UNIVERSITÀ DEGLI STUDI DI ROMA
TOR VERGATA

FACOLTÀ DI INGEGNERIA

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

A.A. 2013/2014

Tesi di Laurea

Un framework unificato per il monitoraggio di risorse e la migrazione di macchine virtuali in OpenStack

RELATORE

Prof.ssa Valeria Cardellini

CANDIDATO

Claudio Pupparo

Indice

Ringraziamenti

Innanzitutto ringraziare la professoressa Valeria Cardellini è quanto meno d'obbligo, per la grandissima disponibilità di questi mesi, durante i quali mi ha accolto in studio anche dopo esser stata assediata per ore da orde di studenti.
Un ringraziamento speciale alla mia famiglia; a mia madre, mio padre e mio fratello per avermi supportato e soprattutto sopportato in questi mesi di fuoco; ai miei zii e cugini, sempre in prima fila in caso di necessità.
Ed ora l'impresa ardua, riassumere in poche righe i ringraziamenti per tutti gli amici che mi hanno accompagnato in questi anni. Andrea, Gianluca e Pasquale, le giornate a base di nerding e "alimenti dall'alto contenuto calorico" (non vorrei traumatizzare il lettore) sono passate alla storia. Francesco aka "Pra", prima o poi lo realizziamo il nuovo Gothic. Christian, Damiano e le centinaia di ore sui progetti, tra il fomento per nuove tecnologie e le maledizioni per i bug dell'ultimo minuto. Il Gattone, di cui non ricordo assolutamente il nome, e il suo portatile che "va una bomba". Il grande Michele, che fin dalle superiori si sorbisce i miei deliri da "prodotto scalare". Aleffio, l'imbattibile pro del gruppo. I miei compagni di tesi e di deliri da laboratorio, Matteo e Rosario. Serena, e l'appartamento vicino Spizzala di cui sono immensamente invidioso. Il Gringo, Matteone e Daniele, sempre pronti per quattro chiacchere, che siano sull'ultimo videogame, serie TV o patch del kernel. Dario, e il pc che non ne vuole sapere di funzionare. Giorgio, e la strategia geniale della Lufthansa. I mitici

del “piano di sopra”, Francis, Valerion e Alessio, per i quali sono sempre stato “Das Claudien”. I “secchioni” che hanno finito da tempo, Daniele, Andrea, Alessandro e Paolo, con cui fra una chiaccherata e l’altra passano mesi, ma è sempre un gran piacere. E fuori dall’università come non nominare Luca, Monica e Veronica, e il classico “scommettiamo che il locale scelto da Claudio è chiuso!?”. Roberto, dall’animo tanto nerd quanto altruista. Il mio Team e le nuove amicizie uscite da InnovAction Lab, e i mesi in trincea dietro ai pitch. Grazie a tutti!

Abstract

Il Cloud Computing costituisce oramai da anni una grande rivoluzione nel mondo IT. Se in precedenza l'ingresso nel mercato richiedeva un cospicuo investimento iniziale nell'infrastruttura, il Cloud Computing ha reso disponibile a piccole e medie aziende, ma anche ai singoli utenti, la potenza virtualmente infinita dei Data Center. D'altra parte, la sua rapida diffusione comporta nuove sfide per i fornitori di questi servizi. La violazione degli accordi siglati dai provider con i clienti in termini di *Service Level Agreement*, comporta un grande danno economico, oltre che di immagine.

È quindi sorta la necessità di introdurre nuove tecniche per aumentare l'efficienza dell'infrastruttura gestita da un provider di livello IaaS. Fra queste è sicuramente importante citare la **migrazione di macchine virtuali** (*Virtual Machine*, *VM*), un processo per il trasferimento di una VM da un host ad un altro; a trasferimento completato, la VM nell'host sorgente può terminare e riprendere l'esecuzione nell'host destinazione. Il processo è totalmente trasparente al sistema operativo della VM, e può essere reso trasparente anche agli utenti delle applicazioni in esecuzione sulla macchina.

La migrazione di VM trova applicazioni in numerosi casi d'uso, come il bilanciamento del carico, la riduzione del consumo energetico, la risposta a failure e la manutenzione. L'introduzione di politiche di migrazione può ottimizzare la gestione di questi eventi, eliminando la necessità di interventi manuali.

In questo lavoro di tesi, ci siamo focalizzati sulla realizzazione di politiche di migrazione nell'ottica del consolidamento di macchine virtuali. Questa è una tecnica che consiste nel migrare le VM su un insieme più ristretto di host, ponendo i rimanenti host in stato di risparmio energetico, con evidenti benefici per lo IaaS provider.

Come piattaforma è stato scelto **OpenStack**, un progetto open source costituito da una collezione di strumenti per la gestione di una infrastruttura cloud. Il punto di forza di OpenStack è sicuramente la sua natura open source, che elimina le problematiche di *vendor lock-in* presenti in soluzioni proprietarie, quali ad esempio *Amazon Web Services* e *Microsoft Azure*. OpenStack è in rapida diffusione, basti pensare che importanti player quali *HP*, *Red Hat*, *Mirantis* e *IBM* offrono servizi di cloud computing basati su questa piattaforma.

Nell'ecosistema ufficiale di OpenStack non esiste un progetto per la gestione di politiche di migrazione di VM. Per supplire a questa mancanza, nel contesto di una tesi di dottorato nell'Università di Melbourne, è stato realizzato **OpenStack Neat** [7], il cui focus è in particolare il consolidamento di VM. OpenStack Neat raggiunge sicuramente dei risultati interessanti; in particolare, su una piattaforma di test costituita da quattro nodi, porta ad una riduzione del consumo energetico di ogni nodo del 25%-33%, con un impatto sulle performance limitato. Nonostante questi risultati, Neat presenta alcuni aspetti da migliorare. In particolare, le politiche di migrazione sono interamente basate sull'analisi di un'unica metrica, l'utilizzo della CPU. Inoltre, l'integrazione con OpenStack è migliorabile, in quanto per la raccolta di risorse viene utilizzata una soluzione ad hoc, anziché **Ceilometer**, il progetto ufficiale di OpenStack con queste funzionalità.

In questo lavoro di tesi è stato realizzato un framework unificato, a partire dall'integrazione delle funzionalità di Ceilometer in OpenStack Neat. Allo scopo, sono state

realizzate nuove politiche di migrazione, basate sull'uso di una combinazione lineare di metriche relative alle risorse di un Host. In particolare, viene considerata una combinazione dell'utilizzazione della CPU e della RAM.

Poiché Ceilometer non offre il supporto per la combinazione di molteplici metriche, è stato necessario introdurre delle importanti modifiche ai componenti dedicati alla raccolta e pubblicazione dei campioni di metriche. Durante la realizzazione di questa soluzione, è stato inoltre identificato un bug in Ceilometer, opportunamente segnalato alla community, e per cui è stato fornito supporto alla risoluzione [20].

In parallelo con la realizzazione del framework, si è proceduto ad una installazione completa di OpenStack. Per lo scopo è stato realizzato inizialmente un cloud interamente virtuale, basato su altri servizi di Cloud Computing quali *Amazon Web Services* e *Digital Ocean*. Terminate queste prove, i lavori sono proceduti su un cluster all'Università di Tor Vergata configurato con OpenStack, realizzato nel contesto di un altro lavoro di tesi [10].

Tramite una serie di test, basati su tracce di utilizzazione di CPU collezionate da VM di PlanetLab, è stato evidenziato il funzionamento del framework integrato e sono stati rilevati i parametri ottimali per garantire i migliori risultati possibili, con l'obiettivo del consolidamento di VM. I risultati dimostrano che l'utilizzo della soluzione realizzata comporta importanti benefici nella gestione di situazioni di overload ed underload di uno o più nodi dell'infrastruttura.

Il framework realizzato può essere agevolmente esteso per includere il monitoraggio di nuove metriche, e per realizzare nuove politiche di migrazione ancora più complesse.

Capitolo 1

Migrazione di Macchine Virtuali

La migrazione di macchine virtuali (**Virtual Machine - VM**) è un processo che prevede il trasferimento di una VM da un host ad un altro; a trasferimento completato, la VM nell'host sorgente può terminare e riprendere l'esecuzione nell'host destinazione. Il processo è totalmente trasparente al sistema operativo in esecuzione sulla VM (sistema operativo **guest**). Tramite determinate tecniche, approfondite nei paragrafi successivi, è possibile rendere la migrazione totalmente trasparente anche agli utenti delle applicazioni in esecuzione sulla VM. Allo scopo, è necessario non interrompere le connessioni di rete attive ed evitare un eccessivo degrado delle prestazioni.

1.1 Scenari d'utilizzo

La migrazione di VM trova applicazione in numerosi casi d'uso, di seguito presentati:

- *Bilanciamento del carico.* Quando un utente crea una nuova VM, un algoritmo di scheduling determina l'host che la ospiterà; per aumentare l'efficienza dell'infrastruttura, il carico viene equamente bilanciato fra i vari host. A seguito della creazione di nuove VM e distruzione di VM esistenti, questo bilanciamento può

venir meno; una possibile soluzione consiste nella migrazione di un sottoinsieme di VM, per ridistribuirle uniformemente sull'insieme degli host.

- *Consolidamento di VM.* Il consolidamento è una tecnica per affrontare il cosiddetto **server sprawl**, ossia l'impiego di un numero di nodi maggiore del necessario, con conseguente sottoutilizzo di ciascun nodo. Le VM possono essere migrate verso un sottoinsieme minore di host, ponendo i rimanenti host in stato di sospensione. In questo modo si riducono i consumi energetici e l'inquinamento generato dall'infrastruttura, e si massimizza l'utilizzazione degli host attivi.

- *Gestione di situazioni di sovraccarico/malfunzionamento.* Durante l'esecuzione, un host può trovarsi in condizioni di sovraccarico o di malfunzionamento; non essendo tali eventi del tutto predicibili, è necessario prendere dei provvedimenti al verificarsi degli stessi. Migrare tutte le VM in caso di malfunzionamento, o un sottoinsieme di VM in caso di sovraccarico, costituisce una soluzione al problema.

- *Manutenzione.* Periodicamente gli host di un data center devono essere arrestati per poter procedere con operazioni di manutenzione. Migrare le VM ospitate dai nodi in questione, garantisce la continuità di servizio alle applicazioni in esecuzione.

1.2 Confronto con migrazione di processi

Un'alternativa alla migrazione di macchine virtuali è costituita dalla **migrazione di processi** [11]. Sebbene l'operazione possa sembrare più semplice, in quanto coinvolge

uno o più processi anzichè l'intero sistema operativo, risulta essere più problematica. In particolare:

- Overhead nella gestione delle dipendenze. Un processo può dipendere da altri processi e queste relazioni devono essere identificate e mantenute nella migrazione. Inoltre, sul sistema operativo destinazione, deve essere presente la stessa configurazione di librerie per garantire la corretta esecuzione dei processi migrati.

- Mancanza di trasparenza. Nella migrazione di VM non è necessario conoscerne il contenuto; la VM viene migrata in un unico blocco. Nella migrazione di processi è necessario analizzare il rapporto del processo da migrare con gli altri componenti del sistema operativo, per cui la trasparenza viene meno.

Per le problematiche sopra riportate, risulta più semplice migrare una VM nella sua interezza.

1.3 Funzionamento

Il processo di migrazione consiste nel trasferimento dell'intero stato della VM da un host ad un altro. In particolare è necessario copiare sia lo storage che la RAM. Per evitare di trasferire l'intero storage, che può arrivare a pesare centinaia di GB, è possibile appoggiarsi ad uno **storage condiviso**, accedibile tramite un **file system distribuito** (come ad esempio il **Network File System**, *NFS*). In questo modo viene limitata la quantità di dati da migrare.
Dal punto di vista della *trasparenza dell'operazione*, la migrazione è classificabile in

- **Cold Migration**. La VM viene interrotta, migrata e l'esecuzione ripresa sull' host destinazione a migrazione completata. Sebbene questo semplifichi note-

volmente le operazioni di migrazione, il *tempo di downtime* risulta percepibile dall'utente: durante l'intero processo la VM non è raggiungibile. Nel caso di cold migration, l'utilizzo di uno storage condiviso non è obbligatorio, ma comunque utile per ridurre il tempo di downtime.

- **Hot Migration** (o **Live Migration**). La migrazione avviene in maniera totalmente trasparente all'utente, che non percepisce il tempo di downtime. Questo approccio è sicuramente più complesso del precedente, in quanto non è possibile copiare l'intero stato della VM in un unico passaggio, ma è necessario continuare a copiare le pagine di memoria modificate dalla VM ancora in esecuzione; inoltre il tasso di scrittura su memoria deve essere tenuto sotto controllo, ed eventualmente limitato, per garantire il completamento della migrazione. Per garantire la trasparenza dell'operazione, è necessario migrare anche le connessioni attive. A differenza della cold migration, uno storage condiviso rappresenta un requisito fondamentale.

La migrazione della memoria può essere completata secondo i seguenti approcci [6]:

- **Suspend and Copy**. La VM viene interrotta, il suo stato migrato, e l'esecuzione ripresa nell'host destinazione. Rientra nella tipologia di cold migration.
- **Pre Copy**. La memoria è trasferita per intero prima di avviare la VM destinazione. Poichè nel frattempo la VM è in esecuzione sull'host sorgente, alcune pagine di memoria avranno subito modifiche, per cui è necessario riportare queste modifiche sulla destinazione. Quando lo stato rimanente è considerato sufficientemente "piccolo", viene interrotta l'esecuzione della VM sorgente, copiate le restanti pagine di memoria e ripresa l'esecuzione sull'host destinazione. È una tecnica di hot migration.

- **Post Copy**. Durante l'esecuzione della VM, viene copiata una parte della memoria, in modo tale da poter avviare la VM destinazione. Durante l'esecuzione di quest'ultima, è possibile che un processo provi ad accedere ad un'area di memoria non ancora trasferita; in tal caso viene generata un **Page Fault**, che viene inviato alla VM sorgente, da dove viene prelevata l'area di memoria richiesta. Costituisce una tecnica di hot migration.

È bene sottolineare come non tutti gli hypervisor supportino il processo di migrazione. Per un elenco delle feature supportate dagli hypervisor più comuni, è possibile fare riferimento a [17].

1.4 Migrazione in WAN

La migrazione di macchine virtuali ha differenti requisiti a seconda del tipo di hypervisor utilizzato. Uno dei requisiti comuni più stringenti è rappresentato dalla necessità di avere host sorgente e host destinazione nella stessa sottorete; in questo caso, la migrazione è quindi limitata alla rete locale.
Non sono molti gli hypervisor con supporto alla **migrazione in WAN**, come ad esempio *HyperV* [18] e *VMware* [19].
La migrazione in WAN comporta le seguenti problematiche:

- Necessità di migrare non solo la RAM, ma anche l'intero storage. In LAN la migrazione può essere limitata alla RAM tramite l'utilizzo di uno storage condiviso. In LAN i tempi di accesso allo storage risultano accettabili, mentre in WAN la larghezza di banda costituisce un forte collo di bottiglia.

- Necessità di riconfigurare la rete al termine della migrazione. Una volta migrata, la VM possiede l'IP della subnet sorgente, non compatibile con la subnet

destinazione. Di conseguenza vi è la necessità di ottenere un nuovo IP, con conseguente terminazione delle connessioni TCP aperte.

D'altro canto la possibilità di migrare in un'altra sottorete è una caratteristica utile in numerosi scenari d'utilizzo:

- *Disaster Recovery.* Migrare tutte le VM da un data center ad un altro, possibilmente localizzato in un'altra area geografica, in risposta ad un disastro ambientale.

- *Cloud Bursting.* Un modello di deployment delle applicazioni, in cui un'applicazione in esecuzione in un cloud privato o in un data center, viene eseguita in un cloud pubblico durante i picchi di traffico. In assenza di migrazione in WAN, tale tecnica è applicabile tramite la replicazione delle applicazioni, limitandone di fatto l'utilizzo ad applicazioni stateless o con elaborati meccanismi di sincronizzazione.

- *Consolidation.* Un'azienda che offre servizi distribuiti su diversi data center, può affrontare il problema di sottoutilizzo dell'infrastruttura. Il consolidamento delle risorse è un approccio che tenta di aumentare l'efficienza dell'infrastruttura, riducendo il numero di host attivi e aumentandone l'utilizzazione. A seguito di questo approccio, l'azienda in questione potrebbe trasferire le risorse in pochi grandi data center. Tale operazione richiede il trasferimento delle varie applicazioni in esecuzione sui server. In assenza di migrazione in WAN, queste operazioni di trasferimento non sono trasparenti all'utente e il downtime dell'intero procedimento può consistere in diversi giorni.

