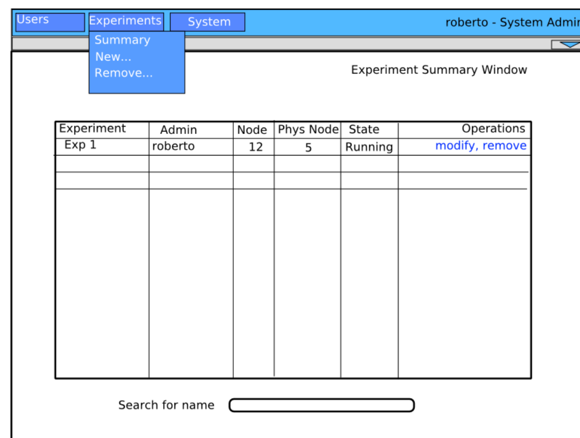


Figura III.6: Interfaccia per la gestione degli esperimenti

In Figura III.5 ed in Figura III.6 sono mostrate le interfacce riservate alla gestione di utenti ed esperimenti. Chiaramente tali interfacce, come d'altronde le successive, sono da intendersi come una guida ed un modello di base, ma non come una definizione puntuale dell'interfaccia completa.

In Figura III.7 è invece mostrata l'interfaccia per la consultazione e modifica dello stato di un esperimento. Chiaramente un'interfaccia del genere presenterà differenti funzioni a seconda del ruolo con cui si accede al sistema.

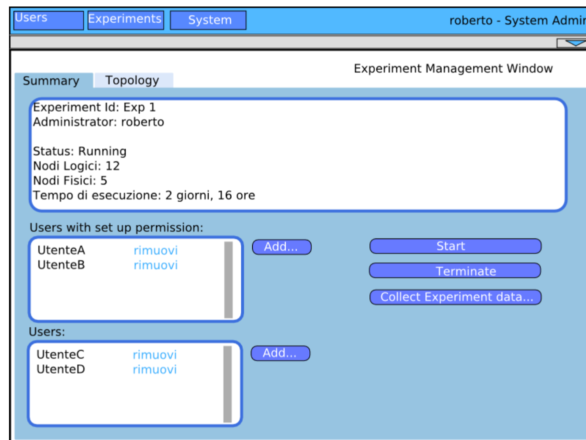


Figura III.7: Interfaccia per la gestione di un esperimento

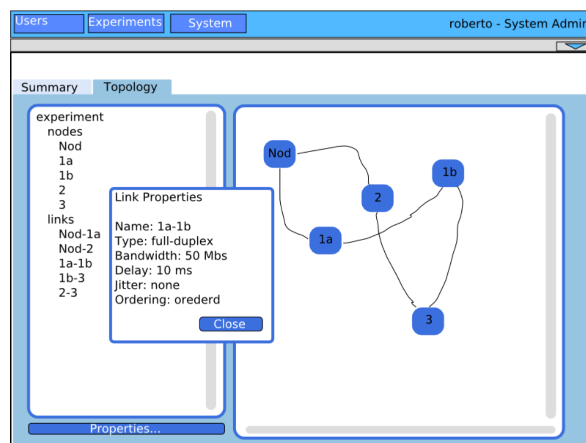


Figura III.8: Interfaccia per la definizione della topologia

L'interfaccia per la definizione della topologia (Figura III.8) fa uso di finestre pop-up interne al browser, realizzate dinamicamente, per rappresentare informazioni strutturate che richiedono operazioni particolari. Questo consente di non adoperare nuove finestre del browser, che potrebbero risultare fastidiose o incappare nelle funzionalità di “blocco pop-up” che generalmente penalizzano l'esperienza d'uso dell'utente.

Per le stesse motivazioni, anche necessità più avanzate come il disegno del grafo della topologia, saranno realizzate cercando di non adoperare complicazioni nell'interfaccia dovute all'utilizzo di *plug-in* esterni/*applets* se non in casi di obiettiva necessità.

III.4 Progettazione dell'architettura

Nel precedente capitolo si è introdotto l'approccio a livelli necessario alla realizzazione delle funzionalità del NIM. Come anticipato, tale approccio è da intendersi come un modello concettuale a cui ricondursi, che non è necessariamente realizzato fedelmente. La suddivisione in tre livelli è infatti una vista molto ad alto livello del sistema, che nasconde anche i dettagli relativi, ad esempio, alle problematiche di realizzazione di un'interfaccia remota.

L'architettura software del NIM verrà quindi ora riesaminata per tener conto di maggiori dettagli, successivamente verrà evidenziato come il modello architetturale più dettagliato presentato nel seguito sia riconducibile alla vista logica a livelli che ne ha guidato la definizione.

La definizione del modello architetturale deve garantire le tipiche caratteristiche richieste ad un'applicazione complessa: modularità, isolamento, manutenibilità. Per quanto indicato in precedenza dai casi d'uso, ma soprattutto dalla descrizione dell'interfaccia utente desiderata, si delinea innanzitutto un vincolo progettuale ben definito. L'interfaccia utente, infatti, è *web based* il che comporta inevitabilmente l'uso di un *browser* ed una comunicazione *client-server* tramite HTTP. Il vincolo appena citato impone che il NIM abbia un'architettura composta da due moduli ben definiti, che vanno a realizzare il lato *client*, che adopererà un browser come ambiente di esecuzione, ed un lato *server*, che, dovendo ricevere richieste tramite HTTP, richiede l'utilizzo di un *web server* per poter rispondere al *client*.

In applicazioni del genere, è ormai quasi uno *standard de facto* l'utilizzo del modello architetturale *Model-View-Controller (MVC)*. MVC è un *pattern* architetturale molto diffuso nello sviluppo di interfacce grafiche di sistemi *software object-oriented*. Originariamente impiegato dal linguaggio Smalltalk [30], il *pattern* è stato esplicitamente o implicitamente sposato da numerose tecnologie moderne.

Il *pattern* è basato sulla separazione dei compiti fra i componenti software che interpretano tre ruoli principali:

- il *model* contiene i dati e fornisce i metodi per accedervi;

- il *view* visualizza i dati contenuti nel model;
- il *controller* riceve i comandi dell'utente (in genere attraverso il *view*) e li attua modificando lo stato degli altri due componenti.

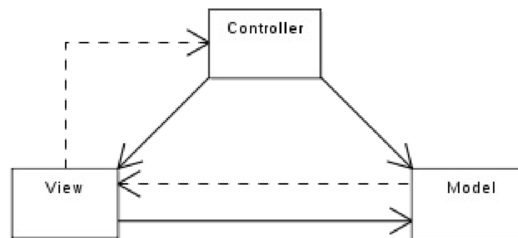


Figura III.9: Il modello di interazione comunemente adoperato in MVC

Questo schema, fra l'altro, implica anche la tradizionale separazione fra la logica applicativa (in questo contesto spesso chiamata "logica di *business*"), a carico del *model*, e l'interfaccia utente, a carico del *view* e del *controller* [31] [32].

Il NIM adopera anch'esso il *pattern* MVC, che presenta indubbi vantaggi nella gestione del software, come illustrato poc'anzi. L'utilizzo del MVC va però adattato alle esigenze particolari dovute alla comunicazione *client-server* ed al modello di *client* adoperato dal NIM. Abbiamo infatti anticipato che il *client* è un'applicazione sufficientemente complessa, capace di elaborazioni proprie non banali. Poiché è il *client* che si occupa dell'interazione con l'utente, è una soluzione plausibile lasciare che l'intera gestione gestione dei componenti del *view* sia ad esso riservata. Viceversa, il *model* richiede interazioni multiple con diversi componenti *software* (ricordiamo che il *model* del NIM deve interfacciarsi certamente con gli *hypervisor* dislocati sulle macchine fisiche del cluster), inoltre, deve garantire la corretta gestione delle interazioni fra più utenti ed è quindi preferibile che sia interamente gestito dal *server*, che può contare su di un ambiente di esecuzione non limitato dalle caratteristiche del *browser*.

Il *controller* è invece un componente che ha interazioni a livelli differenti, poiché opera sia per aggiornare la *view* in base alle azioni intraprese dall'utente, sia per eseguire operazioni complesse sul *model*.

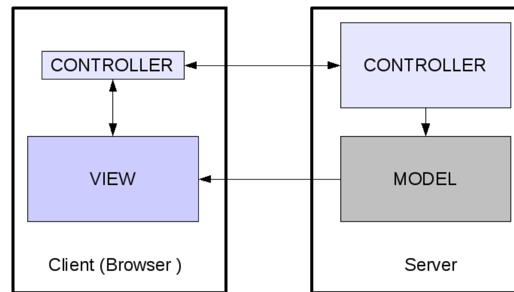


Figura III.10: Architettura del Neptune Infrastructure Manager

In Figura III.10 è mostrato il modello architetturale e la sua dislocazione fra *client* e *server*. Il *controller*, come anticipato, opera su entrambi i lati. In particolare, la componente del *controller* che si trova sul *client* si occupa di gestire gli eventi strettamente connessi all'interfaccia grafica (quindi alla *view*). La componente che risiede sul *server* gestisce invece i comandi complessi ricevuti dal *client* per tradurli in opportune operazioni sul *model*.

All'inizio di questo paragrafo abbiamo detto che era possibile un *mapping* fra la suddivisione logica in livelli e l'effettiva architettura basata su MVC. Tale *mapping* è abbastanza semplice se si esaminano le varie funzionalità realizzate da ciascun componente del modello MVC. Il *view* ed il *controller* vanno a realizzare principalmente le funzionalità riguardanti l'interfaccia utente. Nel modello a livelli, tali funzionalità erano parte del livello *emulation manager (em)* che introduceva oltre a queste caratteristiche anche la gestione degli esperimenti e delle topologie e la gestione della multi-utenza. Quindi, l'*em* certamente include al suo interno i componenti *view* e *controller*, tuttavia, date le funzionalità più complesse, come ad esempio la gestione della logica legata agli esperimenti, anche parte del *model* è inclusa in questo livello.

I rimanenti due livelli, il *physical machines manager* ed il *virtual cluster manager*, operano invece sostanzialmente senza un'interazione diretta con l'utente, se non tramite l'*em* che si occupa di introdurre la logica basata sul concetto di emulazione di rete. Questo dovrebbe chiarire come questi due livelli trovino posto soltanto all'interno del *model* (Figura III.9).

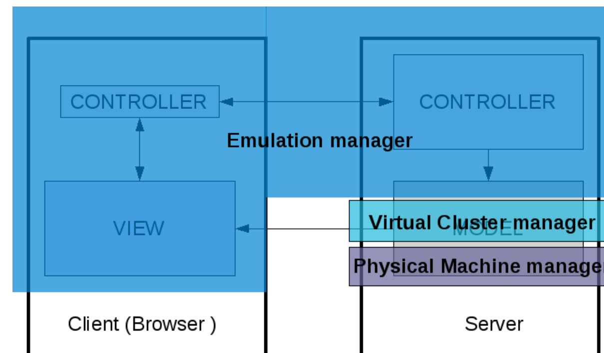


Figura III.11: Mapping del modello concettuale a livelli nell'architettura MVC

III.5 Il sistema per la gestione delle autorizzazioni

Abbiamo fin da principio, parlando del NIM o più in generale di Neptune, specificato la necessità di multi-utenza che caratterizza il sistema. Un'applicazione in cui intervengono più utenti, ciascuno con caratteristiche di accesso proprie, richiede un accurato esame per definire un modo di gestione delle utenze che non risulti troppo macchinoso da un lato, o troppo banale dall'altro. Questo implica la necessità di introdurre un componente di gestione dell'utenza che permetta di definire con precisione le caratteristiche di accesso di un utente secondo le necessità del NIM (da qui in poi adopereremo il termine *policy*, ad indicare un insieme di regole di accesso definite a livello di amministrazione del sistema), allo stesso tempo, tale componente deve risultare un peso minore nel contesto dell'intera applicazione, senza quindi gravare sullo sviluppo dei componenti di *business*, presentando un piccolo peso “logico”, e senza gravare in termini prestazionali.

Con “piccolo peso logico” si intende che il NIM deve essere indipendente dal sistema di gestione delle utenze per quanto possibile, poiché la gestione delle utenze di per se introduce concetti elaborati che esulano dagli interessi del NIM, per quanto siano necessari per garantire la corretta gestione degli utenti.

Il componente per la gestione delle utenze prende il nome di *Identification and Authorization System* (IAS) e deve compiere sostanzialmente due operazioni:

- Identificare un utente;

- Verificare se una data operazione è consentita per un utente (ossia, verificare se l'utente ha i permessi necessari per eseguire l'operazione).

La prima operazione consiste nel riconoscere l'identità di un utente. Più precisamente, il sistema deve ottenere dall'utente una presentazione (nome utente) e verificare tramite una informazione confidenziale che l'utente sia effettivamente chi dice di essere. Per mantenere un approccio quanto più semplice possibile, l'autenticazione supportata in un primo momento è soltanto quella attraverso una *password*, che, d'altronde, è quella più adoperata e funzionale nel contesto del web.

La seconda operazione ha lo scopo di verificare quali operazioni l'utente può effettuare sul sistema. Da qui in poi ci riferiremo alla concessione di una operazione ad un utente utilizzando la parola "permesso".

Gestire i permessi associati a ciascun utente, considerando gli utenti singolarmente, potrebbe divenire un compito di complessità eccessiva viste le necessità dell'applicazione. Infatti, abbiamo già anticipato come nell'utilizzo del sistema siano identificabili sostanzialmente due ruoli, ossia quello di *System admin* e di *Experiment Owner/User*. In realtà, *Experiment Owner* ed *Experiment User* sarebbero due ruoli distinti, ma in questo caso, la discriminante fra i due ruoli è semplicemente la valutazione di chi è il creatore dell'esperimento su cui si vuole eseguire un'operazione. In quest'ultimo caso è quindi una valutazione del permesso gestita per utente e non per ruolo, e verrà presentato successivamente il modo in cui si risolve questa problematica.

Per quanto premesso, si è deciso di adoperare un sistema di associazione dei permessi basato su ruoli (*role based access control*) [33]. Ciascun utente è associato ad un ruolo. Ogni ruolo ha associati dei permessi per delle operazioni.

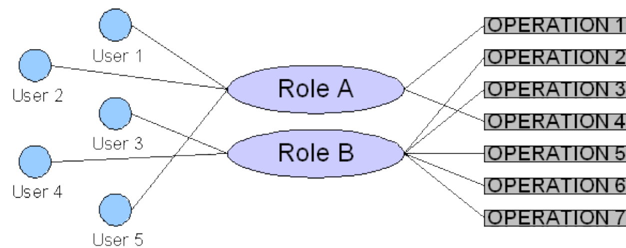


Figura III.12: Associazione utenti-ruoli-permessi

L'amministratore di sistema ha il compito di definire i ruoli ed i permessi associati, e di assegnare a ciascun utente il suo ruolo. Gli utenti li definiremo in modo più generale "*principal*" in seno all'IAS.

Perché alle operazioni sia associato un permesso, ne deve essere esposta l'interfaccia ed un identificativo univoco. In questo modo il permesso è associabile all'operazione cui fa riferimento. IAS mantiene una collezione di tutte le operazioni cui sono associati permessi. Una volta definiti i permessi gestiti dal sistema, questi devono essere associati ai ruoli.

La gestione dei ruoli è compito dell'amministratore di sistema ed è gestibile dinamicamente. In questo modo si evitano complicazioni in caso di ridefinizione delle *business operation* (aggiunta o eliminazione delle stesse). In altre parole, una modifica alla logica di *business* non richiede una riscrittura dell'IAS. Questo risponde in parte al requisito di piccolo peso “logico” identificato in precedenza.

Tramite la definizione dinamica di ruoli (Aggiunta ed eliminazione ruoli, cambiamento dei permessi associati ecc.), si gestiscono dinamicamente le *policy* di accesso al sistema.

III.5.1 Il command pattern

L'IAS adopera un approccio ben noto nell'ingegneria del software per la definizione delle operazioni. L'approccio prende solitamente il nome di *command pattern* [34], e prevede che ciascuna operazione sia incapsulata in un elemento proprio, che ne permetta la gestione come un componente applicativo utilizzabile e scambiabile fra altri componenti,

invece di introdurre l'algoritmo dell'operazione in un componente più grande che richiede un'opportuna logica di intervento per poter eseguire l'operazione desiderata.

Questo, di fatto, crea un insieme di componenti, ciascuno rappresentante di un'operazione, che consentono di gestire in modo molto flessibile le operazioni del sistema, garantendo non solo la possibilità di gestirne i permessi di esecuzione, ma anche la capacità di aggiungere o rimuovere dinamicamente le operazioni definite sul sistema.

III.5.2 Architettura dell'IAS

L'architettura presentata in Figura III.4 riassume le caratteristiche fondamentali dell'IAS [35]. Assumendo un approccio volutamente generico, possiamo dire che il sistema di sicurezza si frappone fra i livelli di *business* e di *presentazione* (il livello che espone l'interfaccia utente). Ogni operazione richiesta dal livello di *presentazione* passa attraverso l'interfaccia dell'IAS che esegue i dovuti controlli per verificarne i permessi di esecuzione e poi richiederne l'esecuzione al *business layer*, qualora i controlli siano superati.

L'IAS esegue i controlli utilizzando le informazioni fornite da *Role Manager* e *Principal Manager*:

- *Role Manager*: gestice i ruoli definiti dall'amministratore di sistema. Tutte le informazioni sui ruoli sono acquisibili tramite questo componente. In particolare il *Role Manager* fornisce gli identificativi dei ruoli e gli identificativi delle operazioni associate ai ruoli stessi.
- *Principal Manager*: gestisce le informazioni relative agli utenti. Tutte le informazioni sugli utenti, quindi l'identificativo, il ruolo, la password di riconoscimento associata, sono ottenibili tramite questo componente.

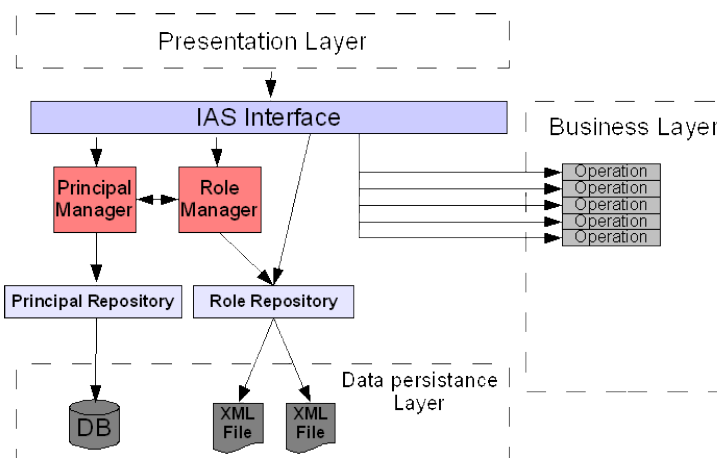


Figura III.13: Architettura dell'IAS

Le informazioni gestite sono conservate in dei repository. Si definiscono due tipi di repository: *Role Repository*, *Principal Repository*. E' importante notare come i repository definiti dall'architettura sono logici, ossia non hanno una specifica implementazione. L'implementazione dei *repository* è infatti affidata al sistema che adopera l'IAS (In figura viene mostrato un *data persistance* layer, ad indicare una parte del sistema designata alla gestione della persistenza dei dati) e può adoperare varie strategie: database, file xml, ecc.

III.5.3 Integrazione dell'IAS nell'architettura del NIM

L'IAS è un componente che nasce per integrarsi fra i livelli di presentazione e di *business*. Nell'architettura del NIM questi livelli, adoperando l'approccio MVC, sono facilmente identificabili. Abbiamo infatti detto che tutte le operazioni risiedono nel componente *model*, mentre l'interfaccia utente è gestita dai componenti *view* e *controller*. L'IAS viene quindi a frapporsi fra il *controller* e il *model*. Più precisamente, potremmo considerare l'IAS come una parte del *controller*, infatti, nell'accezione propria del modello MVC, il *controller* ha il compito di gestire operazioni che richiedono interventi particolari, come ad esempio la verifica di eventuali permessi di esecuzione.

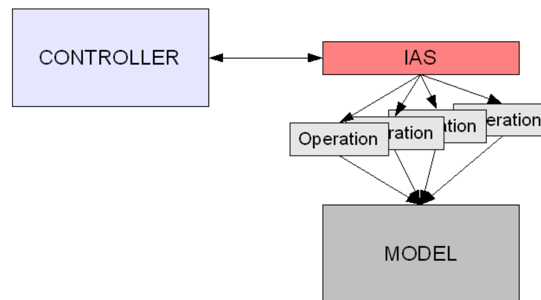


Figura III.14: Integrazione di IAS in NIM

III.6 La scelta della tecnologia

Fino ad ora non si è introdotto nessun vincolo tecnologico, se non quello riguardante la realizzazione dell'interfaccia. Il NIM, infatti, potrebbe essere teoricamente realizzato con qualsiasi tecnologia ritenuta adatta. La scelta della tecnologia da adoperare è dunque un'ulteriore scelta di progetto, che va effettuata al fine di garantire una realizzazione agevole ed efficiente del sistema.

L'architettura del sistema adopera un approccio *client-server*, con interazione fra le due parti tramite il protocollo HTTP. Questo indica chiaramente la necessità di adoperare un *web server*, o comunque un componente in grado di gestire HTTP dal lato *server* (Il lato client è realizzato internamente ad un *browser*, quindi non presenta problematiche particolari a riguardo). La realizzazione *ex novo* di un *web server* non è una strada conveniente, poiché da un lato richiederebbe un tempo di realizzazione consistente, dall'altro, sono tante le tecnologie più che consolidate che gestiscono in modo ottimale queste funzionalità.

Ci si è dunque orientati verso tecnologie che garantissero un supporto al protocollo HTTP e che avessero al contempo comunità vaste e attive, per adoperare, dove possibile, componenti già consolidate. Fra le varie tecnologie esaminate, Java è risultata quella predominante, poiché non solo rispecchiava le necessità appena esposte, ma garantiva una tecnologia di sviluppo unica fra lato *client* e lato *server*, caratteristica molto utile per un rapido sviluppo dell'applicazione.

Senza soffermarci troppo sulle possibilità di integrazione con *web server* di Java, più che note, è bene sottolineare come la comunità Java, intesa come insieme di liberi sviluppatori e aziende, abbia negli ultimi anni sviluppato strumenti utili allo sviluppo di applicazioni web evolute e con interfacce *desktop-like*. Fra tutti questi strumenti, uno in particolare è risultato particolarmente indicato per la realizzazione del lato *client* dell'applicazione: il *Google Web Toolkit(GWT)* [36].

GWT è un tool sviluppato da *Google*, che permette la realizzazione di applicazioni *client* che adoperano come ambiente di esecuzione un *browser*. GWT adopera la sintassi Java ed un insieme di librerie software che realizzano le funzionalità tipiche dei *framework* per le interfacce utente, oltre ad esporre le funzionalità collegate al *document object model* ed al *browser*. Il tool, per realizzare il client e permetterne l'esecuzione all'interno del browser, effettua una traduzione della sintassi Java in linguaggio Javascript, effettuando anche delle ottimizzazioni sul codice prodotto. Il risultato è la possibilità di sviluppare in Java l'intera applicazione [37], ignorando quasi del tutto la necessità di dover eseguire il lato *client* internamente ad un *browser*, se non per gli ovvi vincoli che tale ambiente di esecuzione presenta.

Questo insieme di considerazioni ha quindi fatto ricadere la scelta tecnologica su Java, inteso come insieme delle tecnologie Java, Servlet, Jsp, per la realizzazione del lato server con l'integrazione di GWT per la realizzazione del client.

III.7 Introduzione dei vincoli tecnologici

Avendo fissato la tecnologia da adoperare, vediamo come questa scelta va ad influenzare il disegno architetturale generale presentato in precedenza.

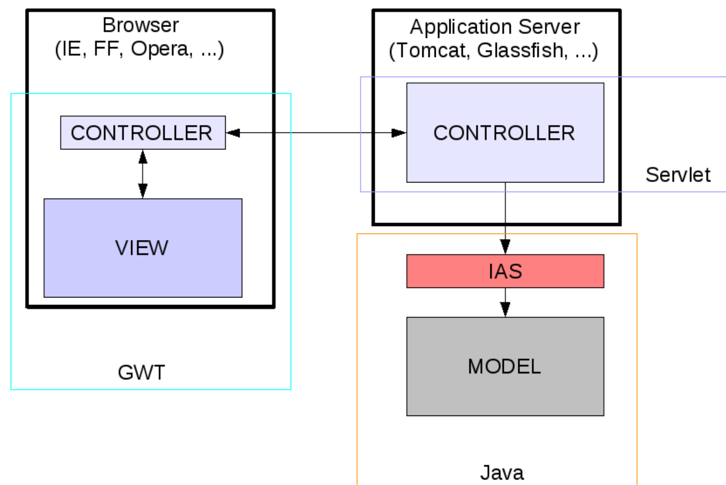
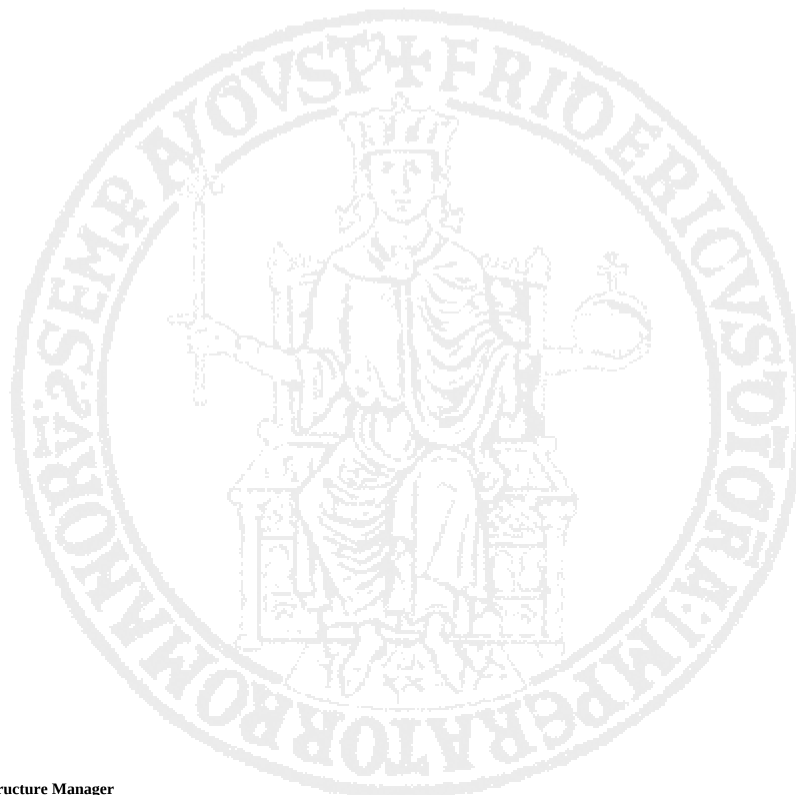