- *Follow the Sun.* È una strategia utilizzata da team che collaborano sullo stesso progetto in diverse aree geografiche. I team durante le ore lavorative devono

avere accesso a bassa latenza ai servizi del progetto. Una soluzione praticabile è nuovamente la replicazione delle applicazioni, che come già visto per il Cloud Bursting, ne limita le possibilità di applicazione.

- *Organizzazione in sottoreti degli host in un data center.* Per garantire maggiore flessibilità all'infrastruttura, le centinaia di host in un data center possono essere organizzate in differenti sottoreti. La presenza di un limite sull'insieme di host verso cui è possibile migrare una VM, ne limita seriamente l'efficacia.

1.4.1 Trasferimento dello storage

A seguire sono descritti una serie di approcci per la risoluzione del problema del trasferimento della storage su WAN.

Shared Storage

Usando uno Storage Condiviso, la VM sorgente e la VM destinazione condividono lo storage, e il trasferimento è limitato alla sola RAM, sensibilmente inferiore in dimensioni rispetto allo storage.

Il problema di questa soluzione consiste nel fatto che i tempi di accesso allo storage remoto sono sensibilmente più alti rispetto a quelli di accesso in locale, per cui le performance delle applicazioni risultano degradate e i vantaggi della migrazione vanificati.

On Demand Fetching

Soluzione analizzata in [14]. Lo storage non viene completamente migrato nella destinazione. Ad ogni accesso della VM destinazione ad un blocco di dati non presente sulla destinazione, viene inviata una richiesta di prelevamento degli stessi dalla sorgente.

In questa soluzione, la larghezza di banda risulta essere un possibile collo di bottiglia

per le operazioni di I/O, per un periodo di tempo continuato successivo all'avvio della VM destinazione. Inoltre, la stessa VM destinazione rimane dipendente dalla VM sorgente per il recupero dei dati. Di conseguenza in caso di Disaster Recovery, in cui il funzionamento della sorgente non è garantito, la VM destinazione cesserebbe il funzionamento.

Pre Copying - Write Throttling

In [2] viene proposta una soluzione basata su *pre copying* e *write throttling*. È stata realizzata come parte di **XenoServer**, un progetto per la costruzione di un'infrastruttura pubblica di computazione distribuita [1], per cui prevede che sulla sorgente e la destinazione sia in esecuzione l'hypervisor *Xen*. È importate sottolineare come lo storage di una VM sia basato su un template, su cui vengono effettuate successive modifiche durante la vita della VM. Di conseguenza, non è necessario trasferire l'intero storage, ma unicamente le modifiche effettuate sul template di base.

Il processo di migrazione è suddiviso in diverse fasi.

Durante la fase di **initialisation**, il *migration client* sulla sorgente contatta il *migration daemon* sulla destinazione, stabilendo l'inizio del processo di migrazione.

Viene quindi avviata la fase di **bulk transfer**, in cui mentre la VM è ancora in esecuzione, lo storage viene copiato e migrato sulla destinazione. Tutte le modifiche successive allo storage, vengono inserite in una coda di *delta*, per una successiva applicazione.

Terminata la fase precedente, viene invocata la funzione di Xen per la migrazione incrementale della RAM (fase di *Xen migration*). Nel caso in cui il tasso di scrittura superi una soglia prefissata, si utilizza un meccanismo di *write throttling*, per cui la scrittura viene rallentata.

Durante la migrazione della RAM, comincia in parallelo la fase di *delta application*,

in cui vengono applicati alla destinazione i delta inseriti in coda.

Per completare la migrazione, la VM sorgente viene interrotta e le rimanenti modifiche copiate in blocco. Questa soluzione non presenta le stesse problematiche dello *Storage Condiviso* e di *On Demand Fetching*, e presenta un downtime inferiore rispetto al semplice *Suspend and Copy*.

In Fig. 1.1 è mostrato uno schema dell'intero processo.

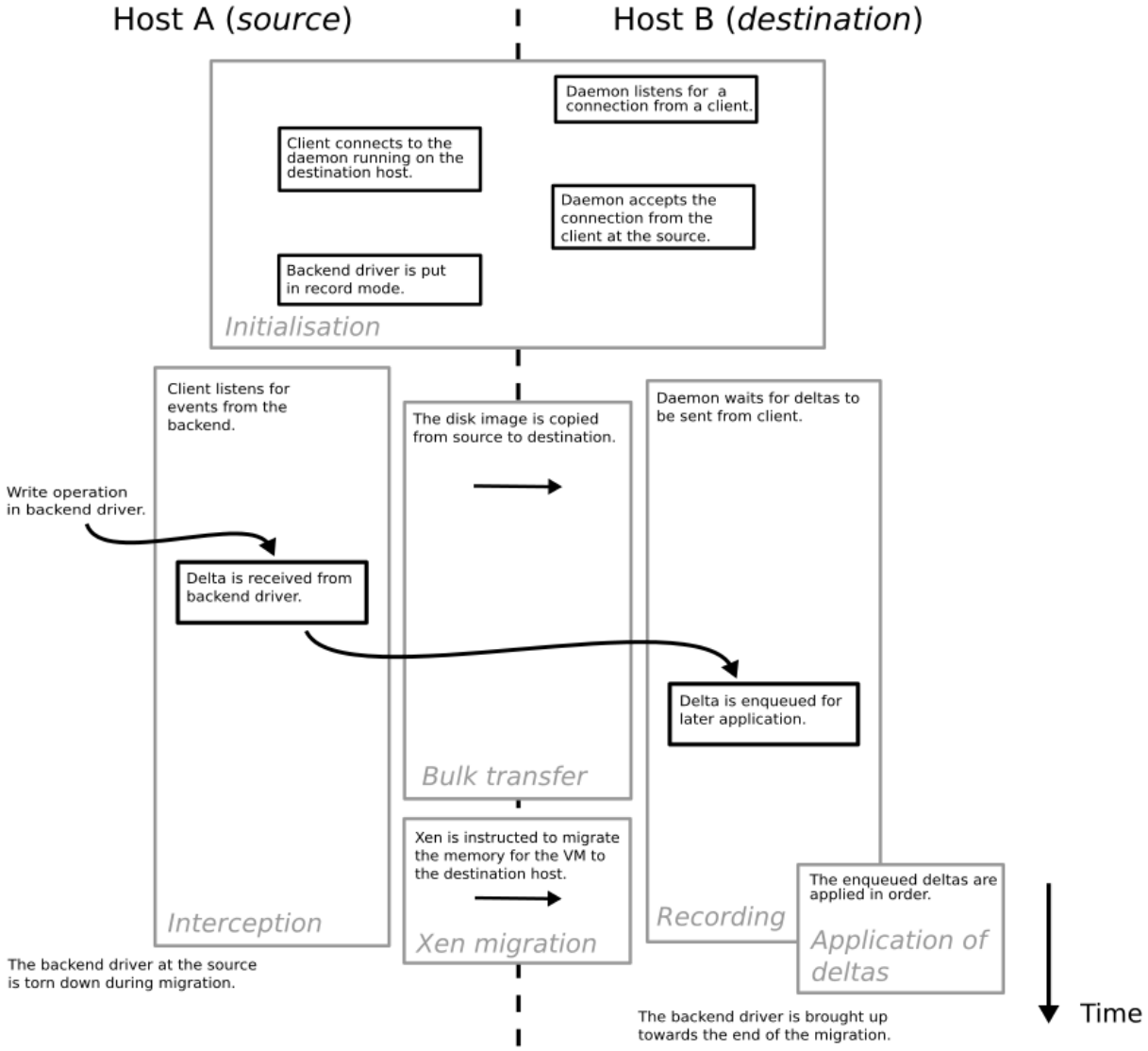

Figura 1.1: Schema del processo di migrazione basato su *Pre Copying* e *Write Throttling*

1.4.2 Mantenimento delle connessione attive

Relativamente al problema dell'interruzione delle connessioni attive al termine di una migrazione in WAN, in letteratura sono state proposte le seguenti soluzioni.

Tunnelling - Dynamic DNS

In [2] è proposta una soluzione basata su *Tunnelling* e *Dynamic DNS*. Questa soluzione prevede un cambio di IP della VM migrata.

Poco prima di interrompere la VM per completare la migrazione, viene instaurato un tunnel tra la VM con il vecchio IP sull'host sorgente (VM A) e la VM con il nuovo IP sull'host destinazione (VM B). A migrazione completata, quando la VM B è totalmente configurata, tramite *Dynamic DNS* viene aggiornato l'indirizzo IP del servizio fornito dalla VM A; in questo modo le nuove richieste verranno inviate alla VM B. Nel frattempo, le richieste che arrivano alla VM A, vengono redirette tramite il tunnel alla VM B. Per rispondere a richieste di entrambi i tipi, la VM B è configurata con due indirizzi IP.

Tale soluzione presuppone un accoppiamente fra sorgente e destinazione anche al termine della migrazione, fino alla chiusura delle connessioni con il vecchio IP. Questo accoppiamento rappresenta un problema nel caso di *Disaster Recovery*, come già precedentemente descritto.

Routing

In [3] viene proposta una soluzione basata su **OpenFlow** [29]. OpenFlow è un protocollo che permette la realizzazione di **Software Defined Network** [12], un approccio al networking in cui vi è un disaccoppiamento fra il *Control Plane*, lo strato che prende decisioni su dove inviare i pacchetti, e il *Data Plane*, lo strato che applica le regole di forwarding sui pacchetti.

In questa soluzione, la VM migrata non ha necessità di cambiare indirizzo IP.

Tramite OpenFlow è possibile modificare le **forwarding table** di uno switch, in cui sono definite le regole di inoltro dei pacchetti. In particolare, al completamento della migrazione di una VM da un host nella sottorete A ad un host nella sottorete B, gli switch delle due sottoreti devono essere configurati nel seguente modo:

- Lo switch in A deve inviare inviare allo switch in B i pacchetti aventi come destinazione l'IP della VM migrata (avente subnet A). Questa regola è necessaria per gestire connessioni instaurate precedentemente alla migrazione.
- Lo switch in B deve accettare pacchetti aventi come destinazione l'IP della VM migrata, sebbene questo IP appartenga alla sottorete A.

Le problematiche di questa soluzione sono le medesime di [2], in quanto presuppone una collaborazione fra sottorete sorgente e sottorete destinazione anche al termine della migrazione, potenzialmente dannosa in caso di *Disaster Recovery*.

Layer 2 Expansion

Un altro tipo di approccio consiste nell'estendere geograficamente una rete di livello 2, ponendo quindi sulla stessa sottorete diversi data center. In questo modo il problema di migrare una VM in una diversa sottorete, viene ricondotto al problema di migrazione nella stessa sottorete.

Una soluzione basata su questo approccio è stata proposta in [4] ed approfondita in [13]. In questo lavoro, gli autori propongono **CloudNet**, una piattaforma basata su **Virtual Private LAN Services** (*VPLS*), ossia un tipologia di **Virtual Private Network** (*VPN*) che sfrutta **Multi-Protocol Label Switching** (*MPLS*).

Una volta migrata, la VM può mantenere l'IP originale in quanto la migrazione avviene all'interno della stessa sottorete virtuale, per cui le connessioni TCP rimangono aperte. Poiché la collaborazione fra sorgente e destinazione termina a migrazione con-

clusa, non sono presenti le problematiche di [2] e [3], in caso di *Disaster Recovery*.

Isolation Boundary

In [5] il problema è stato affrontato da un punto di vista differente. La necessità di dover riconfigurare la rete di una VM a seguito di una migrazione in una sottorete differente, è una conseguenza della stretta correlazione che esiste fra la connessioni tra due host e gli IP degli stessi host; infatti una connessione è identificata dalla tupla {*IP_sorgente, IP_destinazione, Porta_sorgente, Porta_destinazione*}. L'obiettivo è quindi quello di disaccoppiare connessioni ed IP.

Viene di conseguenza proposta la creazione di uno strato intermedio fra livello applicativo e livello di trasporto, definito **Isolation Boundary**. È inoltre introdotto il concetto di **Transport Indipendent Flow** (*TIF*), ossia un flusso di dati indipendente da TCP. Quando a seguito di una migrazione, l'indirizzo IP della VM migrata cambia, per cui le connessioni TCP vengono chiuse, i due endpoint continuano a comunicare tramite il TIF, grazie all'aggiornamento del mapping tra TIF e la nuova connessione TCP associata al nuovo IP, come mostrato in Fig. 1.2.

La soluzione proposta in [5] non presenta problematiche dal punto di vista del *Disa-*

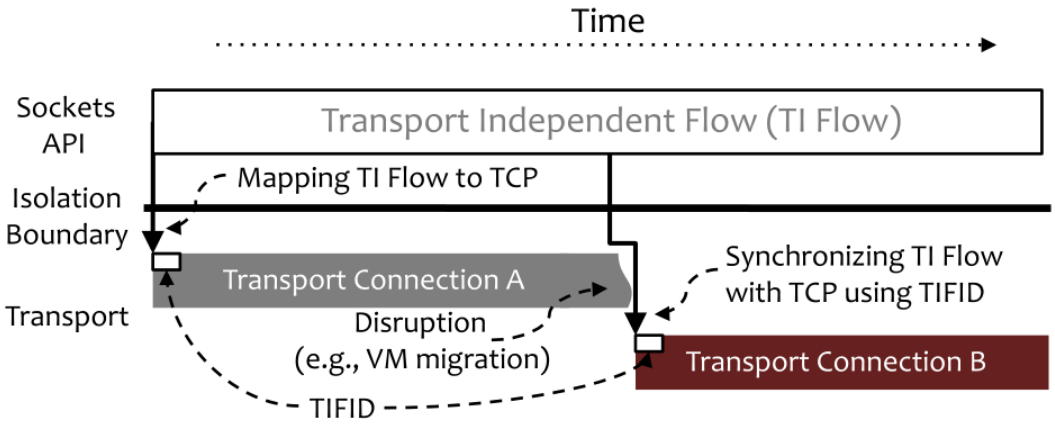

Figura 1.2: Mapping fra *Transport Independent Flow* e TCP

ster Recovery come in [2] e [3], né richiede la configurazione di una VPN come in [4]. Affinché il cambio di IP risulti trasparente, client e server devono essere consapevoli di questo strato intermedio, per cui è necessario applicare una patch al kernel dei rispettivi sistemi operativi. In caso contrario, essendo la soluzione proposta retro-compatibile, la migrazione può essere completata, ma la riconfigurazione della rete comporta la caduta delle connessioni attive.

Capitolo 2

OpenStack

OpenStack [30] è un progetto open source costituito da una collezione di strumenti per la gestione di un cloud, offrendo funzionalità di **Infrastructure as a Service** (**IaaS**). È strutturato in una serie di componenti, ognuno dedicato alla gestione di un particolare aspetto o risorsa del cloud.

Lo sviluppo è cominciato nel 2010 come un progetto congiunto della *NASA* e di *Rackspace*, per poi essere gestito a partire dal 2012 dalla *OpenStack Foundation*, un'organizzazione no profit a cui hanno aderito più di 200 società. Lo sviluppo è strutturato in rilasci di milestone ogni sei mesi. L'ultima release alla data corrente ha nome in codice *Juno*, rilasciato ad ottobre 2014, mentre per aprile 2015 è previsto il rilascio di *Kilo*. OpenStack Juno è composto dai seguenti progetti:

- **Nova (Compute)** Si occupa della gestione di pool di risorse di computazione. Supporta differenti tipologie di virtualizzazione e di hypervisor, come *KVM* (virtualizzazione completa), *Xen* (paravirtualizzazione) e *LXC* (virtualizzazione a livello di sistema operativo). Per una lista completa è possibile fare riferimento a [17].