Figura III.15:

In Figura III.3 è presentato il modello architetturale generale, rivisto in seguito all'introduzione dei vincoli tecnologici. Sono due gli aspetti fondamentali da mettere in evidenza su tale modello:

- Il livello che abbiamo identificato come *emulation manager*, ossia il livello che gestisce oltre alla logica di emulazione anche l'interfaccia utente, è l'unico che adopera una realizzazione che prevede una sinergia fra più tecnologie: GWT, Servlet, Java. Questo aspetto era atteso, poiché a questo livello si realizza la parte di comunicazione fra *client* e *server* e dunque, è il livello che si confronta con i vincoli derivanti dall'interfaccia utente;
- I livelli *vcm* e *pmm* sono entrambi realizzati in pura tecnologia Java, non presentano quindi complessità legate all'eventuale introduzione di *framework* o tecnologie esterne. Quest'aspetto è fondamentale se si considera che, generalmente, gli elementi costitutivi la logica di *business* tendono a consolidarsi nel tempo e ad assumere caratteristiche di robustezza derivanti dall'esperienza maturata nel corso dello sviluppo e dell'utilizzo nel tempo. Legare questi componenti a *framework* esterni comporterebbe la necessaria manutenzione dei componenti stessi a seguito di variazioni dei *framework* adoperati. Inoltre, i componenti risulterebbero nel complesso meno indipendenti, poiché l'utilizzo di *framework*, per quanto talvolta comodo, impone un modello di sviluppo che è quello del *framework* stesso. Questo

potrebbe comportare serie difficoltà qualora dovessero cessare le attività di supporto a tale *framework*, o, comunque, se non venissero supportate in modo adeguato le eventuali nuove tecnologie emergenti.



Capitolo IV
La realizzazione del Neptune Infrastructure Manager

Dopo aver esaminato nel dettaglio i principali aspetti progettuali del NIM da un punto di vista generale, in questo capitolo si esaminano i dettagli riguardanti la realizzazione del progetto. Per garantire un approccio efficiente alla realizzazione del NIM, è stato innanzitutto necessario gestire l'organizzazione del progetto, in senso più gestionale che progettuale. Infatti, per quanto l'organizzazione del progetto sia una attività puramente gestionale, slegata dal contesto della progettazione e della realizzazione, è tuttavia necessario introdurre i principi che hanno guidato questa attività, al fine di chiarire come è stato effettivamente condotto il lavoro per garantire tempi di realizzazione brevi, senza però penalizzare lo sforzo di modularità e coerenza della realizzazione con quanto fin qui presentato.

Per la realizzazione del NIM si è deciso di non seguire una suddivisione dei progetti basata sui livelli presentati nell'architettura logica, preferendo piuttosto una suddivisione in base ai componenti del modello MVC. Questa decisione deriva dalle problematiche di integrazione che potrebbero presentarsi se si adoperasse anche nella pratica la suddivisione logica in livelli presentata. I componenti del modello MVC nascono invece per essere poco legati fra loro, favorendo la suddivisione del progetto in sotto-progetti di dimensione minore. Si è dunque suddiviso il progetto in tre sotto-progetti:

- Neptune management library: è il progetto che realizza il *model*. In questo progetto sono realizzati tutti i meccanismi necessari al funzionamento del sistema Neptune. Trovano dunque realizzazione i livelli *pmm*, *vcm* e la parte del livello *em* che

riguarda l'emulazione. La restante parte, che riguarda la gestione degli utenti e la realizzazione della *user interface*, è lasciata agli altri sotto-progetti;

- Identification and authorization system: è il progetto che realizza il sistema IAS. E' un progetto realizzato per garantire un sistema flessibile e semplice per la gestione di utenti e ruoli, che è sviluppato in modo del tutto slegato dalla logica di Neptune [35]. Viene integrato al fine di gestire le problematiche riguardanti l'identificazione e l'autorizzazione delle operazioni;
- Neptune Web Interface: è il progetto che si occupa di realizzare l'interfaccia utente e quindi l'interazione fra *client* e *server*. Per queste ragioni è il progetto legato a tecnologie esterne come *GWT* e *Servlet*. Oltre all'interfaccia utente, il progetto realizza anche la gestione degli utenti, adoperando il sistema IAS. Le operazioni realizzate adoperano la *neptune management library* per essere eseguite.

IV.1 Neptune management library

La *neptune management library (nml)* gestisce tutte le operazioni legate alle funzioni fondamentali di Neptune, ossia realizza per intero i livelli *physical machine manager* e *virtual cluster manager*, inoltre, realizza la parte di gestione dell'emulazione del livello *emulation manager*.

Allo stato attuale, il livello *pmm* non è realizzato, ad eccezione di piccole funzionalità essenziali ai livelli superiori, che sono comunque in fase iniziale di sviluppo. Questo impone l'uso di un'ipotesi di base per garantire il funzionamento del sistema: le macchine fisiche devono essere già configurate opportunamente, con l'installazione di un *hypervisor* supportato (per ora è supportato il solo *Xen*).

La *nml* è realizzata con un approccio altamente modulare, con una precisa strutturazione in *package*. Ciascuno dei livelli *pmm*, *vcm* ed *em* è realizzato in uno o più *package*. Oltre alle componenti strettamente legate a questi livelli, sono definiti anche dei *package* che sono comuni ai livelli stessi, ossia, sono adoperati da più livelli poiché realizzano funzionalità generiche o di interesse per l'intera libreria.

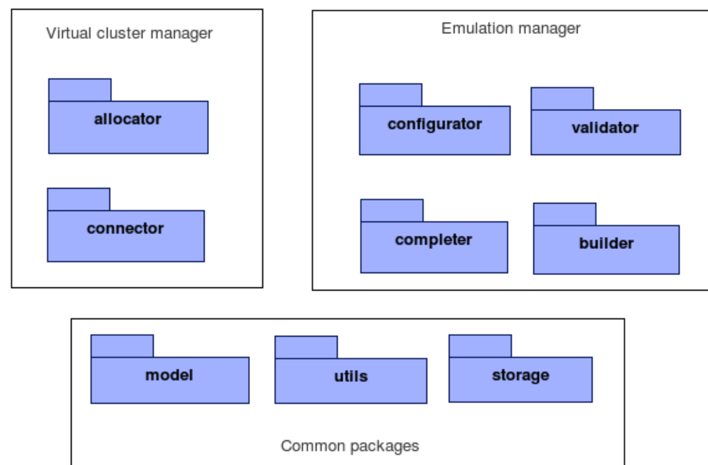


Figura IV.1: Organizzazione dei package della nml

IV.1.1 La definizione della struttura dei dati

Uno degli aspetti fondamentali della progettazione della *nml* è la definizione della struttura dati rappresentativa del modello dei dati. L'importanza di questo aspetto deriva dalla necessità di trattare tale struttura attraverso tutti i livelli dell'architettura, infatti, per quanto la struttura dati avrà chiaramente componenti legati in modo specifico ad alcuni livelli (ad esempio, un elemento *link* sarà certamente legato al livello *emulation manager*), la maggioranza degli elementi è invece adoperata nell'intera applicazione. Per procedere alla definizione in classi della struttura dati, si fa riferimento al modello dei dati definito nel capitolo 2.

La definizione della struttura dati che caratterizza una topologia è la prima ad essere condotta. Questa scelta potrebbe risultare non chiara, visto che la topologia fa capo al livello *emulation manager*, che nell'architettura del NIM è quello più dipendente dagli altri livelli, tuttavia, una topologia, come abbiamo visto nel capitolo 2, descrive larga parte dei dati che sono poi adoperati anche per altri scopi. Quindi è chiaro come la struttura dei dati della topologia include gran parte delle descrizioni necessarie a caratterizzare tutti i dati trattati dal sistema.

IV.1.1.1 Struttura dati della topologia

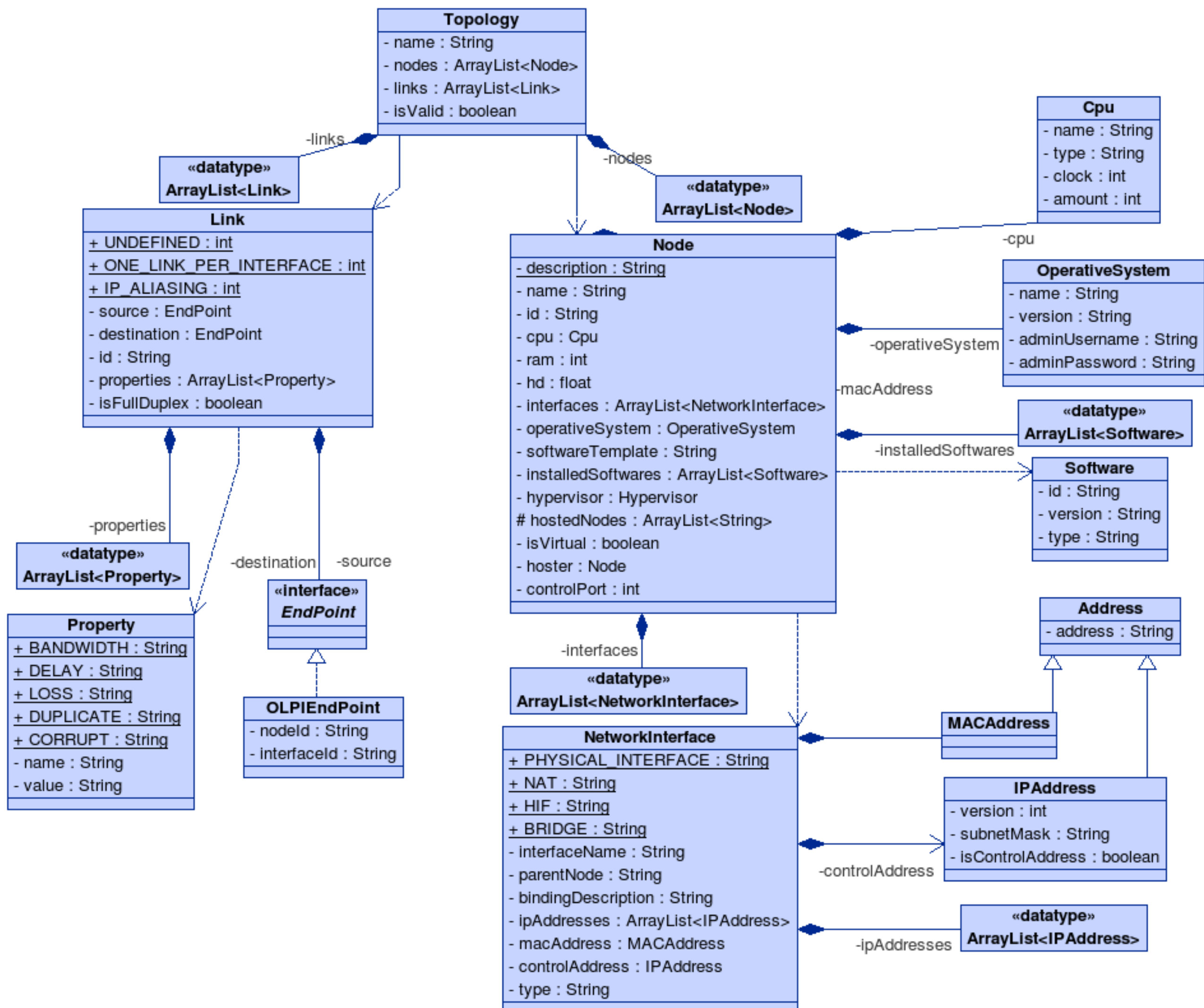


Figura IV.2: Struttura delle classi descrittive della topologia

La struttura dati descrittiva della topologia, riportata in Figura IV.2, segue fedelmente quanto definito nell'albero della topologia. Alcune cose sono da puntualizzare e chiarire:

- Dalla figura, si può notare la presenza di classi del tipo *ArrayList<Type>*, ad indicare collezioni di oggetti di tipo *type*. A livello concettuale, l'*ArrayList* non è altro che una lista *linkata*, ossia, un *array* dinamico. La scelta di adoperare questo contenitore per le collezioni di elementi è data dalla natura dinamica della struttura dati, che può essere modificata per elaborazioni più o meno complesse. In figura si

è preferito evidenziare la presenza di queste collezioni piuttosto che nasconderle dietro una semplice relazione UML di composizione;

- La classe *Node* presenta un oggetto di tipo *Hypervisor*, un oggetto di tipo *Node (hoster), ed* una collezione di stringhe che prende il nome di *hostedNodes*. Questo aspetto non è stato trattato direttamente nella descrizione dell'albero della topologia, ma è stato inserito per supportare la gestione di eventuali macchine virtuali residenti su di un nodo. Da questa definizione, potrebbe anche essere possibile realizzare su di un nodo virtuale ulteriori nodi virtuali, ossia effettuare un *nesting* delle macchine virtuali. Per gestire le macchine virtuali e le relazioni *padre-figlio*, si è introdotto l'oggetto *Node*, che è un puntatore all'*hoster* del nodo attuale, e la collezione *hostedNodes*, che riporta i nomi identificativi dei nodi residenti sul nodo attuale. L'oggetto *Hypervisor* descrive il tipo di *hypervisor* in esecuzione sul nodo, in modo che il sistema sappia come interagire con il nodo stesso per gestire le *vm* ospitate.
- La classe *NetworkInterface* presenta oltre ad una collezione di indirizzi IP (*ipAddresses*), anche un singolo indirizzo IP indicato come *control address*. Questo indirizzo è un semplice puntatore ad uno degli indirizzi definiti nella collezione *ipAddresses*, e rappresenta un modo rapido per verificare se un'interfaccia è di controllo e, in questo caso, accedere all'indirizzo di controllo. Da questa definizione della classe, è chiaro come sia contemplata la presenza di un unico indirizzo di controllo sull'interfaccia.
- La classe *OperativeSystem*, oltre a presentare tipo e versione del sistema operativo, contiene anche le informazioni per l'accesso come amministratore a tale sistema operativo.
- La classe *Link* contiene i puntatori agli *EndPoint* del link che rappresenta. Questi puntatori fanno riferimento ad una interfaccia *EndPoint*, che viene poi implementata per rappresentare l'opportuno tipo di *end point*. Per ora, l'unico *end*

point supportato è quello realizzato con la tecnica *one link per interface*, ed è rappresentato dalla classe *OLPIEndPoint*.

IV.1.1.2 Struttura dati dell'esperimento

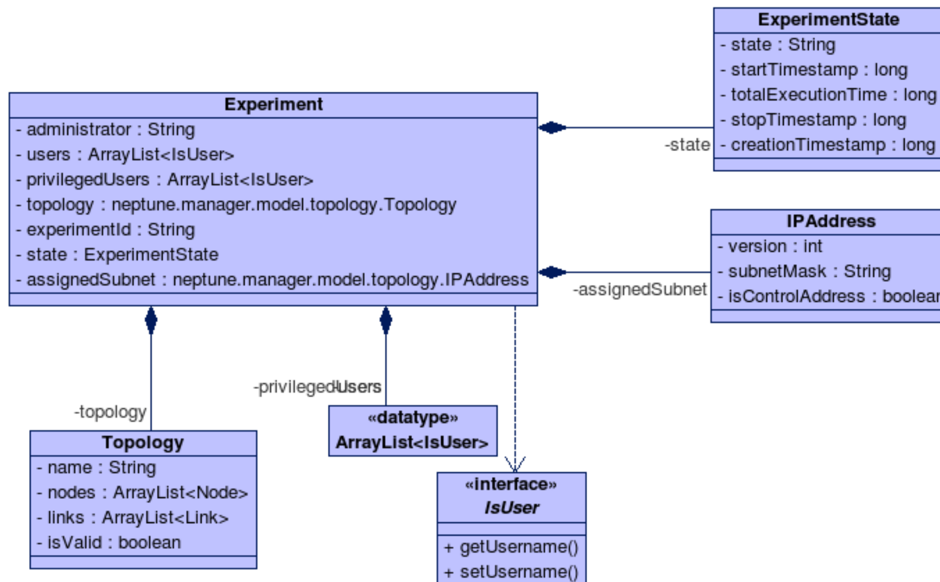


Figura IV.3: Struttura delle classi descrittive dell'esperimento

La realizzazioni in classi dell'esperimento è abbastanza lineare rispetto alla definizione del modello dei dati presentata nel Capitolo II . Sono da evidenziare soltanto due aspetti:

- Ad un esperimento è associato un *IPAddress*, che è l'indirizzo della *experiment subnet* dell'esperimento;
- Per rappresentare gli utenti che possono accedere all'esperimento o l'utente amministratore dello stesso, si adopera l'interfaccia *IsUser*. Questa interfaccia deve essere poi implementata dai componenti software che adoperano la *neptune management library*, al fine di gestire opportunamente gli utenti. Quindi, il concetto di utente qui presentato non ha lo scopo di gestire gli accessi, ma soltanto di rappresentare il modello dei dati dell'esperimento.

IV.1.1.3 Struttura dati del cluster

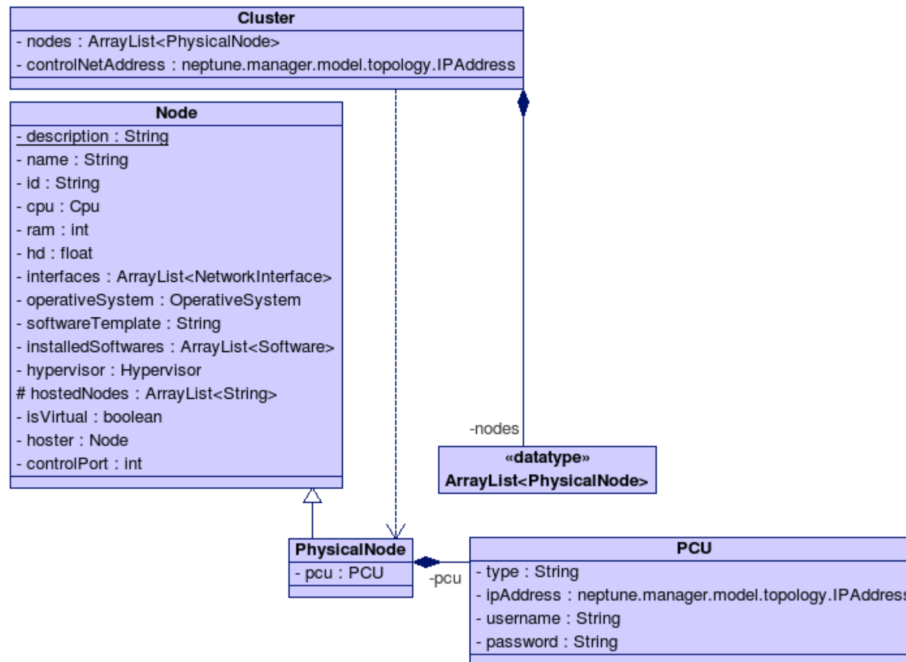


Figura IV.4: Modello delle classi descrittivo del cluster

Il cluster è descritto come una collezione di *PhysicalNode*, che sono delle classi che estendono la classe *Node*, introducendo semplicemente l'elemento PCU, come descritto nel modello dei dati presentato nel Capitolo II .

IV.1.2 Realizzazione del virtual cluster manager

Le funzionalità del livello *virtual cluster manager* sono la gestione e la creazione delle macchine virtuali che sono ospitate sulle macchine fisiche del cluster. Il livello deve quindi innanzitutto affrontare il problema della gestione della comunicazione con gli *hypervisor*, prima di poter effettivamente intervenire sulle macchine virtuali.

L'interazione con gli *hypervisor* è una funzionalità di non semplice realizzazione, poiché da un lato si devono risolvere le problematiche di comunicazione remota con l'*hypervisor*, dall'altro si devono affrontare i problemi di generalizzazione dell'interfaccia per supportare molteplici *hypervisor*.

Queste problematiche sono fortunatamente già state affrontante da una comunità *open source* che ha realizzato la libreria software *Libvirt* [38]. Libvirt si occupa di fornire

un'interfaccia di alto livello per la comunicazione con molteplici *hypervisor* (inizialmente la libreria supportava il solo *Xen*, ma nel tempo è stato esteso il supporto ad altri *hypervisor*), definendo opportune tecniche di descrizione delle macchine virtuali.

Il compito del *virtual cluster manager* è quindi integrare *libvirt* e costruire su questa base le funzionalità di allocazione e gestione dei nodi.

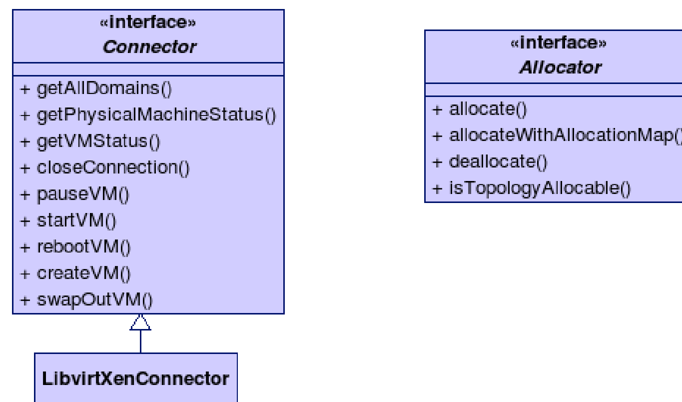


Figura IV.5: Modello delle classi rappresentative del vcm

L'interfaccia *Connector* incapsula le funzionalità di libvirt, disaccoppiandole da libvirt stesso, in modo da poter sostituire l'implementazione in caso di necessità, sia in fase operativa che in caso di necessità di *testing*. I metodi previsti, tutt'ora in fase di continua revisione ed ampliamento, sono quelli necessari a gestire il ciclo di vita di una macchina virtuale. Un *connector* è quindi un punto di accesso alle funzionalità di un singolo *hypervisor*.

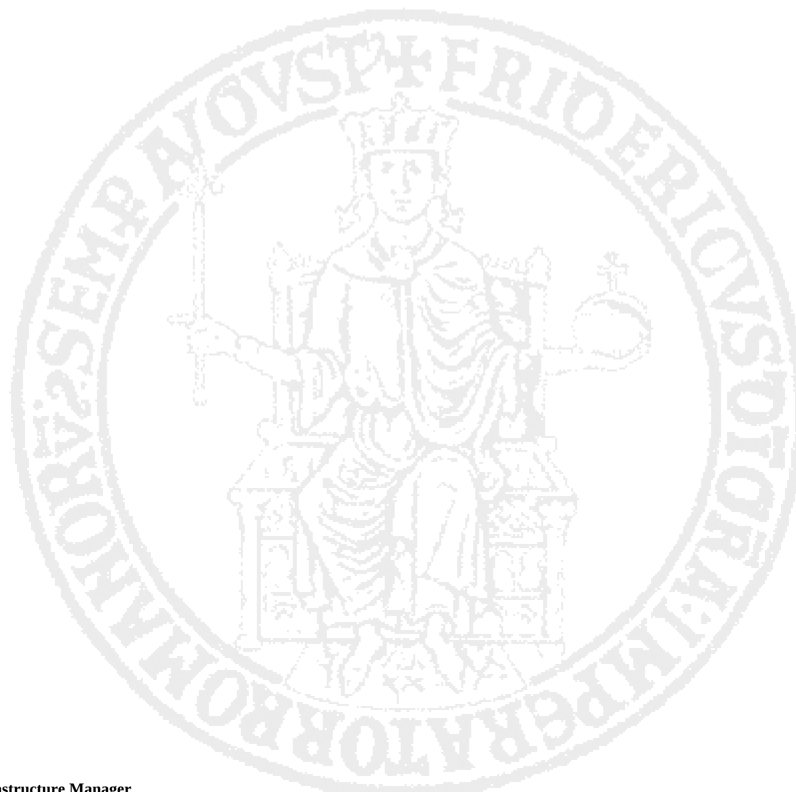
L'interfaccia *Allocator* è invece l'interfaccia che realizza le funzionalità di gestione dell'allocazione delle risorse, che ha una vista d'insieme sull'intero *cluster*. L'*allocator* prevede due tipologie di allocazione: *automatica* o tramite *allocation map*. Nel primo caso, lo specifico *allocator* (ossia una implementazione dell'interfaccia *Allocator*) realizza un algoritmo di allocazione automatica delle macchine virtuali sui nodi del cluster. Nel secondo caso, all'*allocator* è passata una struttura dati che contiene l'informazione di dove (su quale macchina del cluster) deve essere allocata ciascuna macchina virtuale.

Per compiere le operazioni di allocazione, l'*allocator* adopera una opportuna implementazione di un *connector*, definita a tempo di compilazione o dinamicamente, in base alle scelte di realizzazione del sistema.

E' importante sottolineare come la coerenza dell'allocazione non sia gestita a questo livello. In altre parole, non si verifica l'effettiva disponibilità delle risorse al momento dell'allocazione. Infatti, la non allocabilità genera opportune eccezioni che sono gestite dal sistema, ma solo quando si è già verificata l'impossibilità di allocare un nodo, ossia, quando il controllo è stato già passato all'opportuno *hypervisor*.