- **Swift (Object Storage)** Servizio di storage di file ed oggetti. Offre ridondanza dei dati, tramite la replicazione su più dischi distribuiti su differenti server, e

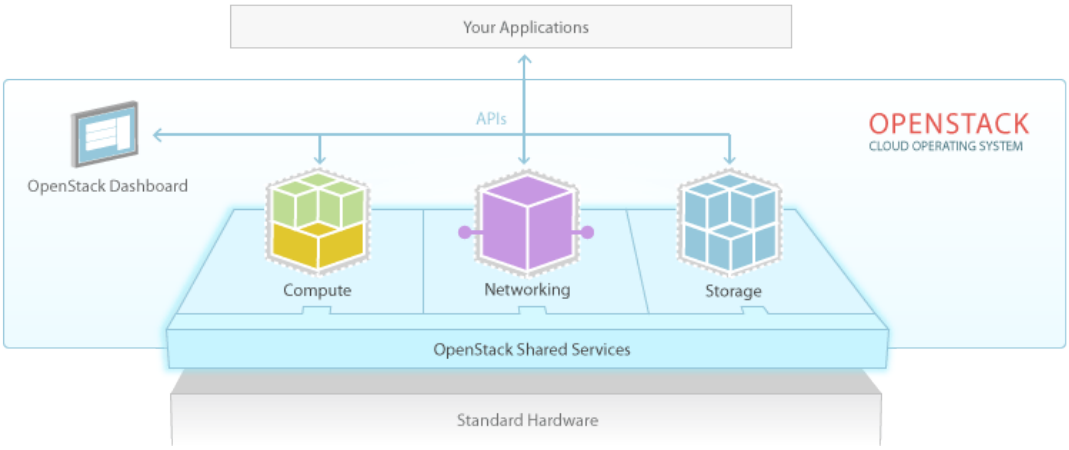

Figura 2.1: Struttura OpenStack

scalabilità orizzontale, tramite l'aggiunta di nuovi server.

- **Cinder (Block Storage)** Progetto per la gestione di device a blocchi, denominati *volumi*, associabili alle istanze di computazione.

- **Neutron (Networking)** Un Componente per la gestione delle reti e degli indirizzi IP. Tramite Neutron, gli utenti possono definire reti virtuali, in cui avviare le proprie istanze. Agli amministratori di sistema fornisce gli strumenti per la creazione di **Software Defined Network** (**SDN**), tramite il protocollo **OpenFlow**.

- **Horizon (Dashboard)** Una interfaccia web per la gestione del cloud. Offre strumenti agli utenti e agli amministratori di sistema.

- **Keystone (Identity Service)** Sistema di autenticazione per la gestisce gli utenti di un cloud e dei relativi diritti di accesso per ogni servizio.

- **Glance (Image Service)** Componente dedicato alla gestione delle immagini di disco.

- **Ceilometer (Telemetry)** Servizio di monitoraggio di risorse e collezione di misure di metriche. Offre la possibilità di creare metriche personalizzate e di impostare allarmi sulla base di statistiche sulle misure collezionate.
- **Heat (Orchestration)** Componente di orchestrazione per la gestione di risorse e applicazioni, definite tramite template. Offre funzionalità di **autoscaling** tramite la definizione di allarmi con Ceilometer.
- **Trove (Database)** Offre funzionalità di *Database as a Service*, con il supporto a *database relazionali* e *NoSQL*.
- **Sahara (Data Processing)** Servizio per la creazione di cluster di *data processing* basati su *Hadoop* o *Spark*.

I componenti appena descritti rappresentano i progetti ufficialmente facenti parte dell'ecosistema di OpenStack, versione Juno. Esistono anche progetti non ancora inclusi nelle versioni ufficiali, che offrono funzionalità aggiuntive.

A seguire verrà effettuata un'analisi di OpenStack, in relazione al supporto alla migrazione di macchine virtuali. Si introducono quindi una serie di strumenti utilizzati in questo lavoro di tesi. Per cominciare verrà presentato OpenStack Neat, un progetto per il consolidamento di macchine virtuali. Si procede quindi con Ceilometer, il progetto per il monitoraggio di risorse in OpenStack. Infine si conclude con la presentazione di KWAPI, un framework per la raccolta di dati sul consumo energetico.

2.1 Migrazione in OpenStack

Il supporto alla migrazione è fornito da Nova, il componente dedicato alla gestione delle risorse di computazione. Le tipologia di migrazione supportate sono [26]:

- **Migration**. *Cold Migration* in cui un'istanza viene interrotta prima di eseguire la migrazione.
- **Live Migration**. *Hot Migration* in cui l'istanza viene migrata durante l'esecuzione. Sono supportate tre modalità:
  - **Shared storage-based live migration**. La live migration avviene tramite storage condiviso, per cui è necessario migrare unicamente la memoria, come descritto nel Capitolo 1.
  - **Block live migration**. Non è necessario configurare uno storage condiviso. Vengono migrati memoria e disco.
  - **Volume-backed live migration**. Se la VM è associata ad un volume (gestito dal progetto *Cinder*), non è necessario configurare uno storage condiviso. Viene migrata unicamente la memoria. Solo gli hypervisor gestiti tramite le API **libvirt** supportano questa modalità.

La migrazione in WAN (par. 1.4) è praticabile unicamente se l'hypervisor con cui è stato configurato Nova, supporta questa caratteristica.
Fra i progetti ufficiali facenti parte dell'ecosistema di OpenStack, non ne esiste alcuno per la definizione di politiche di migrazione.

In [7] è stato proposto **OpenStack Neat** [31] per supplire a questa mancanza.

2.2 OpenStack Neat

OpenStack Neat è un progetto dell'Università di Melbourne avente l'obiettivo di ottimizzare l'utilizzo delle risorse fisiche di un data center, riducendo il consumo energetico dell'infrastruttura. Tale obiettivo è perseguito tramite il *consolidamento* di macchine

virtuali (VM), sfruttando le tecniche di *live migration* descritte nel capitolo 1.
Il problema del consolidamento di VM può essere suddiviso in quattro sottoproblemi:

1. Determinazione della condizione di **sottoutilizzo** di un host. In tal caso è necessario migrare tutte le VM in esecuzione sull'host, ponendolo successivamente in uno stato di risparmio energetico.

2. Determinazione della condizione di **sovraccarico** di un host. Per evitare un degrado delle prestazioni, è necessario far rientrare l'host in normali condizioni di utilizzo, migrando un sottoinsieme di VM verso altri host.

3. **Selezione** delle VM da migrare in un host in sovraccarico.

4. **Posizionamento** delle VM da migrare a seguito del rilevamento di condizione di sottoutilizzo o sovraccarico di un host.

Tramite la suddivisione in sottoproblemi, si hanno vantaggi in termini di

- Flessibilità: ogni problema è risolvibile in maniera indipendente dagli altri.

- Scalabilità: è possibile incaricare componenti diversi per la soluzione di ogni problema.

Neat offre nativamente diversi algoritmi per la soluzione di ogni sottoproblema. È inoltre configurato per essere facilmente esteso tramite l'aggiunta di nuovi algoritmi.

2.2.1 Architettura

OpenStack Neat è organizzato in tre tipologie di componenti, come mostrato in fig. 2.2:

- **Global Manager**. È il componente centralizzato che si occupa di prendere decisioni a livello globale. In particolare esegue l'algoritmo di posizionamento delle VM, ed avvia il processo di migrazione sulla base dei risultati ottenuti.

- **Local Manager**. È un componente distribuito, localizzato su ogni nodo di computazione. Prende decisioni locali, quali la determinazione dello stato di un host e la selezione delle VM da migrare in un host sovraccarico. Una volta rilevata una condizione di sottoutilizzo o sovraccarico, invia una richiesta al global manager.

- **Data Collector**. È distribuito assieme al local manager su ogni nodo di computazione; si occupa di collezionare dati sull'utilizzo della **CPU** da parte delle VM e degli host. I dati collezionati vengono salvati in locale, a disposizione del Local Manager, ed inviati al **database centrale**, per essere utilizzati dal Global Manager nell'algoritmo di piazzamento delle VM.

I componenti distribuiti garantiscono intrinsecamente scalabilità e resistenza a failure:

- Se è necessario monitorare un nuovo host, si crea semplicemente una nuova replica del local manager e del data collector. L'algoritmo di selezione delle VM può essere eseguito da una replica, senza necessità di collaborare con le altre.

- Non è presente un singolo punto di fallimento per la raccolta delle risorse e il rilevamento dello stato degli host.

Queste caratteristiche non sono invece presenti nel componente centralizzato. La resistenza alle failure può essere garantita creando una seconda replica del Global Manager, entrante in funzione al fallimento della prima. Il discorso della scalabilità è più complesso. È possibile creare più repliche del Global Manager, in esecuzione

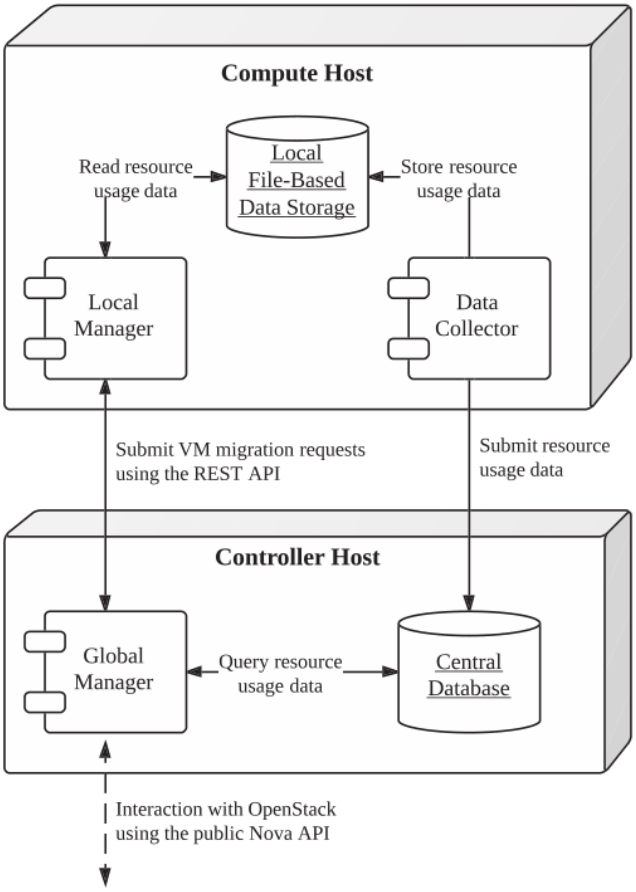

Figura 2.2: Architettura OpenStack Neat

parallela, posizionate dietro ad un load balancer. Vi è però la necessità di modificare l'algoritmo di VM Placement, aggiungendo collaborazione fra le repliche; in caso contrario, qualora due repliche migrassero VM sullo stesso host, le informazioni in loro possesso risulterebbero datate, non tenendo conto l'una delle azioni dell'altra.

L'unica metrica presa in considerazione dal data collector è rappresentata dall'utilizzo della CPU da parte delle VM e dell'host; di base, non è quindi possibile utilizzate altre metriche, come il consumo della memoria, per definire politiche di migrazione.
Per la raccolta dei dati, il data collector non si appoggia su *Ceilometer*, il progetto

ufficiale di OpenStack per la collezione di misure, in quanto quest'ultimo è stato introdotto in *OpenStack Havana* (Ottobre 2013), ossia in data successiva al rilascio di Neat (2012).

OpenStack Neat non è l'unico lavoro per il consolidamento di VM in OpenStack. In [15] è stato effettuato uno studio analogo, utilizzando come piattaforma OpenStack versione *Diablo* (Settembre 2011). In questo lavoro viene effettuata un'analisi delle conseguenze derivanti dal consolidamento di VM. Oltre ai benefici legati alla riduzione del consumo energetico, vengono sottolineate tutte le problematiche scaturenti dall'interazione fra le VM consolidate. Una serie di esperimenti pone in evidenza la riduzione delle capacità di computazione e di rete, all'aumentare del carico sulle VM, e l'overhead introdotto dagli hypervisor nel caso in cui le VM si scambino reciprocamente grandi quantità di dati.

Prima di esaminare il lavoro di tesi realizzato, si procede nella descrizione approfondita di altri due componenti di OpenStack: Ceilometer e **Kwapi**.

2.3 Ceilometer

Ceilometer [22] è il progetto di OpenStack dedicato alla collezione di misure di metriche. Il monitoraggio delle risorse in un cloud è utile sotto diversi punti di vista.

- Amministratori del cloud. Monitoraggio dello stato degli host per garantire un adeguato livello di servizio agli utenti del cloud (nel rispetto dei **Service Level Agreement** (SLA)).
- Utenti del cloud. Monitoraggio delle VM per garantire un adeguato livello di servizio agli utenti delle applicazioni in esecuzione sulle VM.

- Business. Monitoraggio delle risorse utilizzate da ogni utente del cloud, da elaborare per ottenere dati sul *billing*.

In Fig. 2.3 è mostrato lo schema logico del funzionamento di Ceilometer. Ceilometer

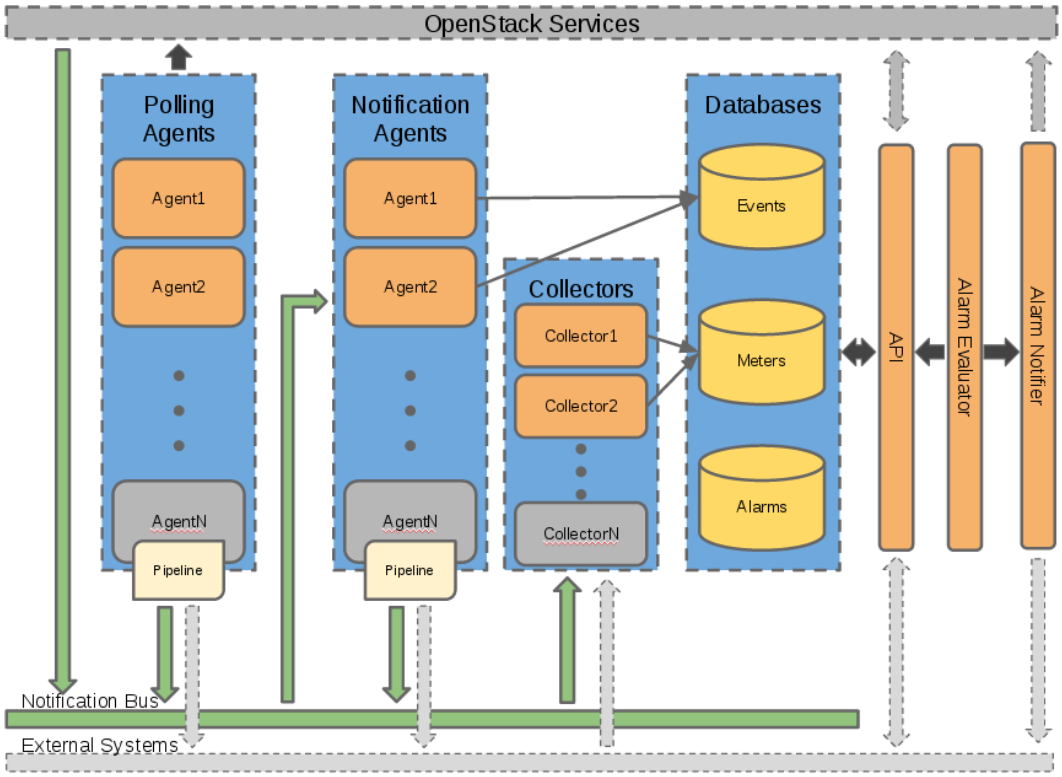

Figura 2.3: Architettura Ceilometer

è caratterizzato da un'architettura modulare, i cui componenti sono stati strutturati per poter scalare facilmente in maniera orizzontale. In dettaglio:

- **Polling Agent**. Servizi che si occupano di eseguire il polling di campioni di metriche dai vari componenti di OpenStack. Si dividono in:
  - **Compute Agent**. Distribuiti sui nodi di computazione.
  - **Central Agent**. Localizzato sul nodo controllore. È dedicato alla raccolta di campioni di metriche non legate alle istanze o ai nodi di computazione.

- **Notification Agent**. Servizio progettato per rimanere in ascolto delle notifiche inviate sulla coda dei messaggi. Una volta recuperate, le notifiche vengono convertite in campioni.

- **Collector**. Servizio designato alla raccolta degli eventi e dei campioni di metriche recuperati dai Polling e Notification Agent. Una volta verificata l'autenticità dei dati collezionati, tramite il controllo della firma associata ad ogni dato, è possibile procedere con la memorizzazione su database.

- **API**. Componente per la visualizzazione dei dati collezionati dal Collector.

- **Alarming**. Servizio per la definizione di allarmi sulla base di regole definite sugli eventi o sulle metriche collezionate.