Il controllo sulla gestione corretta delle risorse (o meglio, il controllo che non si richiedano più risorse di quelle disponibili) è invece fatto a livello della struttura dati. La classe *Node* presenta infatti il metodo *addHostedNode(Node)* che, oltre ad aggiornare la lista *hostedNodes*, effettua dei controlli sulla possibilità effettiva di ospitare il nodo. Tali controlli sono a loro volta realizzati nel metodo *verifyHostingCapacity*, definito sempre nella classe *Node*.

IV.1.3 Realizzazione dell'emulation manager



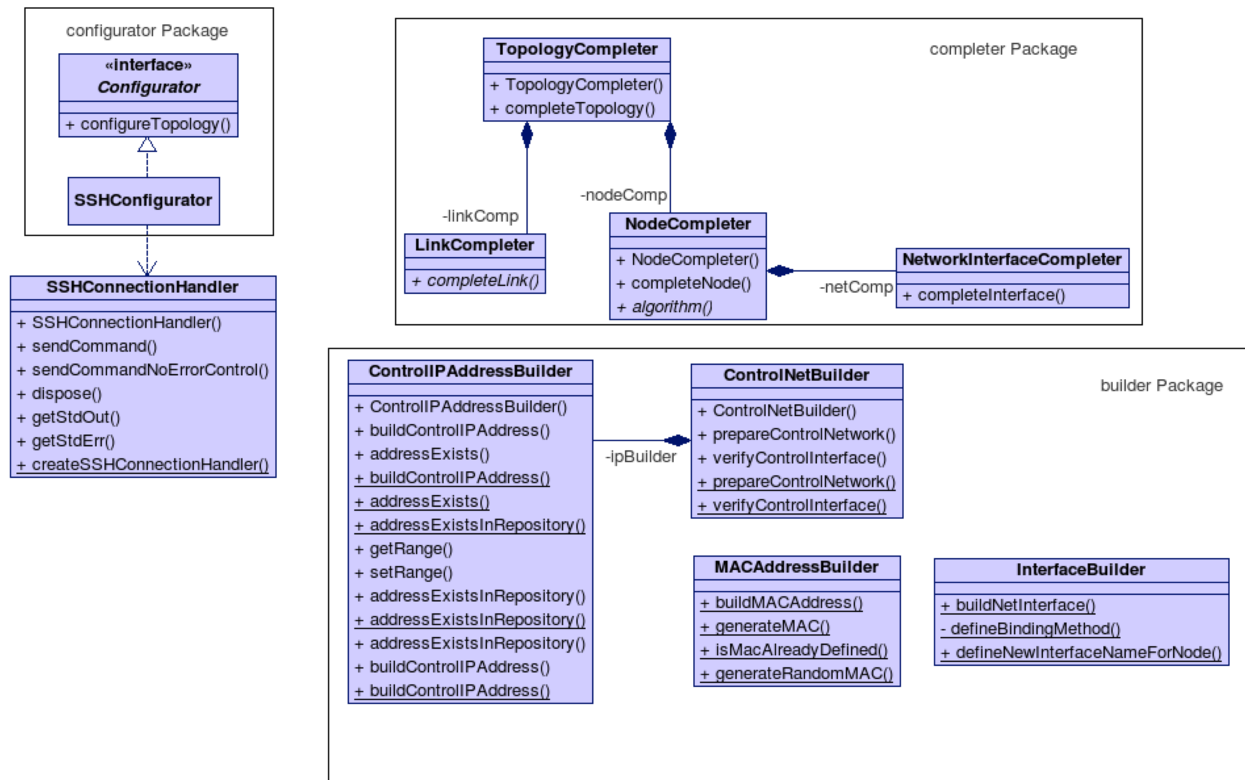


Figura IV.6: Modello delle classi che realizzano l'emulation mangaer

L'*emulation manager* è il componente che svolge, a livello concettuale, le azioni più complesse. Questa complessità delle operazioni svolte è dovuta all'alto livello di astrazione a cui opera l'*emulation manager*, che comporta l'interazione con i livelli sottostanti, per realizzare la logica di emulazione. L'operazione fondamentale svolta dai *package* di questo livello è quella di configurazione della topologia virtuale, partendo dal presupposto che i nodi della topologia siano tutti istanziati e raggiungibili.

A questa operazione basilare, tuttavia, si aggiungono un insieme di operazioni necessarie al corretto svolgimento della funzionalità. In particolare, sono da risolvere le problematiche legate al completamento di una topologia. Per completamento della topologia, come specificato nel Capitolo II , si intende la capacità del sistema di intervenire nella definizione di aspetti della topologia che non possono essere definiti dall'utente o che sono stati semplicemente omessi.

Con la funzione di completamento, il sistema consente all'utente di definire topologie ad un livello descrittivo puramente logico, occupandosi poi di inserire le informazioni necessarie perché la topologia possa essere istanziata sul cluster. Un esempio dell'applicazione di questa funzionalità è la definizione degli indirizzi IP della rete di controllo, che non richiede alcun intervento da parte dell'utente.

Nei prossimi paragrafi esamineremo inizialmente la sola funzionalità di configurazione e successivamente come il sistema gestisce il completamento della topologia.

IV.1.3.1 Topology enforcement

L'operazione di configurazione della topologia, a cui ci riferiremo col nome di *topology enforcement*, consiste nella configurazione delle macchine virtuali secondo quanto presentato nelle tecniche di *link multiplexing*. Questa operazione è infatti costituita sostanzialmente dalla configurazione dei link, e quindi, dalla realizzazione delle tecniche di *link multiplexing*, delle quali, attualmente, solo la tecnica OLPI è realizzata. Per realizzare la tecnica OLPI è necessario adoperare un insieme di comandi che fanno capo agli strumenti *route* e *tc*/*netem*. Questo presuppone la raggiungibilità del sistema operativo da remoto e la successiva capacità di inviare gli opportuni comandi. Il metodo di comunicazione adoperato è il protocollo *ssh*, che consente di ottenere una *shell* remota del sistema operativo. L'attivazione sul nodo da configurare di un server *ssh* raggiungibile tramite un opportuno indirizzo IP (un indirizzo di controllo definito sempre dal sistema) è realizzata tramite l'utilizzo di *template* di *vm* che contemplino la presenza del server *ssh* e tramite la configurazione dell'interfaccia di controllo che è, all'atto dell'allocazione del nodo, realizzata dal livello *vcm*.

Quindi, il livello *vcm* garantisce la presenza di un server *ssh* e di un indirizzo IP di controllo noto per una data macchina virtuale. Il livello *em* adopera queste informazioni per inviare gli opportuni comandi di configurazione dei nodi della topologia. Per l'invio dei comandi tramite *ssh*, è adoperata una ulteriore libreria *software open source*, che realizza lo *stack* protocollare *ssh*. Le funzioni della libreria sono incapsulate all'interno della classe *SSHConnectionHandler*.

IV.1.3.2 Completamento della topologia

Il completamento della topologia è eseguito sostanzialmente per l'assegnazione delle seguenti informazioni, nel caso in cui la topologia non le definisca:

- *indirizzi MAC:* vengono definiti gli indirizzi *MAC* per tutte le interfacce;
- *rete di controllo*: la topologia viene completata aggiungendo ad ogni nodo una interfaccia di controllo con gli opportuni indirizzi IP;
- *link subnet*: ad ogni link definito sulla topologia viene assegnata una *link subnet*, ed alle interfacce che realizzano gli end-point del link sono assegnati gli opportuni indirizzi IP.

Ciascuna di queste operazioni è più o meno articolata, ad esempio, la generazione degli indirizzi MAC è molto semplice, e richiede la sola realizzazione di un indirizzo MAC valido e univoco (appartenente al dominio dei MAC *privati*), mentre la realizzazione della *link subnet* deve tenere in conto aspetti quali la presenza di altre *link subnet*, gli indirizzi già definiti per un *end-point*, ecc.

L'aspetto fondamentale di questa operazione è che garantisce la possibilità di assegnare una stessa descrizione di topologia a più esperimenti, senza necessità di operare modifiche (se la topologia presenta le sole entità di definizione logica). Infatti, il sistema si occuperà di generare, per ogni esperimento, gli opportuni indirizzi fisici che vanno poi a realizzare effettivamente la topologia.

I *package completer* e *builder* realizzano proprio questa funzionalità. Il *package completer* contiene un insieme di classi, ciascuna con il compito di effettuare il completamento di una parte della topologia. Queste classi adoperano le funzionalità del *package builder* che contiene invece classi di utilità per la generazione delle informazioni.

IV.1.3.3 Validazione della topologia

Il processo di validazione della topologia consiste nella verifica della correttezza delle informazioni specificate. Tale correttezza assume significati differenti se la validazione è compiuta per verificare la struttura logica della topologia o la sua allocabilità. In particolare, la validazione per l'allocabilità presuppone che la topologia sia valida anche al

livello logico, quindi, questo tipo di validazione aggiunge ulteriori regole in aggiunta a quelle della validazione della struttura logica.

Il processo di validazione in se può variare molto velocemente al variare dei requisiti dell'applicazione o anche al variare della definizione del modello dei dati. Questa caratteristica, in aggiunta alla presenza di più tipi di validazione, ha reso necessaria la definizione di un apposito sistema che consentisse la gestione dell'intero processo in modo semplice e dinamico.

Il sistema è realizzato interamente all'interno del *package validator*, che contiene la definizione di due classi astratte, che caratterizzano l'intero modello di validazione: *ValidatorRule* e *Validator*.

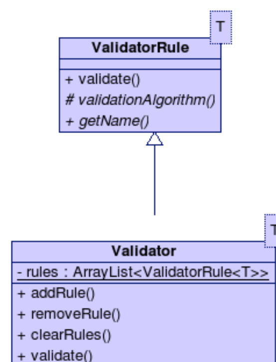


Figura IV.7: Struttura delle classi validator

La classe *ValidatorRule* contiene il metodo astratto *validationAlgorithm* che, nelle implementazioni concrete, conterrà la logica di validazione. La classe *Validator* eredita da *ValidatorRule*, quindi è a sua volta una *ValidatorRule*, ma può contenere ulteriori regole di validazione, che possono essere aggiunte tramite il metodo *addRule(ValidatorRule)*.

Il metodo *validate*, che nella classe *ValidatorRule* è una semplice applicazione dell'algoritmo definito in *validationAlgorithm*, nella classe *Validator*, dove viene ridefinito, oltre all'applicazione dell'algoritmo, si occupa di chiamare per ogni *ValidatorRule* contenuta nel *Validator* il corrispondente metodo *validate*. Questo approccio consente di definire strutture ad albero per l'applicazione delle regole di

validazione. L'approccio ad albero viene adoperato anche per il *report* che contiene il risultato della validazione.

Ogni *ValidatorRule*, infatti, a seguito della validazione, genera un oggetto *ValidatorResult*, che può contenere altri oggetti *ValidatorResult*. Questi oggetti sono gestiti automaticamente dalla struttura dei *Validator:* il metodo *validate* accetta in ingresso un *ValidatorResult,* il risultato della validazione viene conservato in un oggetto *ValidatorResult* che sarà incluso nel *ValidatorResult* passato come argomento al metodo *validate*.

Per garantire una semplice leggibilità del rapporto di validazione da parte dell'utente, è stato anche definito un sistema di *logging* “ad albero”, che è integrato negli oggetti *ValidatorResult*.

Adoperando questo sistema, sono state definite le regole ed i validatori per effettuare la validazione della topologia.

IV.1.4 La persistenza dei dati

Fino ad ora si è tralasciata la necessità di dover rendere persistenti le informazioni riguardanti topologie, esperimenti, template e cluster. La gestione della persistenza è stata realizzata in un opportuno *package* che prende il nome di *storage*.

Come anticipato, il formato di memorizzazione dati scelto è l'XML, per via della struttura gerarchica delle informazioni, ma soprattutto perché la gestione dell'XML non comporta l'aggiunta di software esterno al sistema (come ad esempio un DBMS). In ogni caso, tutte le classi *repository*, che incapsulano le funzioni per la persistenza, sono in realtà delle interfacce, di cui l'implementazione tramite XML è solo una soluzione che in un tempo futuro, se dovesse rendersi necessario, può essere sempre sostituita.

In realtà, la definizione del tipo di *repository* da adoperare è gestita tramite il *pattern abstract factory*. In questo modo, la sostituzione del tipo di implementazione del repository avviene semplicemente sostituendo il tipo di *factory*. (Ricordiamo che la *factory* è responsabile della creazione dell'oggetto e, quindi, le *factory* definite attualmente

generano oggetti delle classi che realizzano la persistenza tramite XML). La struttura appena descritta è presentata in Figura IV.4.

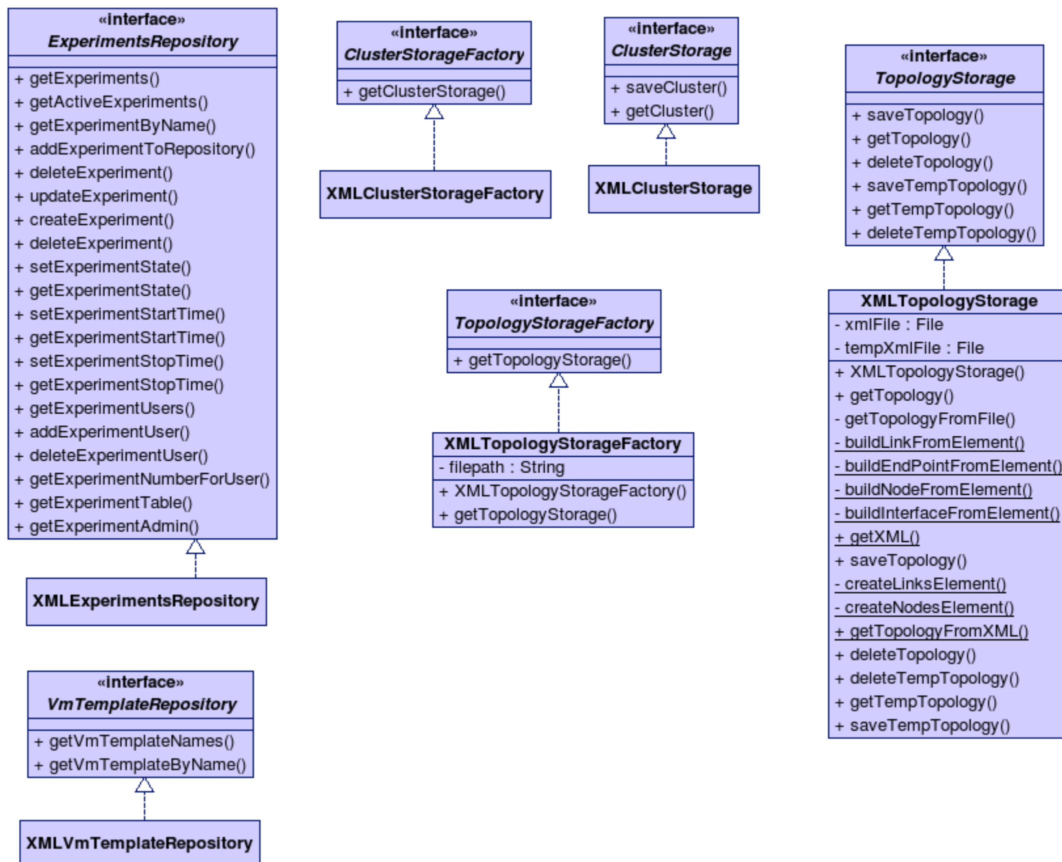


Figura IV.8: Il package storage

IV.2 Identification and Authorization System

L'IAS è il sistema che gestisce l'identificazione e l'autorizzazione degli utenti. Per usare una terminologia più generale, ci riferiremo agli utenti con il termine "principal" che indica una qualsiasi entità capace di compiere azioni.

Lo scopo dell'IAS è associare ai principal le operazioni che possono eseguire. Tale associazione è fatta per via indiretta attraverso i ruoli. In altre parole ogni *principal* è associato ad un ruolo. Ogni ruolo è associato a permessi corrispondenti a determinate operazioni.

Da questa rapida descrizione del problema si individuano tre entità: **Principal, Role, Operation**. Tali entità sono gestite dall'IAS con omonime classi:

- *Principal:* Il principal è una qualsiasi entità che può compiere operazioni. Le informazioni associate al principal sono un identificativo univoco ed il nome del ruolo cui il *principal* è associato. Questa classe ha lo scopo di incapsulare queste informazioni;
- *Role:* Il ruolo non è altro che un insieme di permessi, quindi, le informazioni ad esso associate sono un identificativo univoco e l'insieme di permessi concessi al ruolo. Anche in questo caso la classe ha semplicemente il compito di incapsulare queste informazioni;
- *Operation:* l'*operation* è l'entità più complessa da descrivere poiché ciascun dominio applicativo definisce operazioni proprie e specializzate. Per queste ragioni, esamineremo la classe Operation separatamente nel prossimo paragrafo.

IV.2.1 L'interfaccia Operation

Con la premessa fatta nel precedente paragrafo, è chiaro come sia necessario mantenere una generalità spinta nella descrizione dell'entità *Operation* e, quindi, si utilizza un'interfaccia al posto di una classe, per permettere che siano modellate tutte le possibili operazioni. La struttura dati dell'*Operation* ha una semplice informazione associata che è il nome identificativo dell'operazione, che permette di associare permessi all'operazione in questione. La vera peculiarità dell'*Operation* è che modellando un'operazione, è necessario fornire un modo univoco per richiamare l'esecuzione dell'operazione di cui l'*Operation* è astrazione. A tal proposito viene definito il metodo execute(). Osserviamo la realizzazione dell'interfaccia prima di fare ulteriori considerazioni:

```
public interface Operation {
        public String getID();
        public Object[] execute(Object[] args) throws IncorrectParamsException,
Exception;
        public String getDescription();
}
```

Esaminiamo i metodi dell'interfaccia:

- **getID()**: ha lo scopo di forzare lo sviluppatore ad inserire un identificativo all'operazione che sta definendo;
- **getDescription()**: è un metodo di supporto che viene proposto per consentire una maggiore comprensione di quello che fa l'operazione, definendo una descrizione testuale della stessa.

Esaminiamo nel dettaglio il metodo fondamentale dell'interfaccia: **execute().** Innanzitutto si può notare come il parametro di ingresso sia un vettore *Object[]*. Questa scelta è dettata dalla generalità dell'interfaccia. Utilizzando un vettore di *Object* è possibile passare qualsiasi tipo di parametro. La stessa scelta è stata fatta per i parametri di ritorno. Notiamo infine le eccezioni lanciate:

- **IncorrectParamsException**: Vista la generalità dei parametri passati, è possibile incorrere facilmente in errori di chiamata della funzione. A tal proposito, chi implementa il metodo, dovrebbe includere per prima cosa un controllo sui parametri di ingresso, al cui fallimento viene generata questa eccezione. In realtà, date le caratteristiche di Java, questo accorgimento sarebbe superfluo, tuttavia, la definizione di un'interfaccia generica può permettere l'uso di approcci alla programmazione differenti, e questa eccezione ha lo scopo di dar supporto a tali approcci;
- **Exception**: Analogamente ai parametri, un'operazione di business può generare eccezioni strettamente legate al suo dominio applicativo. Questa eccezione riassume tutte le possibili eccezioni che si potrebbero generare.

IV.2.2 La gestione dell'autenticazione

Come detto inizialmente, l'IAS deve verificare l'identità dei principal. Per farlo adopera il componente *PrincipalManager*, presentato nel precedente capitolo nell'architettura dell'IAS. Il *PrincipalManager* prende gli identificativi dei *principal*, le *password* a questi collegate, e li confronta con quelli da lui conservati in un *repository* gestito dal sistema. Se

i dati coincidono viene creato un oggetto *Principal* che viene restituito al richiedente dell'identificazione (Presumibilmente il *Presentation Layer*). Nella creazione di questo oggetto, il *PrincipalManager* ottiene anche il ruolo del *Principal* recuperandolo sempre dal *repository*.

Se ci dovessero essere problemi nell'identificazione, viene lanciata una eccezione *ImpossibleToIdentifyPrincipalException*.

Da notare la necessità di un *repository* per gli identificativi e le password. Il *repository* non è definito internamente all'IAS, ma ne è solo specificata l'interfaccia, che deve poi essere realizzata dal sistema che adopera l'IAS.

IV.2.3 Gestione delle operazioni

Il sistema di gestione delle operazioni realizzato dall'IAS prevede che le operazioni siano richieste all'interfaccia di accesso all'IAS che, verificando che il *principal* che le deve eseguire goda dei necessari permessi, le restituisce tramite l'interfaccia Operation.

Chiaramente l'IAS deve utilizzare un "magazzino" dove conserva tutte le operazioni definite. A tal proposito è definita la classe *OperationList*. Tale classe conserva tutte le operazioni presenti nel sistema, e le fornisce all'IAS quando questo le richiede per restituirle a sua volta, dopo aver fatto i dovuti controlli. La presenza della *OperationList* consente un approccio alla definizione delle operazioni supportate dall'IAS molto flessibile, è infatti possibile aggiungere o rimuovere dinamicamente le operazioni gestite.

Anche *OperationList* fa riferimento ad un *repository*, in questo caso viene però usato soltanto per esportare in modo umanamente leggibile le stringhe restituite dalle funzioni getID() e getDescription() definite in Operation. Il sistema che adopera l'IAS dovrebbe garantire un modo di accesso a tali informazioni a chi si occupa di compilare le *policy*.

IV.2.4 Esempio d'uso

Le operazioni protette da IAS devono essere definite implementando l'interfaccia *Operation*. Per fare un esempio definiremo l'operazione "Hello World":

```
public class HelloWorld implements Operation {

	public static String ID = "HelloWorld";

	public Object[] execute(Object[] args) throws IncorrectParamsException,
		Exception {

		System.out.println("Hello World!");
		return null;
	}

	public String getDescription() {
		return "Print 'Hello World'";
	}

	public String getID() {
	   return ID;
	}
}
```

A questo punto possiamo registrare l'operazione cosi' definita al gestore delle operazioni (che e' un oggetto *OperationList* ottenuto dall'IAS):

```
IAS ias = IAS.getIAS();
ias.getOperations.add(new HelloWorld());
```

Infine, possiamo adoperare IAS per autenticare un utente e per richiedere il permesso per eseguire un'operazione:

```
try{
    IAS ias = IAS.getIAS();

    Principal user = ias.getPrincipal("user","password");
    Operation operation = ias.getOperation(HelloWorld.ID, user);

    // The operation HelloWorld has no parameters defined,
    // so the argoument passed is just null
    operation.execute(null);

} catch (ImpossibleToIdentifyPrincipalException e){
    System.err.println("Principal authentication failed!");
} catch (UnauthorizedOperationException e){
    System.err.println("Unauthorized operation!");
} catch (RoleUndefinedException e){
    System.err.println("User role doesn't defined!");
}
```

L'esempio mostrato riporta un caso "operativo", poiché il sistema risulta già pronto ad essere adoperato. In realtà, per adoperare l'IAS, abbiamo visto che è necessario fornire le opportune informazioni sugli utenti e sui ruoli definiti nel sistema.

E' quindi richiesta la realizzazione esterna dei *repository* di utenti e ruoli. Questa realizzazione si concretizza semplicemente nell'implementare le interfacce *RoleRepository* e *PrincipalRepository*.

Entrambe le interfacce possono essere realizzate nel modo che più si adatta al sistema su cui e' adoperato l'IAS. Ad esempio, e' possibile realizzare i *repository* con file di testo, file XML, DBMS, ecc.

IV.3 Neptune Web Interface

Nel progetto *Neptune Web Interface* sono realizzate tutte le funzionalità riguardanti la gestione degli utenti collegata agli esperimenti e la realizzazione dell'interfaccia grafica. Mentre i precedenti progetti erano sostanzialmente dei componenti software autonomi, il *Neptune Web Interface* dipende strettamente dalla *Neptune Management Library* e dall'IAS.

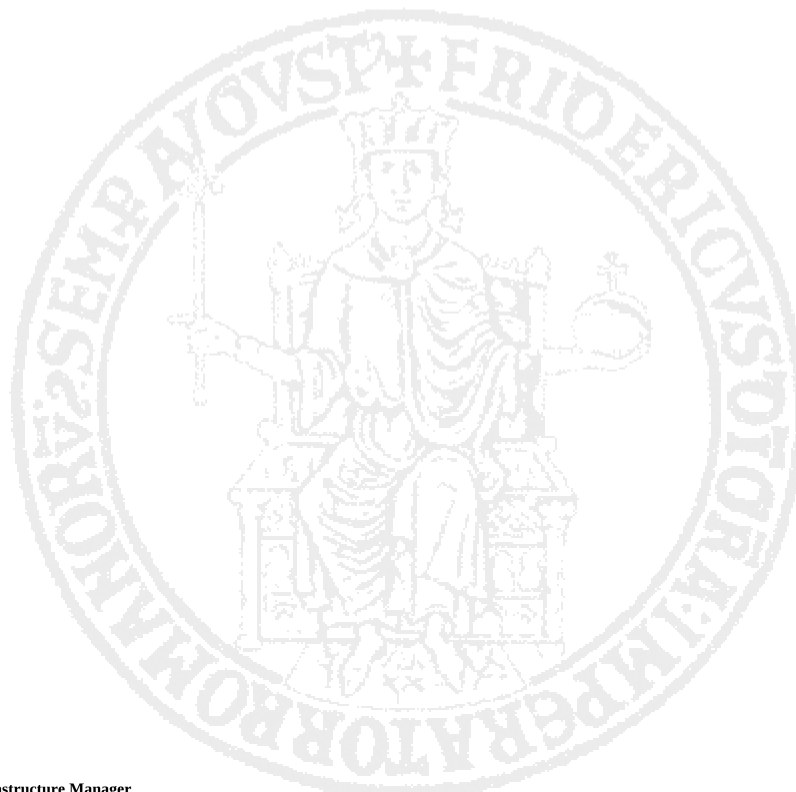
Neptune Web Interface è composto da due parti separate: il *client* ed il *server*. La parte *client* realizza l'interfaccia grafica e gestisce l'interazione con l'utente, mentre la parte *server* adopera le funzionalità della Neptune Management Library e dell'IAS per eseguire la logica del sistema. Prima di entrare nel dettaglio della descrizione delle due parti, chiariremo il sistema di comunicazione fra *client* e *server*, che adopera un meccanismo realizzato dal *framework* GWT.