I dati collezionabili sono relativi a metriche ed eventi.
La misura di una metrica rappresenta il valore assunto da una determinata risorsa in un certo istante di tempo, come l'utilizzazione della CPU virtuale assegnata ad un'istanza. I tipi di metriche sono tre:

- **Cumulative**. Valori crescenti nel tempo.

- **Gauge**. Valori discreti e variabili nel tempo.

- **Delta**. Variazioni di valori in diversi istanti di tempo.

Un evento rappresenta lo stato di un oggetto al verificarsi di una situazione di interesse, come la creazione di una nuova istanza in Nova. Un amministratore può definire sia metriche che eventi personalizzati, a seconda delle necessità.
La collezione di dati può avvenire secondo tre modalità:

- **Bus listener agent** Alcuni progetti di OpenStack sono configurati per inviare notifiche sul bus di comunicazione (Fig. 2.3). I Bus listener agent sono in ascolto su questo bus e trasformono in campioni le notifiche che ricevono. Questo metodo è supportato dal Notification Agent.

- **Push agent** Poichè non tutti i progetti sono configurati per inviare automaticamente notifiche, sono necessarie altre tipologie di agenti. I Push agent sono distribuiti sui nodi da monitorare. I campioni delle metriche vengono recuperati contattando i *Pollster* locali, componenti che si interfacciano con le sorgenti dei campioni, per poi essere pubblicati. È supportato dai Compute Agent.

- **Polling Agent** Componenti centralzzati che eseguono, ad intervalli prefissati, il polling dei campioni di determinate metriche, sfruttando il meccanismo dei *Pollster*. Poiché i Polling Agent devono essere installati su un nodo centralizzato, questa modalità di raccolta delle risorse è la meno preferibile, per problemi di scalabilità. È supportata dal Central Agent.

Come appena descritto, un **Pollster** è un componente richiamato periodicamente da un Polling Agent, Compute o Central, per ottenere campioni di una determinata metrica per ogni risorsa specificata. Un Pollster può essere associato ad un **Discovery**, ossia un meccanismo tramite cui recuperare tutte le risorse da cui eseguire il polling dei campioni.

Per fare un esempio, di base Ceilometer definisce il *CPUPollster*. Questo Pollster preleva ad ogni intervallo di tempo, un campione della metrica *cpu*, ossia il tempo di CPU utilizzato, per ogni macchina virtuale specificata dal Discovery. Poichè *CPUPollster* non definisce un Discovery di default, viene utilizzato quello del Compute Agent; nello specifico viene utilizzato il Discovery *InstanceDiscovery* che recupera tutte le macchi-

ne virtuali in esecuzione sull'host su cui è dislocato il Compute Agent. Il *CPUPollster* eseguirà quindi il polling dei campioni della metrica *cpu* per ogni istanza in esecuzione sull'host.

I Discovery sono organizzati per priorità. Se un discovery di una determinata priorità non è specificato, viene utilizzato quello a priorità minore, secondo il seguente ordine:

- Discovery associati ad una Sorgente di campioni (opzionale).
- Discovery associati ad un Pollster (opzionale).
- Discovery associati ad un Agent (obbligatorio).

Una volta collezionato i campioni di una metrica, questi vengono inviati nelle **Pipeline** (Fig. 2.4), un meccanismo che sottopone i campioni ad una serie di trasformazioni, prima di pubblicarli. In particolare, una pipeline è costituita da zero o più **transformer** e termina con uno o più **publisher**. È possibile definire una pipeline per ricevere

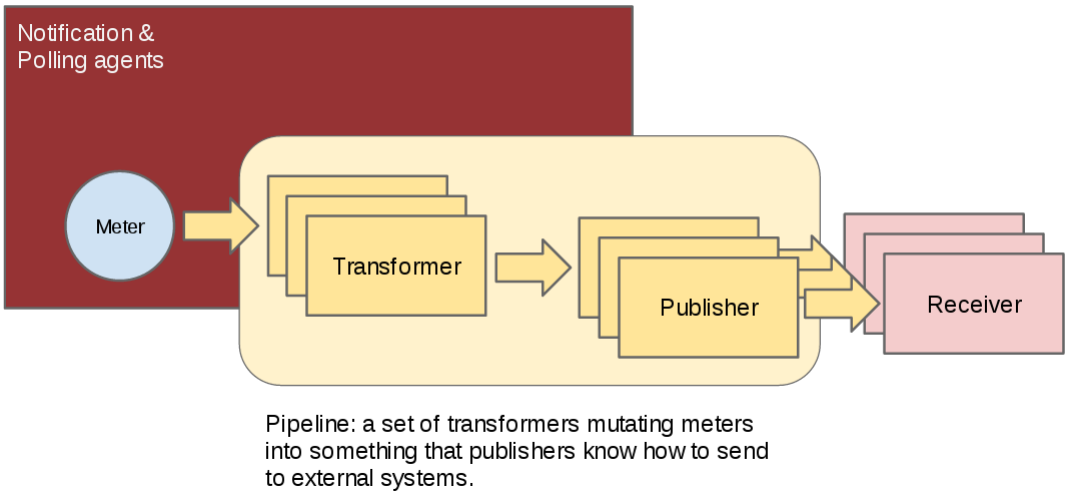

Figura 2.4: Funzionamento Pipeline

campioni di diverse metriche, con la restrizione che la **sorgente sia la stessa**; ad esempio, la sorgente *Disk* può essere configurata per inviare in una pipeline i campioni

disk.read.bytes e *disk.write.bytes*. Come si vedrà successivamente, in questo lavoro di tesi è stato necessario superare questo limite, in quanto troppo restrittivo.
Una volta entrati in una pipeline, i campioni vengono sottoposti ad un insieme di *trasformatori*, che produce in uscita campioni di una *metrica derivata*. Ad esempio, a partire da *disk.read.bytes* e *disk.write.bytes* è possibile produrre un campione della metrica derivata *disk.read.write.bytes*, che rappresenta la somma dei byte letti ed inviati in un certo intervallo di tempo. Un altro esempio è rappresentato dai campioni di *cpu_util*, l'utilizzazione della cpu, prodotti a partire dai campioni della metrica *cpu*.
Una pipeline termina con una serie di *publisher*, che pubblicano i campioni, ognuno con un metodo differente.
I trasformatori disponibili sono:

- **Rate of change**. Per derivare statistiche percentuali sulle metriche.
- **Unit conversion**. Per cambiare scala.
- **Aggregator**. Per aggregare un certo numero di campioni di misure, e selezionare unicamente un sottoinsieme da mandare avanti nella pipeline.
- **Accumulator**. Per accumulare un determinato numero di campioni, prima di procedere nella pipeline.
- **Multi meter arithmetic transformer**. Per eseguire calcoli aritmetici su campioni provenienti da una o più metriche. Questo trasformatore è limitato a metriche relativa alla stessa risorsa (e.g. istanza) e aventi lo stesso intervallo (che determina quanto spesso i campioni debbano essere inviati in una pipeline). È stato introdotto in OpenStack Juno.

I possibili publisher sono:

- **udp**. Per pubblicazione tramite protocollo UDP.
- **rpc**. Per pubblicazione tramite RPC.
- **notifier**. Per pubblicazione tramite invio di notifiche su bus di comunicazione.

La definizione delle pipeline avviene tramite il file di configurazione **pipeline.yaml**, di cui è possibile visionare un estratto in list. 2.1.

```
sources:
   [...]
    - name: cpu_source
      interval: 600
      meters:
          - "cpu"
      sinks:
          - cpu_sink
    [...]
sinks:
   [...]
    - name: cpu_sink
      transformers:
          - name: "rate_of_change"
            parameters:
                target:
                    name: "cpu_util"
                    unit: "%"
                    type: "gauge"
                    scale: "100.0 / (10**9 * (resource_metadata.cpu_number or 1))"
      publishers:
          - notifier://
     [...]
```

Listing 2.1: Estratto d'esempio del file pipeline.yaml

In list. 2.1 è definita una nuova pipeline dall'associazione sorgente-sink. In particolare è possibile notare:

- La definizione della sorgente *cpu_source*. Tale sorgente produce un campione della metrica *cpu* (tempo di cpu) ogni intervallo di *600* secondi. Il campione viene inviato al sink *cpu_sink*.
- La definizione del sink *cpu_sink*. Qui il campione di *cpu* viene sottoposto al trasformatore *rate_of_change* (i dettagli del fattore di scala verranno descritti nel

capitolo successivo), che produrrà in uscita un campione della metrica *cpu_util* (utilizzazione di cpu), di tipo *gauge* e con la percentuale come unità di misura. Tale campione sarà pubblicato per mezzo del publisher *notifier*.

Nell'ottica di monitoraggio delle risorse di un cloud, gli allarmi di Ceilometer (introdotti in OpenStack Havana) risultano molto importanti. Ad esempio, il progetto di orchestrazione OpenStack **Heat**, permette la definizione di allarmi per scatenare azioni di **autoscaling**.

Un allarme è caratterizzato da una regola, da uno stato, e da una serie di azioni da compiere quando avvengono transizioni di stato. Tornando all'esempio di autoscaling, è possibile definire due allarmi, uno per scatenare l'azione di upscale, l'altro per scatenera l'azione di downscale.

```
ceilometer alarm-threshold-create \
    --name cpu_high --description 'overloaded' \
    --meter-name cpu_util \
    --threshold 70.0 --comparison-operator gt  --statistic avg \
    --period 600 --evaluation-periods 3 \
    --alarm-action 'http://controller:9710/overload' \
    --repeat-actions true \
    --query resource_id=INSTANCE_ID
```

Listing 2.2: Creazione allarme upscaling

Il comando in list. 2.2 crea un allarme sull'utilizzazione della CPU da parte dell'istanza di cui è riportato l'id; se la media dei campioni raccolti è superiore per tre periodi di tempo successivi, ciascuno della durata di dieci minuti, alla soglia prefissata del 70%, allora viene inviata una richiesta *POST* all'URL specificato. L'URL rappresenta un **webhook**, e costituisce l'endpoint su cui è in esecuzione un servizio che alla ricezione di una richiesta di overload, esegue l'azione di upscaling. L'allarme viene valutato ogni **evaluation_interval**, un parametro posto di default a 60 secondi. Questo valore deve essere superiore a quello dell'intervallo di collezione dai campioni,

per cui il valore di default è eccessivamente piccolo. L'opzione *repeat-actions*, specifica se l'azione deve essere ripetuta ad ogni intervallo di valutazione, nel caso in cui la condizione sia ancora valida. Di default questo parametro è disabilitato.

```
ceilometer alarm-threshold-create \
    --name cpu_low --description 'underloaded' \
    --meter-name cpu_util \
    --threshold 20.0 --comparison-operator lt  --statistic avg
    --period 600 --evaluation-periods 3 \
    --alarm-action 'http://controller:9710/underload' \
    --repeat-actions true \
    --query resource_id=INSTANCE_ID
```

Listing 2.3: Creazione allarme downscaling

Il comando in list. 2.3 definisce un allarme per reagire ad una situazione opposta a quella precedente; l'istanza analizzata si trova in una condizione di overload se per tre periodi successivi da dieci minuti ciascuno, l'utilizzazione media della CPU è stata del 20%. In caso l'allarme si attivi, viene inviata una richiesta *POST* all'url specificato, dietro cui è in esecuzione il servizio per la gestione del downscaling.
Gli allarmi possono essere definiti anche per metriche relative a gruppi di risorse, utilizzando i metadati. In Heat i metadati sono utilizzati per impostare gli allarmi di autoscaling sull'insieme di VM definite in un template. In questo modo viene limitato il numero di allarmi da creare.
Per associare un metadato ad una VM, è possibile utilizzare due comandi delle API di nova:

- **nova meta**. Definisce un metadato per una singola VM.
- **nova host-meta**. Definisce un metadato per ogni VM *correntemente* in esecuzione sull'host specificato.

Un metadato, per poter essere utilizzato da Ceilometer, deve utilizzare il prefisso "**metering**". In list. 2.4 è mostrato un esempio di comando per l'associazione, a

tutte le VM in esecuzione sull'host *compute1*, di un metadato contenente il nome dell'host. Per garantire la consistenza, questa informazione deve essere aggiornata al momento in cui una VM viene migrata su un altro host, utilizzando i comandi precedentemente specificati.

```
nova host-meta compute1 set metering.compute=compute1
```

Listing 2.4: Associazione di un metadato di tipo *metering* a tutte le VM in esecuzione sull'host compute1

Una volta associato ad un gruppo di VM il metadato in list. 2.4, è possibile definire un allarme su questo gruppo di istanze tramite il comando in list. 2.5.

```
ceilometer alarm-threshold-create \
    --name cpu_low --description 'underloaded' \
    --meter-name cpu_util \
    --threshold 20.0 --comparison-operator lt  --statistic avg
    --period 600 --evaluation-periods 3 \
    --alarm-action 'http://controller:9710/underload' \
    --query resource_id=resource_metadata.user_metadata.compute=compute1
```

Listing 2.5: Definizione di un allarme sull'utilizzazione media di CPU da parte di tutte le VM in esecuzione sull'host *compute1*

Questo allarme è del tutto analogo a quello definito in list. 2.3, ma anziché essere applicato ad una singola istanza, è applicato a tutte le VM in esecuzione sull'host *compute1*.

Un allarme può trovarsi in tre stati diversi:

- **insufficient data**. Non sono presenti sufficienti dati per valutare lo stato della risorsa monitorata.
- **ok**. I dati sono stati valutati e la regola non è stata violata.
- **alarm**. I dati sono stati valutati e la regola è stata violata.

Ceilometer permette la definizione di **meta allarmi**, ossia di allarmi basati sulla combinazione **logica** (tramite operatori **OR** e/o **AND**) dello stato di altri allarmi.
Ceilometer può essere esteso per monitorare anche metriche relative al consumo energetico, tramite il progetto Kwapi.

2.4 Kwapi

KiloWatt API (*Kwapi*) [8] [23] è un framework per la collezione di metriche sul consumo energetico, fornendo in particolare informazioni sulla **potenza** (*Watt*) e sull'**energia** (*chilowattora*, *kWh*) dissipata da un host.
Sebbene sia possibile stimare in maniera indiretta il consumo energetico sulla base dell'utilizzo delle risorse di sistema, Kwapi permette di misurare direttamente i consumi tramite l'utilizzo di sonde collegate a dei *Wattmetri*. Ogni sonda fornisce informazioni sulla potenza dissipata da un host, in Watt, da cui viene ricavata l'energia dissipata in kWh.
Kwapi è costituito da una serie di componenti, organizzati secondo il pattern *publish/subscribe*. Uno schema dell'architettura è mostrato in figura 2.5. In particolare sono presenti i seguenti componenti:

- **Driver** (Publisher). I driver Kwapi sono in ascolto sui wattmetri, a cui sono collegati tramite rete o porte seriali. Una volta ricevuto nuovi dati, questi vengono impacchettati in un messaggio su cui viene applicata una firma per l'autenticazione, e quindi pubblicati sul bus su cui sono in ascolto i Plugin. Kwapi implementa diversi Driver, ognuno specifico per un determinato wattmetro presente sul mercato. È anche disponibile un driver *dummy*, che produce dati casuali, per eseguire test senza essere provvisti di un wattmetro.

- **Plugin** (Subscriber). Si occupano di collezionare i dati ricevuti dai driver, verificandone l'autenticità. Offre un'interfaccia *REST*, utilizzabile da servizi esterni per il recupero dei campioni. Questa interfaccia è sfruttata da due Pollster di Ceilometer, *EnergyPollster* e *PowerPollster*, rispettivamente per il recupero di campioni di energia e potenza dissipata. Inoltre, è possibile visualizzare dei grafici relativi ai dati collezionati, tramite l'interfaccia web fornita dal plugin **Kwapi RRD**.

- **Forwarder**. Componente opzionale con l'obiettivo di diminuire il traffico di rete. Se più di un plugin è in ascolto sulla stessa sonda, i campioni vengono inviati un'unica volta, e il forwarder si occupa di duplicarli ed inviare una copia ad ogni listener.