Fino ad ora abbiamo solo anticipato che la comunicazione fra *client* e *server* avviene tramite il protocollo HTTP. Perché il sistema possa essere realmente adoperato è però necessario definire quanto meno un sistema di serializzazione dei dati, che consenta di trasferire le informazioni fra le due parti ed un modello di programmazione, che possa essere adoperato per implementare il sistema. GWT definisce questi aspetti e si occupa di realizzare per intero il sistema, in modo quasi del tutto trasparente all'utente. Sostanzialmente, viene adoperato un modello di chiamata a procedura remota (più propriamente una chiamata a metodo remoto) che viene implementato tramite la definizione di interfacce Java e la realizzazione di una particolare *servlet*.

Senza entrare eccessivamente nel dettaglio, il sistema prevede la definizione di una interfaccia classica Java, una cui implementazione generata da GWT rappresenta lo *stub client* del servizio realizzato sul *server*. Dal lato *server* il servizio è realizzato tramite una *servlet* che implementa a sua volta l'interfaccia. L'utente ha il compito di fornire un'opportuna implementazione ai metodi dell'interfaccia, mentre la gestione della comunicazione è realizzata anche in questo caso da uno *stub server* generato automaticamente da GWT. Il formato di serializzazione dei dati adopera il linguaggio JSON [39], che presenta notevoli vantaggi rispetto all'utilizzo di altre tecniche [28]. Nei prossimi paragrafi sono presentati il lato *client* ed il lato *server*.

IV.3.1 Lato client

Abbiamo visto che il lato *client* deve gestire sostanzialmente l'interfaccia utente. Seguendo l'approccio MVC, l'interfaccia è realizzata come un insieme di viste, ciascuna a sua volta formata da un insieme di componenti più o meno complessi. Per avere un metodo di accesso alle viste ben definito e per consentire una composizione anche complessa dell'interfaccia, le viste sono state organizzate in una struttura ad albero. In altre parole, si prevede la possibilità che alcuni tipi particolari di vista possano contenere a loro volta altre viste.



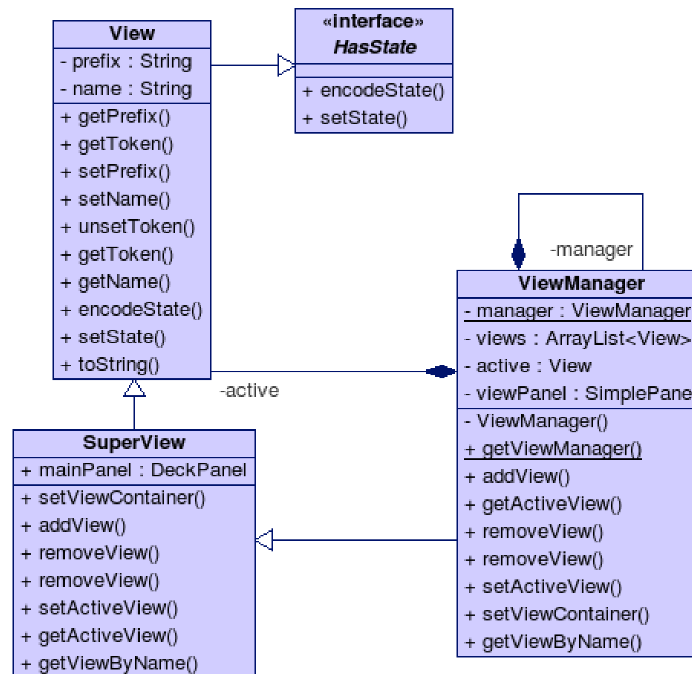


Figura IV.9: Struttura delle classi per la rappresentazione delle viste

In Figura IV.3 è presentata la struttura delle classi che realizza la gestione delle viste. La classe fondamentale è la classe *View*, che realizza una vista vera e propria. Una vista è caratterizzata da un nome che la individua in modo univoco. In aggiunta, ogni vista è descritta anche attraverso un prefisso. L'utilizzo del prefisso è necessario per gestire una gerarchia delle viste. In altre parole, il prefisso è costruito ricorsivamente in base alla vista che contiene la vista in questione. Il nome completo della vista, che è il nome identificativo della vista preceduto dal prefisso, prende il nome di *token* per ragioni legate alla gestione dell'attivazione delle viste. L'attivazione di una vista è infatti gestita tramite il sistema di *history token* presente nei *browser* (ricordiamo che il lato *client* è sviluppato in GWT e quindi eseguito in un *browser*). Gli *history token* vengono gestiti generalmente in modo automatico dal *browser*, per mantenere la funzionalità del comune tasto *back*, tuttavia è possibile modificare questa *history* aggiungendo *token* secondo una opportuna logica.

La gestione dell'attivazione delle viste adopera il sistema di *history* del *browser*. Ogni vista viene quindi attivata quando è inserito nella *history* del *browser* un *token* che la

identifica. Tale *token* è proprio quello ottenuto tramite il metodo *getToken()* della classe *View*, che restituisce il *token* della vista come definito poc'anzi. La gestione degli *history token* è possibile grazie ad alcune funzionalità di GWT che permettono non solo di aggiungere *token* alla *history*, ma anche di registrare dei *listener* per la stessa.

Questo sistema di attivazione delle viste non solo permette di gestire con semplicità le viste da attivare, ma permette anche di mantenere invariata la funzionalità del tasto *back* del *browser,* solitamente non adoperabile nelle applicazioni AJAX [40].

Dalla figura si può notare come la *View* implementi l'interfaccia *HasState*. Questa interfaccia indica che l'oggetto in questione ha uno stato che può essere serializzato in una stringa e che a partire da una stringa si può impostare lo stato dell'oggetto. Nel caso della *View* l'interfaccia *HasState* è implementata per fornire un sistema che consenta di gestire il salvataggio dello stato della *View* per necessità applicative. Si presuppone che tale stringa di stato sia aggiunta al *token* della *View*, in modo che si integri perfettamente nel sistema di gestione proposto.

Per garantire la possibilità di creare una gerarchia di viste, è stata realizzata la classe *SuperView* che estende la *View* ma aggiunge i metodi per inserire ed eliminare oggetti *View*. Questa classe gestisce per intero il sistema di attivazione della *View* a partire da un *token* attraverso il metodo *setActiveView()*. Allo stesso modo, alle *View* aggiunte alla *SuperView* è in automatico impostato il prefisso. La classe *ViewManager* ha lo scopo di fornire un punto di accesso centralizzato per a tutte le viste, è quindi un *Singleton* ed estende a sua volta la classe *SuperView*, poiché è un contenitore per tutte le altre viste.

IV.3.1.1 Widget e componenti grafici

La realizzazione dell'interfaccia grafica è stata fatta attraverso l'uso di componenti standard, detti *Widget* [41], che opportunamente composti permettono la realizzazione di interfacce anche complesse.

Alcune strutture di *Widget* create, dovendo essere utilizzate in più viste, sono state incapsulate in componenti grafici complessi che espongono una propria interfaccia. Questi

componenti, di cui fanno parte ad esempio tabelle con gestione della paginazione, *dialog* di inserimento dati, ecc., sono stati inclusi tutti nel *package components*.

In aggiunta a questi componenti, sono stati realizzati componenti non grafici ma comunque integrati nell'interfaccia utente, per effettuare un controllo sugli input immessi. Tali componenti, presenti nel *package view.input* sono degli *InputValidator*, ossia degli oggetti che presentano un metodo *validate(String)* che restituisce un valore booleano a seconda che la validazione sia o meno corretta. *InputValidator* è un'interfaccia che viene dunque implementata a seconda del tipo di validazione che deve essere compiuta. Alcuni *InputValidator* di esempio sono il validatore di indirizzi IP, di nomi utente, di email, ecc.

IV.3.1.2 Il problema della serializzazione

L'interfaccia grafica presenta alcune funzionalità, come la definizione grafica di una topologia, che richiedono il trattamento della struttura dati di una topologia per elaborazioni complesse. Questo comporta che l'intera struttura dati sia descritta in termini di classi anche all'interno del *browser*, ossia, che sia tradotta in oggetti *Javascript* per la fase di esecuzione.

Per questo motivo, l'intera struttura dati descrittiva della topologia, presente nella *neptune management library,* è replicata all'interno di *Neptune web interface*. Questo punto potrebbe generare confusione, quindi ribadiamo ancora una volta che le classi presenti nei package *client* della *neptune web interface*, anche se descritti con sintassi *java* vengono poi tradotti in *javascript*, mentre le classi della *neptune management library* in fase di esecuzione rimangono classi *java* e non possono quindi essere usate all'interno di un *browser* (ne tanto meno possono essere gestite dal processo di serializzazione di GWT per essere trasferite tramite HTTP).

L'utilizzo di una struttura dati *serializzabile* (che viene tradotta in *javascript*) e di una non *serializzabile* comporta che ogni volta che la struttura dati *serializzabile* deve essere gestita dalle funzioni della *neptune management library* debba essere trasformata nella struttura non *serializzabile*. Allo stesso modo, se la struttura dati recupera dalla *neptune management library* deve essere inviata al client per essere visualizzata, bisogna effettuare

una conversione da struttura non *serializzabile* a struttura *serializzabile*. Questi processi sono gestiti da un'apposita classe *(SerializzationUtils)* presente sul lato *server*.

IV.3.2 Lato Server

Il lato server si occupa di fare da collante fra l'interfaccia *web*, l'IAS e la *Neptune Management Library*. Lo scopo fondamentale di questo strato software è quello di gestire i comandi utente raccolti tramite l'interfaccia web, verificare se sono operazioni autorizzate tramite l'IAS e, in tal caso, eseguirle adoperando la *neptune management library*.

Il lato server è composto dai seguenti *package*:

- ***model***: contiene le classi che descrivono il modello dei dati necessario per *Neptune Web Interface*. Attualmente il solo *ApplicationUser* è definito (implementa l'interfaccia *Principal* di IAS e aggiunge informazioni quali l'e-mail utente);
- ***operations***: contiene le operazioni definite dall'applicazione che devono essere gestite da IAS, sono quindi tutte classi che implementano l'interfaccia *Operation*;
- ***repository***: contiene le classi e le interfacce che realizzano e descrivono i *repository* necessari per il funzionamento di *Neptune Web Interface* (ruoli, utenti, operazioni definite, configurazione);
- ***security***: contiene classi di utilità per la gestione delle funzionalità di IAS
- ***services***: contiene le classi che implementano le interfacce contenute nel corrispondente package *client*, ossia tutte le interfacce che rappresentano i servizi che sfruttano il meccanismo RPC di GWT.

Il lato *server* adopera un *Singleton* per gestire in modo centralizzato l'applicazione e per fornire un punto centrale di inizializzazione e caricamento delle configurazioni. Tale *Singleton* è un oggetto della classe *ApplicationManager*. L'oggetto in questione viene creato all'atto del primo *login* di un utente e rimane conservato in memoria per tutta l'esecuzione dell'applicazione. Al momento del caricamento, un *ConfigurationRepository* (Definito nel *package repository*) si occupa di caricare e rendere disponibile la configurazione. In base a queste informazioni sono avviati e caricati i *repository* di utenti,

ruoli e tutti gli oggetti necessari al funzionamento della *Neptune management library*. In questo procedimento viene inizializzato anche l'IAS.

L'oggetto della classe *ApplicationManager* è poi adoperato, nel corso della vita dell'applicazione, come punto di accesso per l'IAS e per la *neptune management library*.