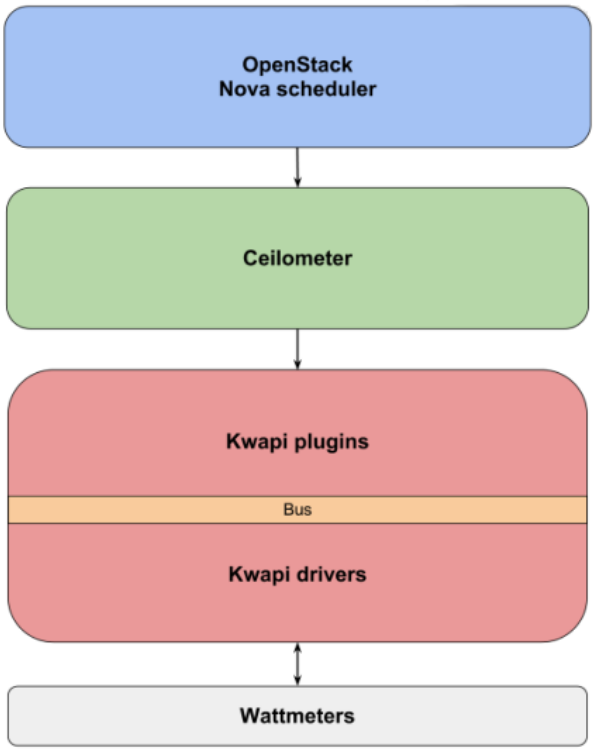

Figura 2.5: Architettura Kwapi

Capitolo 3

Estensione di Neat e Ceilometer

Nei precedenti capitoli sono state descritte le soluzioni per la migrazione di macchine virtuali e sono stati presentati Neat, un framework per il consolidamento di macchine virtuali in OpenStack, e Ceilometer, il progetto di OpenStack per il monitoraggio di risorse. In questo capitolo verrà descritto nel dettaglio il lavoro di tesi svolto.
Con l'obiettivo di realizzare politiche di migrazione basate sulla combinazione lineare di metriche prodotte da sorgenti eterogenee, sono state realizzate due estensioni; è stato modificato il meccanismo di Ceilometer di collezione, trasformazione e pubblicazione dei campioni, al fine di abilitare la possibilità di creare metriche derivate a partire da metriche eterogenee; è stato quindi esteso Neat per procedere all'integrazione con Ceilometer. L'integrazione ha reso disponibile a Neat un maggior numero di metriche, su cui definire le nuove politiche.

3.1 Estensione di OpenStack Neat

L'integrazione con Ceilometer ha richiesto diverse modifiche alla struttura di Neat. Il meccanismo di collezione dei campioni di Neat è stato sostituito con il corrispondente di Ceilometer. L'utilizzo degli allarmi di Ceilometer, per la rilevazione dello stato degli host, ha richiesto la creazione di un nuovo componente in Neat, addetto alla selezione

delle VM da migrare. Infine è stato necessario modificare anche il componente addetto al posizionamento delle VM, con l'introduzione di un algoritmo più raffinato.

3.1.1 Sostituzione del Data Collector con il Compute Agent

Come descritto nel paragrafo 2.2, OpenStack Neat utilizza un componente apposito per la raccolta di dati, denominato *Data Collector*. Questo componente è dislocato su ogni nodo di computazione, e colleziona dati sull'utilizzo della CPU da parte delle VM e dell'host. I dati sono ottenuti contattando l'hypervisor locale, tramite l'API *Libvirt*.

Per quanto descritto, di base non vi è la possibilità di creare politiche di migrazione che prendano in considerazione l'utilizzo di risorse diverse dalla CPU.

Neat non si appoggia a Ceilometer per la raccolta di dati, in quanto quest'ultimo è stato introdotto nella release *Havana* (ottobre 2013), successiva alla realizzazione di Neat (2012). Una delle prime estensioni realizzate nel lavoro di tesi consiste quindi nell'integrazione dei due progetti. In particolare, il Data Collector di Neat è stato sostituito dal *Compute Agent* di Ceilometer (par. 2.3), nel compito di collezionare misure. In questo modo si ha accesso ai dati sull'utilizzo di un più ampio insieme di risorse. Per un elenco completo delle risorse disponibili è possibile fare riferimento a [21].

L'interfacciamento con Ceilometer avviene sfruttando il client ufficiale, che implementa le relative API.

3.1.2 Sostituzione del Local Manager con l'Alarm Manager

In Neat su ogni nodo di computazione è dislocato il *Local Manager*. Sulla base dei dati collezionati dal Data Collector, il Local Manager esegue gli algoritmi per il rile-

vamento delle condizioni di overload o underload; in caso di riscontro positivo, viene eseguito l'algoritmo per la selezione delle VM da migrare, ed inviata una richiesta al *Global Manager*.

In Ceilometer è possibile utilizzare gli **allarmi**, per ricevere notifiche al verificarsi di determinati condizioni. Gli allarmi forniscono quindi un valido meccanismo per definire condizioni di overload e underload di un nodo.

Per questo motivo, è stato introdotto un nuovo componente, l'**Alarm Manager**, basato sugli allarmi di Ceilometer, che si sostituisce al Local Manager. L'Alarm Manager è uno strato software intermedio fra l'*Alarm Notifier*, componente di Ceilometer che gestisce gli allarmi, e il Global Manager di Neat. In Fig. 3.1 è mostrato il funzionamento di questo nuovo componente. In particolare, l'amministratore del cloud crea

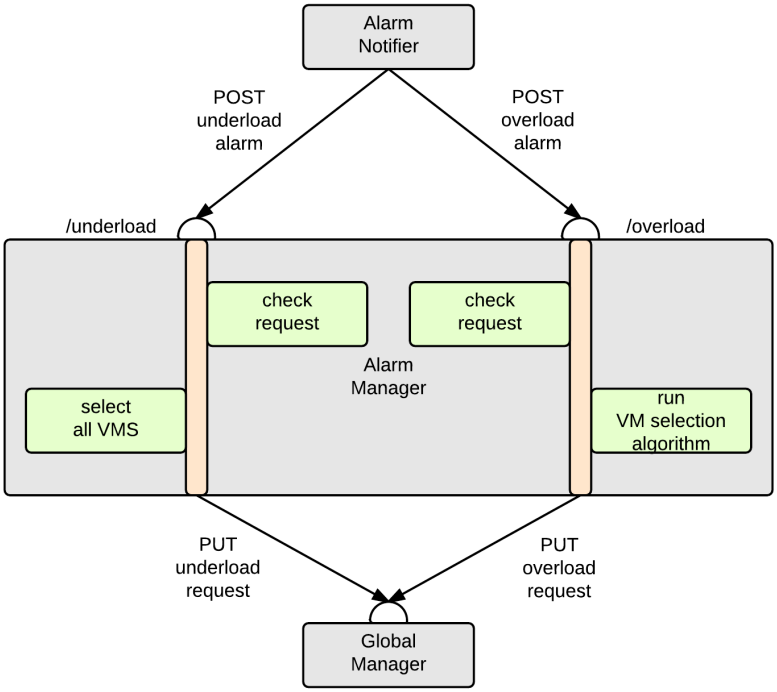

Figura 3.1: Alarm Manager

degli allarmi tramite il client di Ceilometer (di cui è possibile vedere degli esempi in

list. 2.2 e list. 2.3), definendo condizioni di sovrautilizzo e sottoutilizzo di un nodo. Tramite la definizione di un *web hook*, allo scattare di un allarme, l'Alarm Notifier invia una richiesta all'endpoint su cui è in ascolto l'Alarm Manager.

L'Alarm Manager invia al Global Manager informazioni diverse a seconda del tipo di richiesta ricevuta. In particolare, due sono le tipologie di richieste: underload e overload.

Underload

L'Alarm Notifier ha inviato una richiesta di underload, ossia di sottoutilizzo delle risorse di un host. È necessario migrare **tutte** le VM in esecuzione sull'host, per poi porlo in uno stato di risparmio energetico (comando **pm-suspend**).

L'host verrà svegliato successivamente tramite la funzione di **Wake On Lan** (comando **ether-wake**). Poiché tale funzione non è necessariamente supportata dall'host, è possibile che questo rimanga in esecuzione, anche dopo aver migrato tutte le VM verso altri host. In questo caso, se l'allarme è stato definito tramite l'attributo **repeat-action**, l'Alarm Notifier invierà ogni intervallo di valutazione una richiesta di underload all'Alarm Manager, per tutto il tempo in cui la condizione di underload continuerà ad essere verificata. Per evitare di inoltrare richieste inutili al Global Manager, viene verificato che l'host in sottoutilizzo abbia VM in esecuzione; in caso contrario la richiesta viene scartata.

Nel caso in cui l'host abbia VM in esecuzione, la richiesta viene inoltrata al Global Manager.

Overload

L'Alarm Notifier ha inviato una richiesta di overload, ossia di sovrautilizzo delle risorse di un host. È necessario selezionare un **sottoinsieme** di VM da migrare dall'host, per riportarlo in condizioni di normale utilizzo. Prima di eseguire l'algoritmo di **VM**

Selection, l'Alarm Manager verifica la validità della richiesta di overload; se la soglia di overload è stata impostata erroneamente ad un valore troppo basso, è possibile che l'host venga rilevato in condizioni di overload anche senza VM in esecuzione. Prima di procedere oltre, l'Alarm Manager verifica la presenza di VM sull'host in questione. Neat permette facilmente l'integrazione di nuovi algoritmi per la selezione delle VM. Esempi di algoritmi sono:

- *Random.* La VM da migrare viene selezionata in maniera casuale, senza tenere conto di alcuna metrica. Utile per finalità di testing. Algoritmo preesistente.
- *Minimum Migration Time Maximum CPU Utilization.* Vengono prima selezionate le VM con RAM allocata minima (indipendentemente dall'effettivo utilizzo della RAM allocata), per minimizzare il tempo di migrazione (si presuppone la presenza di storage condiviso, come definito nel paragrafo 1.4). Da questo sottoinsieme viene quindi estratta la VM con massimo utilizzo di CPU, effettuando una media sulle ultime n misure, per massimizzare la riduzione dell'utilizzo totale della CPU dell'host. Algoritmo preesistente.
- *Maximum Allocated RAM Maximum CPU Utilization.* Viene selezionata la VM con maggiore allocazione di RAM e maggiore utilizzo di CPU, in modo tale da massimizzare la riduzione del consumo di entrambe le risorse. Algoritmo appositamente realizzato.

Gli algoritmi di VM selection, a parte l'algoritmo di selezione casuale, necessitano di campioni di CPU e RAM per poter essere eseguiti. Tali dati vengono ora collezionati tramite i Compute Agent, in sostituzione dei Data Collector, come precedentemente descritto.

Dopo aver ottenuto un sottoinsieme di VM da migrare, l'Alarm Manager inoltra la

richiesta di overload al Global Manager, fornendogli le informazioni aggiuntive appena calcolate. Sebbene fosse stato possibile inserire le funzionalità dell'Alarm Manager nel Global Manager, si è deciso di tenere i componenti divisi per due motivi:

- Mantenere separate funzionalità diverse. L'Alarm Manager sostituisce il Local Manager e prende decisioni locali, a livello di singolo host; in particolare seleziona le VM da migrare da un host. Il Global Manager prende decisioni globali, in quanto esegue l'algoritmo per il posizionamento delle VM da migrare sui vari host del cloud.

- Resistenza a *failure*. Integrando l'Alarm Manager nel Global Manager, verrebbero riunite in un unico componente tutte le funzionalità di Neat, generando un unico punto di fallimento.

3.1.3 Integrazione di Ceilometer nel Global Manager

Il Global Manager è il componente di Neat che prende decisioni globali; in particolare decide per ogni VM selezionata, l'host su cui migrarla. È configurato per essere facilmente estendibile con nuovi algoritmi. In particolare sono presenti due algoritmi:

- *Random*. Le VM vengono disposte su host selezionati casualmente. Utile per finalità di testing. Algoritmo appositamente realizzato.

- *Bin Packing - Best Fit Decreasing*. Il problema di Bin Packing consiste nel posizionare oggetti di una certa dimensione, all'interno del numero minimo di contenitori, ciascuno con una capacità massima. Questo problema può quindi essere equiparato a quello del posizionamento delle VM sugli host, in cui ogni VM ha determinati requisiti e ciascun host una capacità massima. Capacità e requisiti sono espressi in termini di CPU e RAM.

Il Bin Packing è un problema *NP-Hard*, per cui è richiesta un'euristica per trovare una soluzione; in Neat viene utilizzata una versione modificata dell'algoritmo **Best Fit Decreasing**. Questo algoritmo esegue un ordinamento decrescente per le VM, in relazione all'utilizzo di CPU, e crescente per gli host, in relazione alla frequenza di CPU ancora disponibile. Partendo dalla VM con maggiore utilizzazione e dall'host con minore disponibilità di CPU, una VM viene posizionata su un host solo se quest'ultimo ha capacità sufficiente per rispettare il requisito di CPU e RAM. Gli host inattivi vengono utilizzati solo come ultima risorsa (ossia dopo aver controllato tutti gli host attivi), in quanto riattivarli costituisce un costo aggiuntivo. In questo algoritmo si realizza il consolidamento di macchine virtuali, in quanto le VM vengono concentrati su un numero ristretto di host. Algoritmo preesistente.

- *Bin Packing - Best Fit Decreasing Modificato.* L'algoritmo normale di Best Fit Decreasing presenta una problematica; nella scelta dell'host prende in considerazione unicamente la disponibilità di CPU e RAM, senza tenere conto che l'host a seguito della migrazione possa trovarsi in stato di overload. Per questo motivo si può presentare la situazione in cui una VM viene migrata in continuazione fra due host sovraccarichi, quando potrebbe essere posta su un terzo host sottoutilizzato. Per questo motivo è stata effettuata una modifica all'algoritmo. In particolare, prima di posizionare una VM su un host, viene verificato che la migrazione della VM non comporti un oltrepassamento della soglia di overload da parte dei campioni della metrica combinata di CPU e RAM. Algoritmo appositamente realizzato.

Il Global Manager durante, la sua esecuzione, ha necessità di dati sul consumo di CPU e RAM per ogni VM e host. In precedenza, questi dati erano recuperati da un database centrale, con cui interagivano i Data Collector. Dopo l'eliminazione di questi componenti, è stato necessario creare nuove funzioni per il recupero dei dati in questione. Queste funzioni utilizzano il client di Ceilometer. Per garantire retrocompatibilità con l'architettura precedente, sono state mantenute le vecchie funzioni. Il Global Manager può quindi gestire richieste in arrivo dall'Alarm Manager e dal Local Manager; un parametro nella richiesta *PUT* specifica la provenienza.
L'architettura finale della soluzione realizzata è mostrata in fig. 3.2.

3.1.4 Architettura ad alta disponibilità

Nel paragrafo 2.2.1 è stato affrontato il discorso dell'alta disponibilità dell'architettura originale di Neat. È ora necessario eseguire la stessa analisi per i nuovi componenti. Facendo riferimento alla fig. 3.2, si possono fare le seguenti considerazioni:

- I componenti distribuiti di Ceilometer, come il Compute Agent, sono scalabili per natura intrinseca.
- I componenti centralizzati di Ceilometer (Collector, API, Alarm Notifier, Alarm Evaluator) possono essere replicati, e ogni replica posta dietro un load balancer.
- La base di dati centralizzata, in cui vengono salvati i dati di Ceilometer, può essere replicata. La gestione della replicazione è delegata al DBMS utilizzato (ad esempio **MongoDB**), in maniera del tutto trasparente all'applicazione.
- L'Alarm Manager è un componente centralizzato, che va a sostituire il Local Manager, componente distribuito. Per continuare a garantire resistenza a failure è necessario replicarlo, ed utilizzare un load balancer per distribuire il carico.

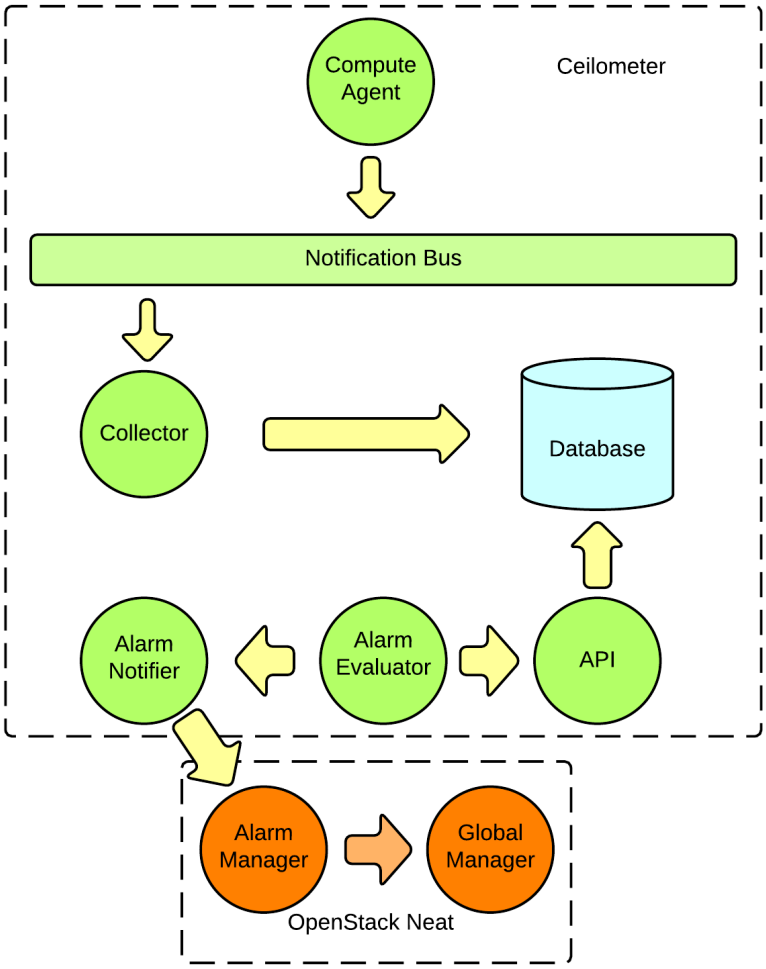

Figura 3.2: Architettura risultante dall'integrazione di Ceilometer in OpenStack Neat

Poichè questo componente prende decisioni locali, ogni replica è indipendente l'una dall'altra e quindi non vi è bisogno di collaborazione. La replicazione garantisce quindi anche scalabilità, senza necessità di modificare gli algoritmi eseguiti da questo componente.

- Per quanto riguarda il Global Manager, le considerazioni rimangono le stesse del par. 2.2.1.

3.2 Estensione di Ceilometer

Ceilometer offre la possibilità di definire metriche derivate, tramite il meccanismo dei **transformer** (par. 2.3); di particolare interesse per questo lavoro, risulta essere il **multi meter arithmetic transformer** (da qui in poi *arithmetic transformer* per semplicità), introdotto nella release *Juno* (ottobre 2014). Questo transformer è utile per effettuare calcoli aritmetici con una o più metriche e/o relativi metadati. In particolare, può essere utilizzato per definire nuove metriche a partire dalla **combinazione lineare** di altre. Questo meccanismo, così come definito nella release *Juno*, presenta dei limiti. Per superarli è stato necessario effettuare una serie di importanti modifiche alla base del funzionamento dei *polling agent*.

3.2.1 Bug nel multi meter arithmetic transformer

In list. 3.1 è mostrato un estratto di esempio del file *pipeline.yaml*, utilizzato per la definizione di nuove pipeline. In particolare, a partire dall'associazione fra la sorgente *disk_source* e il sink *disk_read_write_sink*, viene definita una nuova pipeline per produrre campioni di *disk.read.write.bytes*; questa metrica rappresenta la somma dei byte letti e scritti da/su disco, ed è prodotta combinando campioni di *disk.read.bytes* e *disk.write.bytes*.

La definizione di questa metrica ha portato alla luce l'esistenza di un bug, che impediva l'utilizzo dell'arithmetic transformer con metriche differenti; era possibile effettuare unicamente combinazioni lineari di campioni della stessa metrica. Nello specifico, la pipeline provava a pubblicare i campioni della metrica derivata, prima che fossero disponibili entrambi i campioni delle due metriche semplici, portando al fallimento dell'operazione.

Il bug è stato segnalato ed è stato fornito supporto nella relativa risoluzione [20].

```
sources:
   [...]
    - name: disk_source
      interval: 600
      meters:
          - "disk.read.bytes"
          - "disk.write.bytes"
      sinks:
          - disk_read_write_sink
    [...]
sinks:
   [...]
    - name: disk_read_write_sink
      transformers:
          - name: "arithmetic"
            parameters:
                target:
                    name: "disk.read.write.bytes"
                    unit: "B"
                    type: "cumulative"
                    expr: "1*$(disk.read.bytes) + 1*$(disk.write.bytes)"
      publishers:
          - rpc://
     [...]
```

Listing 3.1: Combinazione di metriche provenienti dalla stessa sorgente

3.2.2 Ridefinizione di pipeline

Test successivi hanno messo in risalto un limite intrinseco a Ceilometer: **l'impossibilità di definire combinazioni di metriche prodotte da sorgenti differenti**; l'arithmetic transformer poteva essere utilizzato unicamente con metriche prodotte dalla stessa sorgente, come la metrica *disk.read.write.bytes* mostrata in list. 3.1 . Questa limitazione è dovuta alla definizione di pipeline come *"l'associazione fra* ***una*** *sorgente ed un sink"*, come mostrato in fig. 3.3. La presenza di questa limitazione im-

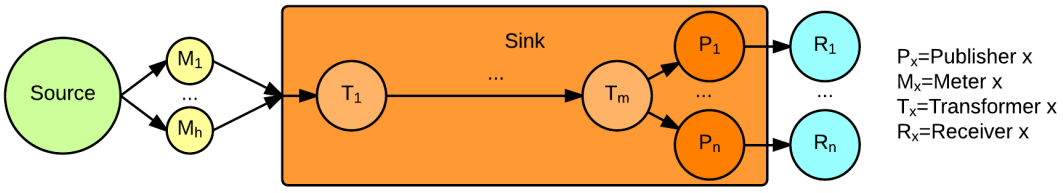

Figura 3.3: Vecchia definizione di pipeline

pediva l'utilizzo dell'arithmetic transformer per la definizione di metriche come quella mostrata in list. 3.2.

```
sources:
   [...]
    - name: memory_source
      interval: 600
      meters:
          - "memory.usage"
      sinks:
          - cpu_memory_sink

    - name: cpu_source
      interval: 600
      meters:
          - "cpu"
      sinks:
          - cpu_memory_sink
    [...]
sinks:
   [...]
    - name: cpu_memory_sink
      transformers:
          - name: "arithmetic"
            parameters:
                target:
                    name: "cpu_memory"
                    unit: "%"
                    type: "cumulative"
                    expr: "1*$(cpu) + 1*$(memory.usage)"
      publishers:
          - rpc://
     [...]
```

Listing 3.2: Combinazione di metriche provenienti da diversi sorgenti

L'esempio riportato definisce la metrica (di semplice esempio) *cpu_memory*, a partire dai campioni di *cpu* e di *memory.usage*, prodotti da sorgenti differenti.

Per aggirare questa limitazione, è stato necessario realizzare delle importanti modifiche nella definizione di pipeline; una pipeline è stata quindi definita come "*l'associazione fra una* ***o più*** *sorgenti ed un sink*", come mostrato in fig. 3.4. Le modifiche alla definizione di pipeline hanno avuto ripercussioni su altri componenti; in particolare, è stato necessario riadattare il meccanismo di polling e di pubblicazione dei campioni, meccanismo condiviso dal *central agent*, dal *Compute Agent*, e in parte dal *notification agent*.

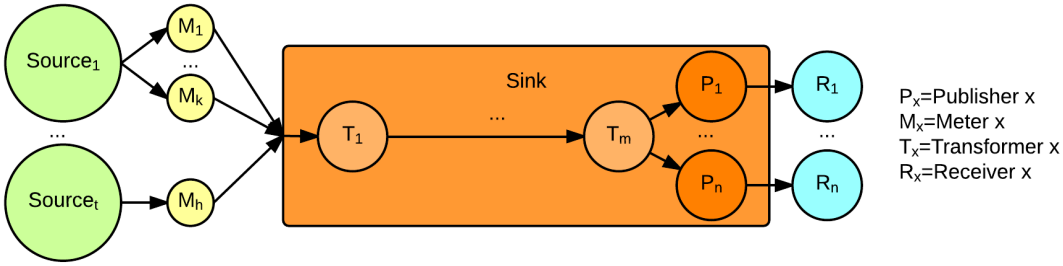

Figura 3.4: Ridefinizione del concetto di pipeline

3.2.3 Introduzione del selective transformer

Nonostante le nuove funzionalità, l'arithmetic transformer presentava ancora un limite: non era possibile creare combinazioni di metriche *derivate*, ma solo di metriche semplici. Tale limitazione si riconduceva all'impossibilità di definire su quali campioni applicare le trasformazioni. In particolare, i campioni di metriche in entrata in una pipeline dovevano subire tutti le stesse trasformazioni. È stato quindi introdotto il concetto di **selective transformer**, con cui è possibile definire su quali tipologie di campioni applicare ogni trasformatore. In fig. 3.5 è mostrato questo nuovo concetto. L'introduzione del selective transformer ha reso il meccanismo dell'arithmetic

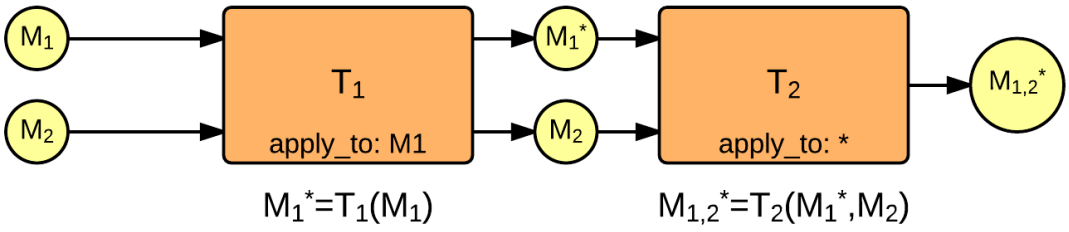

Figura 3.5: Pipeline e trasformatori selettivi

transformer molto più versatile; in particolare *un arithmetic transformer può essere applicato ai campioni di un qualsiasi numero di metriche, semplici o derivate.*

Per utilizzare un selective transformer è necessario utilizzare l'attributo (appositamente creato) **apply_to** nella definizione di una pipeline. Questo attributo accetta

due tipologie di valori:

- La wildcard "*". In questo caso la trasformazione deve essere applicata a tutti i campioni in circolazione nella pipeline. È il comportamento di default nel caso in cui non venisse specificato l'attributo *apply_to*, per garantire retrocompatibilità.
- Una lista di metriche sui cui campioni è necessario applicare la trasformazione. I campioni delle metriche non in lista non vengono modificati dal trasformatore corrente, ma passano direttamente al trasformatore successivo.

Un estratto del file *pipeline.yaml* contenente un esempio di utilizzo dell'attributo *apply_to* è mostrato in list. 3.3. La pipeline riceve campioni di *cpu* e *memory.usage* ed è costituita da due trasformazioni

- *rate of change*. Questa trasformazione è applicata unicamente ai campioni di *cpu* e produce in output campioni di *cpu.util*. I campioni di *memory.usage* non vengono modificati e passano direttamente al trasformatore successivo.
- *arithmetic*. La trasformazione aritmetica viene applicata ai campioni di *cpu.util* prodotti dalla trasformazione precedente e ai campioni di *memory.usage*, per poi produrre in output campioni di *cpu.util.memory.usage*

La pipeline sopra descritta verrà ripresa nel dettaglio successivamente.
Quanto descritto in questo paragrafo costituisce l'insieme delle modifiche strutturali apportate a Ceilometer, allo scopo di eliminare le limitazioni che impedivano la definizione di politiche di migrazione basate sullo stato di più metriche.
Per realizzare determinate politiche, di migrazione, è stato necessario estendere ulteriormente Ceilometer introducendo nuovi **pollster** e **discovery** (par. 2.3), come mostrato nel paragrafo successivo.

```
sources:
   [...]
    - name: cpu_source
      interval: 600
      meters:
          - "cpu"
      sinks:
          - cpu_util_memory_usage_sink
    - name: memory_source
      interval: 600
      meters:
          - "memory.usage"
      sinks:
          - cpu_util_memory_usage_sink
    [...]
sinks:
   [...]
    - name: cpu_util_memory_usage_sink
      transformers:
          - name: "rate_of_change"
            parameters:
                apply_to:
                    - "cpu"
                target:
                    name: "cpu.util"
                    unit: "%"
                    type: "gauge"
                    scale: "100.0 / (10**9 * (resource_metadata.cpu_number or 1))"
          - name: "arithmetic"
            parameters:
                apply_to:
                    - "*"
                target:
                    name: "cpu.util.memory.usage"
                    unit: ""
                    type: "cumulative"
                    expr: "1*$(cpu.util)/100 + 1*$(memory.usage)/100"
      publishers:
          - rpc://
     [...]
```

Listing 3.3: Esempio di utilizzo dell'attributo *apply_to*

3.3 Politiche di Migrazione

In questo paragrafo verranno descritte le politiche di migrazione realizzate. In particolare, ci si è concentrati sugli algoritmi per il rilevamento delle condizioni di underload ed overload di un host.

Gli algoritmi realizzati sono di tipo **con soglia**, per cui lo stato di un host viene rilevato confrontando una statistica dei campioni di una metrica calcolata in un determinato intervallo di tempo, con una soglia prefissata. In list. 3.4 è riportato un semplice esempio di comando per la definizione di un algoritmo a soglia.

```
ceilometer alarm-threshold-create \
    --name compute1_overload --description 'compute1 overloaded' \
        --meter-name compute.node.cpu.percent \
        --threshold 70.0 --comparison-operator gt  --statistic avg \
        --period 600 --evaluation-periods 2 \
        --alarm-action 'WEB_HOOK' \
        --query resource_id=compute1_compute1
```

Listing 3.4: Esempio di allarme a soglia

In particolare, tramite le API di Ceilometer, viene definito l'allarme *compute1_overload*, caratterizzato dal seguente funzionamento:

- Sono presi in considerazione i campioni della metrica *compute.node.cpu.percent*, associati al nodo di computazione *compute1* (identificato dall'id *compute1_compute1*).
- L'intervallo di tempo su cui vengono effettuate le misure, è costituito da due periodi, ciascuno della durata di 600 secondi (parametri configurabili).
- Come statistica viene utilizzata la media.
- L'allarme scatta qualora, per due periodi consecutivi, la media dei campioni venga rilevata superiore alla soglia del 70% (parametro configurabile).

- Allo scattare dell'allarme, viene inviata una richiesta all'endpoint identificato da *WEB_HOOK*.

L'allarme di esempio appena riportato può essere utilizzato per definire la condizione di overload di un host. La metrica presa in considerazione è l'utilizzazione della CPU. Un'altra metrica rilevante per lo stato di un host è costituita dall'utilizzazione della RAM. Mentre per la prima in Ceilometer è presente la metrica *compute.node.cpu.percent* (prodotta da Nova tramite il meccanismo delle **notifiche** [21], inviate ogni minuto e al verificarsi di determinate eventi, come il completamento della migrazione), per la seconda non esiste in Ceilometer una metrica apposita.
In particolare il nostro obiettivo è definire, tramite l'arithmetic transformer, una metrica combinata a partire dai campioni di utilizzo CPU e i campioni di utilizzo RAM. È importante sottolineare come l'arithmetic transformer non possa essere utilizzato con campioni prodotti tramite due meccanismi diversi, il polling e le notifiche. Per avere un maggiore controllo sull'intervallo di produzione dei campioni, si è preferito utilizzare il meccanismo di polling. Sono state quindi definite due nuove metriche per l'utilizzo di CPU e di RAM, come descritto nel paragrafo successivo.

3.3.1 Definizione di nuovi Pollster e Discovery

Per definire nuove metriche è necessario creare dei *pollster*, componenti che eseguono il polling dei campioni. A seconda della tipologia di metrica, potrebbe anche essere necessario creare un nuovo *discovery*, un componente che recupera le risorse da cui effettuare il polling.
Nel paragrafo precedente, è stato descritto come l'obiettivo fosse recuperare dati sul consumo di CPU e RAM di un host. Ceilometer non prevede un discovery per recu-

perare l'host da cui eseguire il polling, in quanto i dati sugli host vengono prodotti da *Nova* tramite il meccanismo delle *notifiche*. Per questo motivo è stato necessario definirne uno. In particolare, è stato creato **l'HostResourceIdDiscovery**, per recuperare l'id dell'host su cui è in esecuzione il *Compute Agent*.

Sono stati quindi definiti due nuovi pollster, **HostCPUPollster** e **HostMemoryUsagePollster**, che recuperano rispettivamente campioni di **host.cpu.time** (tempo di cpu) e **host.memory.usage** (utilizzo di memoria). Per recuperare dati sull'utilizzo di CPU, è necessario un ulteriore passaggio, come spiegato successivamente.