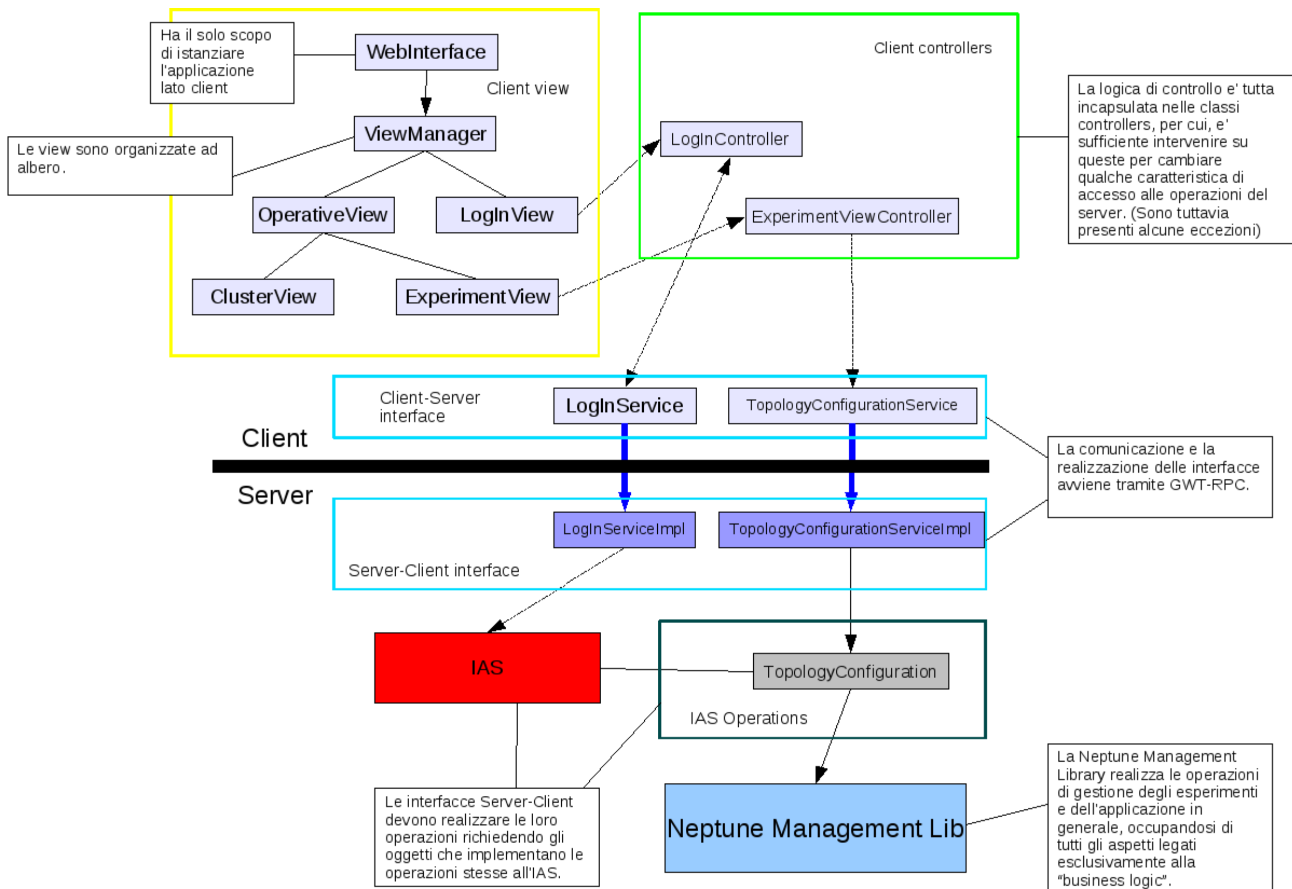


Figura IV.10: Schema riassuntivo (non completo) della Neptune Web Interface

In Figura IV.1 è presentato uno schema riassuntivo della *Neptune Web Interface*. Nella figura sono mostrate solo alcune delle classi componenti lo strato software. In particolare, è mostrata parte della struttura dell'albero delle viste, i corrispondenti *controller* che gestiscono gli input ricevuti da tali viste e gli stub lato *server* e lato *client* che permettono la comunicazione tramite HTTP. Dal lato *server* si evidenzia come gli *stub* adoperino i servizi dell'IAS per ottenere degli oggetti *Operation* che, adoperando la *neptune management library*, realizzano le funzionalità richieste dall'utente.

Conclusioni e sviluppi futuri

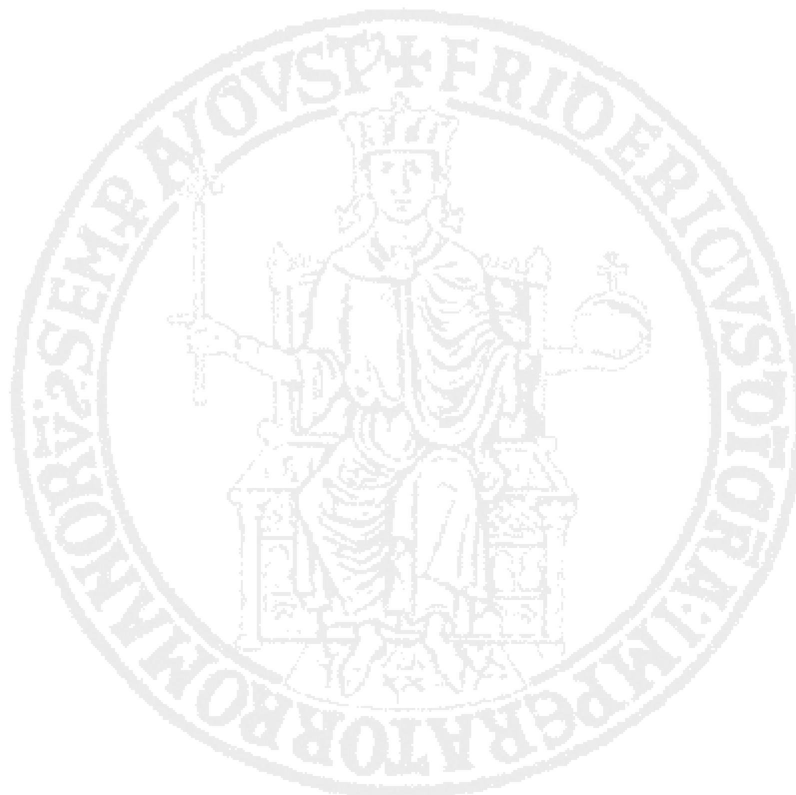
Il progetto Neptune presentato in questo lavoro di tesi è ad uno stadio iniziale di realizzazione. Questo lavoro ha analizzato, approfondito ed esteso i concetti alla base del progetto ed ha gettato le basi dell'architettura software del sistema. Attualmente Neptune è un sistema effettivamente funzionante e pronto all'uso, per quanto presenti solo un sotto-insieme delle funzionalità previste. Fra queste funzionalità sono presenti:

- gestione degli utenti in base ai ruoli;
- creazione e gestione degli esperimenti;
- creazione e modifica della topologia virtuale tramite editor visuale o tramite descrizione XML;
- allocazione e configurazione delle topologie virtuali;
- *monitoring* delle macchine costituenti il *cluster*.

A supporto dello sviluppo del progetto e delle funzionalità mancanti, Neptune ha assunto la licenza *software GPL* ed è presente nelle comunità *open source SourceForge* e *Google Code.*

I passi successivi nello sviluppo del progetto Neptune contemplano sicuramente l'implementazione delle funzionalità mancanti, quali la realizzazione del livello *physical machine manager*, per la gestione avanzata delle macchine fisiche del *cluster*, o il supporto alla gestione della riorganizzazione delle risorse del cluster, tramite la migrazione delle macchine virtuali. Inoltre, allo sviluppo del progetto Neptune così come immaginato fin da principio è possibile affiancare la possibilità di riorganizzare il progetto per dividere gli aspetti di gestione del *cluster virtuale* da quelli di emulazione, al fine di

dar supporto ad un'applicazione più generale ed adatta a più contesti. Quest'ultima ipotesi, anticipata già in precedenza in questo testo, è una evoluzione più che probabile del progetto in prospettiva futura.



Bibliografia

1: Wikipedia, Simulazione, 2008, http://it.wikipedia.org/wiki/Simulazione

2: Wikipedia, Network simulation, 2008, http://en.wikipedia.org/wiki//Network_simulation

3: Sally Floyd, Vern Paxson, Difficulties in Simulating the Internet AT&T, 2001

4: Wikipedia, Testbed, 2008, http://en.wikipedia.org/wiki/Test_bed

5: G. Ventre, Networking Day - L'evoluzione delle esigenze di connettività ad alte prestazioni[...], 2008

6: PlanetLab, PlanetLab, 2008, http://www.planet-lab.org/

7: S. Guruprasad, R. Ricci, J. Lepreau, Integrated Network Experimentation using Simulation and Emulation, 2005

8: Wikipedia, Computer cluster, 2008, http://en.wikipedia.org/wiki/Computer_cluster

9: Emulab, Emulab, 2008, http://emulab.net

10: Emulab, Emulab Tutorial, 2008, https://users.emulab.net/trac/emulab/wiki

11: G. Ventre, R. Canonico, P. Di Gennaro, V. Manetti, Network Emulation on Globus-based Grids: mechanisms and challenges, 2007

12: Wikipedia, Scalabilità, 2008, http://it.wikipedia.org/wiki/Scalabilit%C3%A0

13: G. Ventre, R. Canonico, P. Di Gennaro, V. Manetti, Virtualization Techniques in Network Emulation Systems, 2008

14: Wikipedia, Virtualizzazione, , http://en.wikipedia.org/wiki/Virtualization

15: A. Mauriello, Un sistema per la gestione e la distribuzione di cluster virtuali orientati all'emulazione delle reti, 2008

16: Wikipedia, Paravirtualization, 2008, http://en.wikipedia.org/wiki/Paravirtualization

17: Xen.org, Xen Hypervisor, , http://xen.org

18: K. Keahey, I. T. Foster, T. Freeman, X. Zhang, D. Galron, Virtualworkspaces in the grid, 2005

19: P. Di Gennaro, Neptune: un sistema integrato per la gestione e ildeployment di cluster virtuali orientato all'emulazione, 2008

20: Xen, Xen Networking, 2008

21: EbTables, EbTables, 2008, http://ebtables.sourceforge.net

22: Netfilter, Netfilter/IpTables, 2008, http://www.netfilter.org/

23: Linux.org, Introduction to Linux Traffic Control, 2008, http://www.linux.org/docs/ldp/howto/Traffic-Control-HOWTO/intro.html

24: A. Keller, tc Packet Filtering and netem, 2006

25: Linux foundation, Net: Netem, 2008, http://www.linuxfoundation.org/en/Net:Netem

26: R. Canonico, D. Emma, G. Ventre, An XML Description lenguage for Web-based Network simulation, 2004

27: D. Gelperin, Precise Use Cases, 2008, http://www.methodsandtools.com/archive/archive.php?id=8

28: R. Bifulco, Rich client web applications: the future so near, 2008, http://robertobifulco.it/?q=node/126

29: Mozilla, Same origin policy for JavaScript, 2008, https://developer.mozilla.org/En/Same_origin_policy_for_JavaScript

30: S. Burbeck, Applications Programming in Smalltalk-80(TM):How to use Model-View-Controller (MVC), 1987, http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html

31: Wikipedia, Model-View-Controller, 2008, http://it.wikipedia.org/wiki/Model-View-Controller

32: PureMvc, PureMVC framework, 2008, http://puremvc.org/

33: D. Gollmann, Computer security, 2006

34: E. Gamma, R. Helm, R. Johnson, J. Vissides, Design Patterns: Elements of Reusable Object-Oriented Software, 1995

35: R. Bifulco, Friendly User Authorization System, 2008, http://robertobifulco.it/?q=node/190

36: Google, Google Web Toolkit, 2008, http://code.google.com/webtoolkit/

37: R. Dewsbury, Google Web Toolkit Applications, 2008

38: Libvirt, Libvirt virtualization API, 2008, http://www.libvirt.org/

39: Json.org, Introducing JSON, 2008, http://www.json.org/

40: OpenAjax foundation, AJAX whitepapers, 2008, http://www.openajax.org/

41: Google, GWT Widget Gallery, 2008, http://code.google.com/docreader/#p=google-web-toolkit-doc-1-5&s=google-web-toolkit-doc-1-5&t=DevGuideWidgetGallery

42: Programmazione Web Lato Server – Della Mea, Di Gaspero, Scagnetto, 2007

43: AJAX per applicazioni web – Romagnoli, Salerno, Guidi, 2007

44: Open Web Application Security Project – http://www.owasp.org

45: The Same Origin Policy – http://www.mozilla.org/projects/security/components/same-origin.html

46: Ajax and Mash-up Security – http://www.openajax.org/whitepapers/Ajax%20and%20Mashup%20Security.html

47: Overcome security threats for Ajax applications – http://www.ibm.com/developerworks/library/x-ajaxsecurity.html

48: Javascript description – http://www.w3schools.com/js/default.asp

49: OpenAjax whitepapers – http://www.openajax.org/

50: XSS cheat sheet – http://ha.ckers.org/xss.html

51: JavaScript Hijacking – Chess, O'Neil, West, Fortify Software, 2007

52: Shaping the future of secure AJAX mashups – http://www-128.ibm.com/developerworks/library/x-securemashups/

53: *Safe JSON – Yates, 2007*

54: *Remote JSON - JSONP – http://bob.pythonmac.org/archives/2005/12/05/remote-json-jsonp/*

55: *Security for GWT Applications – http://groups.google.com/group/Google-Web-Toolkit/web/security-for-gwt-applications*

56: *JSON is not as safe as people think it is - http://getahead.org/blog/joe/2007/03/05/json_is_not_as_safe_as_people_think_it_is.html*

57: *Using GWT for JSON Mashups – Morril, 2007*

58: *AJAX and Mushup security – http://ajax.sys-con.com/read/436300.htm*

59: *Web Security – Roberto Bifulco, http://www.robertobifulco.it/PV/projects/Web%20Security/articles/Web%20Security.html*

60: *Computer Networks: a system approach – Peterson, Davie, 2007*

61: *Software + Services – The Architecture Journal 13, Microsoft, 2007*