I pollster si interfacciano con l'hypervisor sottostante, utilizzando delle API; i campioni vengono recuperati unicamente se, per l'hypervisor in esecuzione, sono state definite in Ceilometer le funzioni di **inspect** relative alle metriche. Poichè i test sono stati effettuati con **QEMU/KVM** come hypervisor, sono state create funzioni di inspect che si interfacciano con *libvirt*, l'API per l'accesso a questo hypervisor. Queste funzioni sono state realizzate sulla falsariga delle omologhe in **virsh**, il client da riga di comando per utilizzare l'API **libvirt**.

Il pollster *HostCPUPollster* recupera campioni di *host.cpu.time*, contattando la funzione di inspect **inspect_host_cpu_time**, creata dopo aver esaminato il codice del comando bash **"virsh nodecpustats"**, che recupera statistiche sulla CPU. In particolare, viene recuperato il **cpu user time** e il **cpu kernel time** dell'host, e restituito il **cpu total time**, usando la relazione 3.3.1

$$cpu\_total\_time = cpu\_user\_time + cpu\_kernel\_time \tag{3.3.1}$$

È importante notare come il valore restituito sia una statistica globale, ossia sia dato dalla somma dei tempi di CPU totali per ogni CPU. Al campione **host.cpu.time** è stato quindi associato il metadato **cpuNum**, contenente il numero di CPU nell'host.

Tale metadato sarà utilizzato per normalizzare il tempo di CPU, rispetto al numero di CPU, come sarà successivamente descritto.

Il pollster *HostMemoryUsagePollster* recupera campioni di *host.memory.usage*, contattando la funzione di inspect **inspect_host_memory_usage**, creata sulla base del codice del comando bash **"virsh nodememstats"**, che recupera statistiche sull'utilizzo della memoria. In particolare, viene recuperata la memoria totale, la memoria libera, e restituita la percentuale di memoria occupata tramite la relazione 3.3.2

$$mem\_usage = \frac{mem\_total - mem\_free}{mem\_total} * 100 \tag{3.3.2}$$

3.3.2 Combinazione utilizzo CPU e utilizzo RAM

Dopo aver definito le metriche *host.cpu.time* e *host.memory.usage*, si può procedere con la definizione di una nuova metrica combinata, a partire dai campioni delle due. Allo scopo, si definisce la pipeline in list. 3.5 ed è poi possibile procedere con la definizione di un allarme basato sull'utilizzo della CPU e della memoria. Mentre l'utilizzo della RAM è già fornito dalla metrica *host.memory.usage*, per ottenere l'utilizzo della CPU, è necessario effettuare un ulteriore passaggio, come verrà presto descritto. Viene quindi definita la pipeline in list. 3.5. Nella pipeline appena definita, per produrre la metrica derivata *host.cpu.util.memory.usage*, è necessario applicare due trasformazioni.

Rate Of Change

Questo trasformatore deve essere applicato unicamente ai campioni di *host.cpu.time*, ossia il tempo totale di CPU, per produrre l'utilizzo percentuale della CPU (metrica *host.cpu.util*). Allo scopo viene utilizzato l'attributo **apply_to**, definendo così un trasformatore selettivo.

```
sources:
   [...]
    - name: host_cpu_source
      interval: 120
      meters:
          - "host.cpu.time"
      sinks:
          - host_cpu_util_memory_usage_sink
    - name: host_memory_source
      interval: 120
      meters:
          - "host.memory.usage"
      sinks:
          - host_cpu_util_memory_usage_sink
    [...]
sinks:
   [...]
    - name: host_cpu_util_memory_usage_sink
      transformers:
          - name: "rate_of_change"
            parameters:
                apply_to:
                    - "host.cpu.time"
                target:
                    name: "host.cpu.util"
                    unit: "%"
                    type: "gauge"
                    scale: "100.0 / (10**9 * (resource_metadata.cpu_number or 1))"
          - name: "arithmetic"
            parameters:
                apply_to:
                    - "*"
                target:
                    name: "host.cpu.util.memory.usage"
                    unit: ""
                    type: "cumulative"
                    expr: "0.5*$(host.cpu.util)/100 + 0.5*$(host.memory.usage)/100"
      publishers:
          - rpc://
     [...]
```

Listing 3.5: Combinazione di utilizzo CPU e utilizzo RAM

Il trasformatore *rate_of_change* applica la relazione in 3.3.3

$$transf\_sample.volume = \frac{cur\_sample.volume - prev\_sample.volume}{cur\_sample.time - prev\_sample.time} * scale \quad (3.3.3)$$

Per poter calcolare tale rapporto, occorrono almeno due campioni e quindi devono essere passati almeno due intervalli di tempo (un campione prodotto ogni intervallo di polling) dall'avvio del Compute Agent.

Applicato alla metrica *host.cpu.time*, si ottiene la relazione 3.3.4

$$host.cpu.util = \frac{(cur\_host.cpu.time - prev\_host.cpu.time)[ns]}{(cur\_sample.time - prev\_sample.time)[s]} * \frac{100}{(10^9 * cpu\_number)} \quad (3.3.4)$$

La differenza fra il tempo di CPU attuale e il tempo di CPU precedente produce un valore in nanosecondi, per cui deve essere convertito in secondi dividendo per il fattore 10^9.

Il tempo di CPU è calcolato su tutte le CPU dell'host; deve quindi essere normalizzato sul numero di CPU. Ad esempio, se l'host possiede due CPU ed in un certo istante vengono rilevati, come tempi di CPU, rispettivamente 10 e 15 secondi (dati di semplice esempio), si ha che il tempo totale sarà pari a 25 secondi, da dividere per le due CPU, ottenendo un tempo medio di CPU di 12.5 secondi.

Una volta normalizzato il tempo di CPU, questo deve essere diviso per l'intervallo di tempo trascorso fra la raccolta dei due campioni e moltiplicato per cento, ottenendo l'utilizzo percentuale della CPU nel medesimo intervallo di tempo.

Il trasformatore rate of change viene utilizzato da Ceilometer per produrre campioni *cpu_util* per ogni VM. Questo trasformatore è necessario solo se l'hypervisor utilizzato non espone nativamnte un'interfaccia per ottenere questi dati. Libvirt e Hyper-V ne sono sprovvisti; per VMware e XenAPI, in Ceilometer è stata implementata la funzione **inspect_cpu_util**.

Arithmetic Trasformer

Dopo aver calcolata la percentuale di utilizzo della CPU, si può procedere con la trasformazione successiva. Viene prodotto un campione della metrica *host.cpu.util.memory.usage*, data dalla somma pesata dei campioni delle metriche *host.cpu.util* e *host.memory.usage*. Il valore massimo raggiungibile dai campioni di questa metrica è pari alla somma dei pesi. Il valore dei pesi determina l'importanza di una metrica. In list. 3.5 il valore di entrambi i pesi è pari a 0.5, per cui le metriche avranno la stessa importanza e il valore massimo sarà pari ad 1.

3.3.3 Definizione degli allarmi di overload/underload

Una volta definita la metrica *host.cpu.util.memory.usage*, è necessario creare per ogni host una coppia di allarmi per le condizioni di overload ed underload, come quelli riportati in list. 3.6 e list. 3.7.

```
ceilometer alarm-threshold-create \
   --name compute1_overload --description 'compute1 running hot' \
   --meter-name host.cpu.util.memory.usage \
   --threshold 0.8 --comparison-operator gt --statistic avg \
   --period 360 --evaluation-periods 2 \
   --alarm-action 'http://controller:60180/overload' \
   --repeat-actions true \
   --query resource_id=HOST_ID
```

Listing 3.6: Allarme di overload per un host

```
ceilometer alarm-threshold-create \
   --name compute1_underload --description 'compute1 running cold'  \
   --meter-name host.cpu.util.memory.usage \
   --threshold 0.3 --comparison-operator lt --statistic avg \
   --period 360 --evaluation-periods 2 \
   --alarm-action 'http://controller:60180/underload' \
   --repeat-actions true \
   --query resource_id=HOST_ID
```

Listing 3.7: Allarme di underload per un host

Poichè in list. 3.5 i pesi sono stati posti entrambi a 0.5, il valore massimo è pari a 1. L'allarme di overload è stato impostato per scattare quando per due periodi di tempo, ciascuno della durata di 360 secondi, la media dei campioni della metrica *host.cpu.util.memory.usage* è superiore all'80%. L'allarme di underload è stato impostato per scattare quando il valore della media è inferiore al 30%. Modificando il valore dei pesi e delle soglie degli allarmi, è possibile rendere il sistema più o meno sensibile a valore dei campioni di una metrica. Nel capitolo successivo verranno mostrati risultati diversi, a seconda dei parametri utilizzati.

3.3.4 Kwapi

Nel paragrafo 2.4 è stato presentato *Kwapi*, il framework per la collezione di metriche sul consumo energetico. In particolare, Kwapi esegue il monitoraggio dell'energia e della potenza dissipata da un nodo, analizzando i dati raccolti da wattmetri. In assenza di questo framework, si può avere una misura indiretta del consumo energetico monitorando l'utilizzo delle risorse di un nodo, CPU in primo luogo.
Non avendo wattmetri a disposizione, Kwapi è stato testato tramite il driver *dummy*. Poichè questo driver produce semplicemente dati casuali, si è scelto di non utilizzare Kwapi nella fase finale di testing. Avendo in ogni caso analizzato questo framework, verranno qui presentati i risultati di quest'analisi.
In [9] è stato dimostrato come un server blade arrivi a consumare *450W* nei momenti di pieno carico. Prendendo questo valore come tetto massimo raggiungibile, è possibile definire un allarme di overload come in list. 3.8. In questo allarme, viene impostata come soglia il valore di *315W*, pari al 80% della potenza massima raggiungibile. L'allarme viene definita sulla risorsa identificata dall'id *PROBE_ID*, corrispondente al wattmetro monitorante il nodo *compute1*. Questo allarme può essere utilizzato in

```
ceilometer alarm-threshold-create \
   --name compute1_overload --description 'compute1 running hot' \
   --meter-name power \
   --threshold 360 --comparison-operator gt --statistic avg \
   --period 360 --evaluation-periods 2 \
   --alarm-action 'http://controller:60180/overload' \
   --repeat-actions true \
   --query resource_id=PROBE_ID
```

Listing 3.8: Allarme di overload definito sulla potenza rilevata da Kwapi

combinazione con Neat, in modo da migrare un sottoinsieme di macchine dall'host, per riportare il consumo energetico entro un livello considerato accettabile.

Sempre in [9] è riportato come il consumo energetico in idle di un server blade, sia pari a *270W*. È possibile utilizzare questo valore come soglia minima, oltre la quale inviare all'Alarm Manager di Neat una richiesta di underload.

Nel caso in cui si volesse utilizzare l'arithmetic transformer, creando una metrica combinata che tenga in considerazione anche la potenza dissipata, è possibile definire la pipeline mostrata in list. 3.9. Per ottenere il consumo percentuale di potenza, è necessario dividere il valore rilevato per il valore massimo di *450W*. Come precedentemente descritto, modificando i pesi associati ad ogni metrica, è possibile cambiarne la priorità. Nell'esempio riportato, le tre metriche hanno lo stesso peso, e quindi stessa priorità. Una volta definita questa metrica, è possibile procedere con la creazione di allarmi di overload ed underload.

```
sources:
   [...]
        - name: meter_energy
          interval: 120
          meters:
              - "power*"
              - "energy*"
          sinks:
                - combined_sink
    [...]
sinks:
   [...]
    - name: combined_sink
      transformers:
          [...]
          - name: "arithmetic"
            parameters:
                apply_to:
                    - "*"
                target:
                    name: "combined.meter"
                    unit: ""
                    type: "cumulative"
                    expr: "0.33*$(host.cpu.util)/100 + 0.33*$(host.memory.usage)/100
                        + 0.33*$(power)/450"
      publishers:
          - rpc://
     [...]
```

Listing 3.9: Arithmetic transformer e Kwapi

Capitolo 4

Risultati Sperimentali

In questo capitolo verranno presentati una serie di test eseguiti sul framework integrato, ed analizzati i relativi risultati. I test eseguiti hanno la finalità di mostrare l'utilità del framework realizzato, concentrandosi in particolare sul rilevamento dei parametri ottimali per il consolidamento di VM.

4.1 Piattafoma di test

Poichè si deciso di utilizzare OpenStack come piattaforma di cloud computing, si è presentata la necessità di configurare questo ambiente. Per lo scopo, è stata utilizzata la documentazione ufficiale di OpenStack [24].
Non avendo disponibilità immediata di macchine fisiche, inizialmente si è deciso di utilizzare macchine virtuali, ben sapendo che questa fosse una soluzione temporanea. Tramite piattaforme quali *Amazon Web Services* (*AWS*) [32] e *Digital Ocean* (*DO*) [34], è stato quindi configurato un cloud interamente virtuale, basato su OpenStack.

4.1.1 Cloud virtuale

Su AWS è stata realizzata un'architettura multi-nodo, costituita da un nodo controllore, un nodo network, e due nodi compute. Ogni nodo è stato configurato con *Ubuntu*

Server 14.04, e come versione di OpenStack si è utilizzato *IceHouse*.

Per il nodo controllore e i nodi compute sono state scelte istanze di tipo *T2.small* (1VCPU, 2GB RAM) [33]. Per il nodo network è stata utilizzata un'istanza di tipo *T2.medium* (2VCPU, 4GB RAM), per la necessità di avere tre interfacce di rete (la tipologia *T2.small* pone il limite di due interfacce di rete). In seguito si è deciso di riunire i nodi network e controller in un unico nodo, aumentando così la potenza computazionale a disposizione per le funzionalità del controller, e diminuendo il numero totale di nodi necessari.

Su questa piattaforma sono stati installati i seguenti servizi:

- *Nova*
- *Glance*
- *Horizon*
- *Keystone*
- *Neutron*
- *Ceilometer*

Il cloud realizzato offriva la possibilità di avviare macchine virtuali (Nova), di creare reti virtuali (Neutron) e di monitorare lo stato di determinate risorse (Ceilometer). Sono state eseguite prove con le distribuzioni *CirrOS*, *Ubuntu Server* e *Fedora Server* [25]. CirrOS è una distribuzione ultraleggera, appositamente realizzata per eseguire test su un cloud. In particolare, in questo contesto è stata l'unica ad offrire prestazioni accettabili. Ubuntu e Fedora erano caratterizzate da tempi di avvio sensibilmente superiori. Ovviamente questi problemi derivavano dalla natura dell'architettura realizzata, essendo interamente virtuale.

Oltre alle prestazioni, il problema principale riscontrato risiedeva nell'impossibilità di ottenere un indirizzo ip (né privato né pubblico), da parte delle istanze. Le macchine virtuali non erano quindi raggiungibili e non potevano comunicare con l'esterno. L'unico modo per eseguire il login su una macchina era quindi tramite *noVNC*, un client (fornito da Nova) per *Virtual Network Computing* (*VNC*), un sistema di desktop sharing. Questo limite è riconducibile al meccanismo di AWS degli **Elastic IP** (*EIP*), tramite cui viene eseguito il mapping tra indirizzi privati e pubblici. In particolare, ogni istanza EC2 dotata di almeno due interfacce di rete ha unicamente indirizzi privati, a cui vengono associati indirizzi pubblici tramite EIP. OpenStack invece richiede che il nodo network abbia tre interfacce di rete, di cui una con indirizzo pubblico.
Abbandonato AWS, si è proceduto con Digital Ocean. Tramite questo servizio è stato possibile realizzare un'architettura simile alla precedente, dove ogni nodo è stato configurato con *Debian 7*. In DO ogni nodo può essere configurato con due interfacce di rete, una dotata di indirizzo pubblico, l'altra di indirizzo privato. Le istanze lanciate ottenevano quindi l'indirizzo IP privato. Non era in ogni caso possibile assegnare indirizzi pubblici, non avendone a disposizione.
Essendo sprovvisti di indirizzi pubblici, per il login in un'istanza erano praticabili due soluzioni; appoggiarsi nuovamente a *noVNC*, oppure collegarsi tramite *ssh* all'indirizzo privato dell'istanza. Per la seconda soluzione, è necessario essere collegati al nodo di network ed eseguire il comando mostrato in list. 4.1.

```
sudo ip netns exec network_namespace_id ssh user@private_ip
```

Listing 4.1: utilizzo di ssh per la connessione ad un indirizzo privato tramite network namespace

Questo comando utilizza il meccanismo dei **network namespace**. Ogni network namespace costituisce una copia dell'intero stack di rete, con relative rotte, regole di firewall e device di rete. In OpenStack una rete virtuale e un router sono associati rispettivamente ai namespace DHCP e router. Gli indirizzi privati delle istanze in una rete virtuale sono definiti nel namespace DHCP associato alla rete. La lista dei namespace è ottenibile tramite il comando "*sudo ip netns*", da eseguire sul nodo di rete.

Non avendo indirizzi pubblici, per contattare l'esterno generalmente si può fare affidamente a *NAT*, *Network Address Translation*. Nella pratica le istanze non potevano contattare l'esterno, in quanto Digital Ocean blocca questa tipologia di traffico.

Sono state eseguite prove anche con *DevStack* [35], uno strumento per configurare rapidamente un cloud basato su OpenStack. Viene utilizzato principalmente per eseguire test di nuove funzionalità, senza la necessità di configurare manualmente tutti i servizi di OpenStack. In particolare, è possibile avviare l'installazione di OpenStack su una singola macchina virtuale.

	Controller Node HP Proliant DL580 G3	
CPU	Intel Xeon 7020 Dual-Core HT @2.66 GHz	16 VCore
	Intel Xeon 7020 Dual-Core HT @2.66 GHz	
	Intel Xeon 7020 Dual-Core HT @2.66 GHz	
	Intel Xeon 7020 Dual-Core HT @2.66 GHz	
RAM	4x2GB DDR 2 @400 Mhz HP	24 GB
	4x2GB DDR 2 @400 Mhz HP	
	4x1GB DDR 2 @400 Mhz HP	
	4x1GB DDR 2 @400 Mhz HP	
DISK	RAID 5 (Hardware)	145 GB
	4x HDD 72.8 GB 15K rpm	
OS	Ubuntu Server 14.04.1	
	Linux 3.13.0-34-generic	

Tabella 4.1: Specifiche hardware del nodo Controller

4.1.2 Cluster

Dopo aver eseguito test sulle piattaforme sopra riportate, si è potuto procedere su un cluster costituito da macchine fisiche. Questo cluster è stato realizzato come progetto in un altro lavoro di tesi svolto presso l'università di Tor Vergata [10].

Il cluster è costituito in totale da sette nodi, la cui configurazione è mostrata nelle relative tabelle:

- Un nodo **Controller** (Tab. 4.1).
- Un nodo **Network** (Tab. 4.2).
- Quattro nodi **Compute** (Tab. 4.3).
- Un nodo **Storage** (Tab. 4.4).

	Network Node HP Proliant DL380 G4	
CPU	Intel Xeon Processor Dual-Core HT @2.80 GHz	8 VCore
	Intel Xeon Processor Dual-Core HT @2.80 GHz	
RAM	2x4GB DDR 2 @400 Mhz HP	14 GB
	2x2GB DDR 2 @400 Mhz HP	
	2x1GB DDR 2 @400 Mhz HP	
DISK	RAID 5 (Hardware)	364 GB
	2x HDD 72.8 GB 15K rpm	
	4x HDD 146.8 GB 10K rpm	
OS	Ubuntu Server 14.04.1	
	Linux 3.13.0-34-generic	

Tabella 4.2: Specifiche hardware del nodo Network

	Compute Node HP Proliant ML150 G6	
CPU	Intel Xeon E5504 Quad-Core @2.50 GHz	8 VCore
	Intel Xeon E5504 Quad-Core @2.50 GHz	
RAM	2x4GB DDR 3 @800 Mhz Hyundai/Kingstone	16 GB
	4x2GB DDR 3 @800 Mhz Micron	
DISK	RAID 1 (Hardware)	250 GB
	2x HDD 250 GB 7200 rmp	
OS	Ubuntu Server 14.04.1	
	Linux 3.13.0-34-generic	

Tabella 4.3: Specifiche hardware di un nodo Compute

	Storage Node HP Proliant Microserver NL40	
CPU	AMD Turion II Neo Dual-Core @1.50 GHz	2 VCore
RAM	2GB DDR 3 @1333 Mhz HP	2 GB
DISK	LVM	500 GB
	2x HDD 250 GB 7200 rpm	
OS	Ubuntu Server 14.04.1	
	Linux 3.13.0-34-generic	

Tabella 4.4: Specifiche hardware del nodo Storage

Il Cluster è configurato con OpenStack versione **IceHouse**. I servizi installati sono:

- *Nova.*
- *Glance.*
- *Horizon.*
- *Keystone.*
- *Neutron.*
- *Cinder.*

Partendo da questa configurazione si è proceduto con l'installazione degli altri servizi. In particolare:

- Sul nodo controllore sono stati installati **l'Alarm Manager** e il **Global Manager** di OpenStack Neat (par. 3.1).
- Sul controllore e sui nodi di computazione sono stati installati i componenti di **Ceilometer** (versione **Juno**, modificata come riportato in par. 3.2).
- Sul nodo storage è stato configurato uno storage condiviso (par. 1.4.1), accedibile tramite **Network File System** (*NFS*). Su questo spazio vengono posti i dischi delle istanze avviate sui nodi di computazione. Poichè lo storage ha una capienza limitata a 180GB, solo tre nodi su quattro sono stati configurati per utilizzare questo spazio. Poiché la *live migration* in OpenStack richiede la presenza di uno storage condiviso (par. 2.1), durante i test sono stati utilizzati solo i tre nodi di computazione configurati in questo modo.

4.2 Tool di benchmark

Come strumenti di benchmark sono stata utilizzati dei programmi appositamente realizzati dall'autore di OpenStack Neat [7]. Questi programmi sono organizzati secondo il paradigma *Master-Slave*, in cui il master impone ad ogni slave il carico da eseguire. Il carico è costituito da tracce di utilizzo di CPU. I dati sono relativi a VM di *PlanetLab* in dieci giorni casuali compresi fra Marzo e Aprile 2011 [27]. In particolare sono state scelte tracce con alta variabilità, secondo i seguenti criteri:

- Il 10% dei valori è superiore al 20% di utilizzazione di CPU.
- Il 10% dei valori è superiore all'80% di utilizzazione di CPU.

Il **Workload Distributor** (Master) riceve richieste di carico. Queste richieste vengono servite o scartate a seconda di un parametro definito in un file di configurazione. In questo modo è possibile avviare gli slave nello stesso istante di tempo. Quando il Distributor riceve una richiesta di carico, prende una traccia di CPU fra quelle a disposizione e la invia allo slave. La stessa traccia non viene mai assegnata a due slave diversi. Questo componente centralizzato è stato posto su un nodo esterno al cluster, in modo da non inquinare i risultati dei test.

Il **Workload Starter** (Slave) invia richieste al master per ottenere il carico. Alla ricezione della traccia di CPU, invoca il **Cpu Load Generator**. Questo programma richiama ad intervalli predefiniti (fissati dal master) il tool **Lookbusy** [28], un generatore di carico sintetico. In particolare è possibile generare carico di CPU, RAM e disco.

Per testare OpenStack Neat, l'autore ha utilizzato Lookbusy facendo generare carico su **un'unica** CPU del sistema. Il valore dell'utilizzazione veniva recuperato leggendo ad ogni iterazione la traccia di PlanetLab ricevuta dal master.

Questo comportamento è stato modificato. Nel paragrafo 3.3 è stato descritto come sia stata stata definita una metrica basata sulla combinazione di utilizzo di CPU e RAM. Per questo motivo è stato modificato il Cpu Load Generator per invocare Lookbusy richiedendo utilizzo di entrambe le metriche. In particolare, dopo aver letto il valore di carico CPU dalla traccia di Planetlab, viene invocato Lookbusy per generare questo carico su **tutte le CPU** della VM. Inoltre, viene occupata staticamente l'85% della memoria libera della VM.
L'utilizzo di memoria è stato definito staticamente per due motivi:

- Assenza di informazioni sull'utilizzo di RAM nelle tracce di PlanetLab.
- Quando una VM viene avviata, l'host non fornisce subito tutta la RAM richiesta dalla tipologia di VM. Al momento di effettivo bisogno, questa viene fornita dall'host sfruttando il meccanismo del **Memory Balloon** [16]. In seguito, anche se l'utilizzo della RAM da parte della VM diminuisce, questa non viene subito recuperata dall'host. Utilizzare un carico dinamico di RAM è stato quindi considerato poco utile, in quanto il carico rimane pari al valore più alto di utilizzazione precedentemente riscontrato nella traccia. Di conseguenza, si è preferito lavorare con un valore statico di occupazione di RAM. In particolare, volendo occupare gran parte della RAM richiesta dalla tipologia di VM, è stato deciso il valore dell'85%. Il restante 15% è stato lasciato unicamente per motivi di sicurezza.

Il Workload Starter viene eseguito dalle VM all'avvio del sistema. Ogni VM è stata configurata tramite uno *snapshot* appositamente realizzato.

4.3 Parametri di test

Con lo scopo di effettuare test sul framework integrato, sono state definite nuove tipologie di macchine virtuali, mostrate in tab. 4.5. Non sono state definite tipologie

	VCPU	RAM(GB)	DISK(GB)
Test.1	1	512	5
Test.2	1	1024	5
Test.3	2	1024	5
Test.4	2	2048	5
Test.5	4	2048	5
Test.6	4	4096	5

Tabella 4.5: Tipologie di macchine virtuali di test

con più di 4 VCPU, in quanto questo valore è già elevato essendo pari alla metà del numero di CPU in un nodo di computazione (ogni nodo è costuito da 4 CPU fisiche con *Hyper Threading*, per cui in totale risultano 8 CPU, tab. 4.3).

Esiste una sola tipologia con valore di RAM pari a 4GB, per evitare di saturare con poche VM i 16GB di un nodo. La RAM rappresenta il collo di bottiglia del cluster, come analizzato in [10].

Il disco è stato fissato ad una dimensione minima, non essendo utilizzato durante i test. In particolare, la dimensione deve essere maggiore o uguale a quella del disco da cui è stato effettuato lo snapshot iniziale, a partire da cui viene definita ogni VM. In generale, si è evitato di concentrare in poche VM troppa potenza computazionale, in relazione ai nodi di computazione.

I parametri su cui è stato effettuato il tuning sono i seguenti:

- Intervallo di raccolta dei campioni (par. *interval* nel file *pipeline.yaml*). È stato fissato a 150 secondi per avere un aggiornamento rapido sullo stato delle risorse.

- Intervallo di valutazione degli allarmi (par. *evaluation_interval* nel file *ceilometer.conf*). Sono state eseguite diverse prove con questo parametro. Per essere consistente, deve avere un valore superiore a quello dell'intervallo di raccolta. Come valore finale è stato scelto 200 secondi.

- Numero e durata dei periodi di valutazione delle metriche (par. *period* e *evaluation_period* nella definizione degli allarmi). Sono stati scelti 3 periodi, ciascuno della durata di 150 secondi. Analizzando le metriche per più di un periodo, ci si assicura che lo stato rilevato non sia un evento circoscritto, ma una tendenza del momento.

- Ripetizione delle azioni al termine del periodo dell'intervallo di valutazione degli allarmi (par. *repeat-actions* nella definizione degli allarmi). In caso sia abilitato, vengono inviate richieste ad ogni intervallo di valutazione, qualora la condizione definita dall'allarme sia ancora verificata. È fondamentale che sia attivo. In caso contrario, *l'alarm notifier* non invia ulteriori richieste relative ad un host che per intervalli di valutazione consecutivi venga rilevato in overload.

- Pesi delle metriche semplici *host.cpu.util* e *host.memory.usage* nella metrica combinata *host.cpu-util.memory.usage* (pesi nella definzione della metrica nel file *pipeline.yaml*). Sono stati posti a 0.5 entrambi, in modo da dare lo stesso peso ad entrambe le metriche. Il valore massimo raggiungibile dai campioni della metrica combinata è quindi pari ad 1.

- Soglia di underload (par. *threshold* nella definizione degli allarmi). È stato posto a 0.20, in relazione ad un valore massimo pari ad 1.

- Soglia di overload (par. *threshold* nella definizione degli allarmi). È stato posto a 0.75, in relazione ad un valore massimo pari ad 1.
- Numero totale di VM eseguite.
- Numero di VM per ogni tipologia mostrata in tab. 4.5.
- Algoritmo di selezione delle VM. Di base viene utilizzato il *Minimum Migration Time Maximum CPU Utilization* (par. 3.1.2), per minimizzare i tempi di migrazione.
- Algoritmo di piazzamento delle VM. Sono state eseguite prove con il *Best Fit Decreasing* (par. 3.1.2) usando sia la versione di base sia quella modificata.

4.4 Test

In questo paragrafo sono riportati i test eseguiti, con relativi risultati.

Test 1 - Importanza di *Repeat Actions*

In questo test sono stati utilizzati i parametri riportati in tab. 4.6. I parametri non specificati hanno il valore di default descritto in precedenza. Le VM sono state

Evaluation Interval	200s
Num VM	17
Num VM each type	6 1 4 4 2 0
Initial VMs Distribution	Start on 1 node
Repeat Actions	False

Tabella 4.6: Parametri Test 1

concentrate inizialmente tutte su un nodo; per poter raggiungere questo risultato, il servizio *nova-compute* è stato disattivato su due nodi (su un totale di tre) prima delle creazione delle istanze, per poi essere successivamente riattivato. Il numero di VM per ogni tipo è stato estratto casualmente. Il numero di VM di tipo 6 (tab. 4.5) è stato

forzato a zero, perchè questa tipologia occupa troppa RAM per essere posizionata su un singolo nodo. Il parametro *repeat_actions* è stato posto a false, di conseguenza per un host in stato di overload o underload, *l'alarm notifier* invia la relativa richiesta solo la prima volta in cui l'host viene rilevato in questo stato.

In fig. 4.1 è mostrata la situazione di ogni host, in termini di numero di VM. È pos-

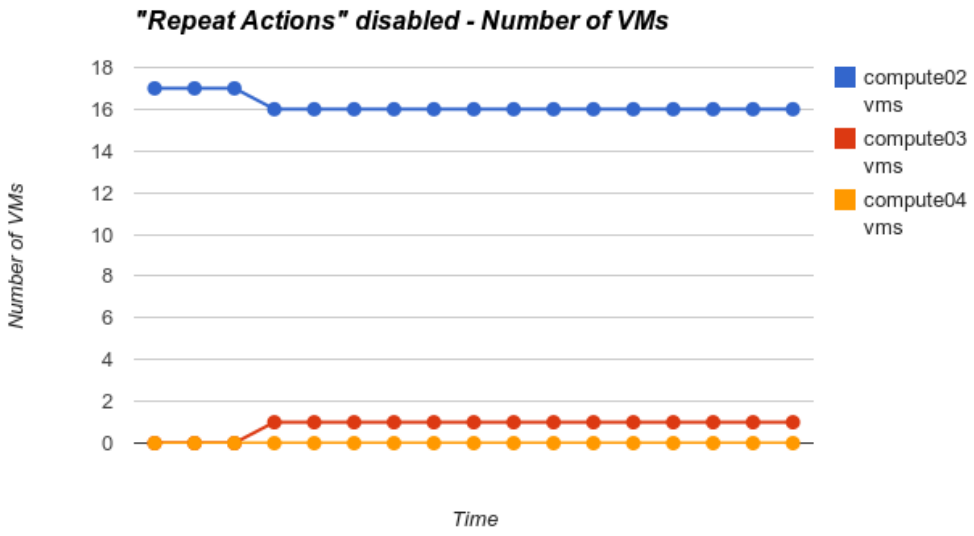

Figura 4.1: Repeat Actions disabilitato - Numero di VM

sibile notare come delle VM posizionate inizialmente sul nodo *compute02*, solo una venga migrata sul nodo *compute03*. In fig. 4.2 viene mostrato lo stato degli allarmi di ogni nodo; in particolare il valore "1" indica overload, "0" condizione normale, "-1" underload. A causa del concentramento delle VM sul'host *compute02*, il relativo allarme si trova perennemente in stato di overload. La fig. 4.3 mostra la causa degli allarmi. Per tutta la durata dei test, i campioni della metrica combinata, su cui sono stati definiti gli allarmi, hanno un valore superiore alla soglia di overload. Da questi risultati si dimostra l'importanza di abilitare il parametro *repeat_actions*.

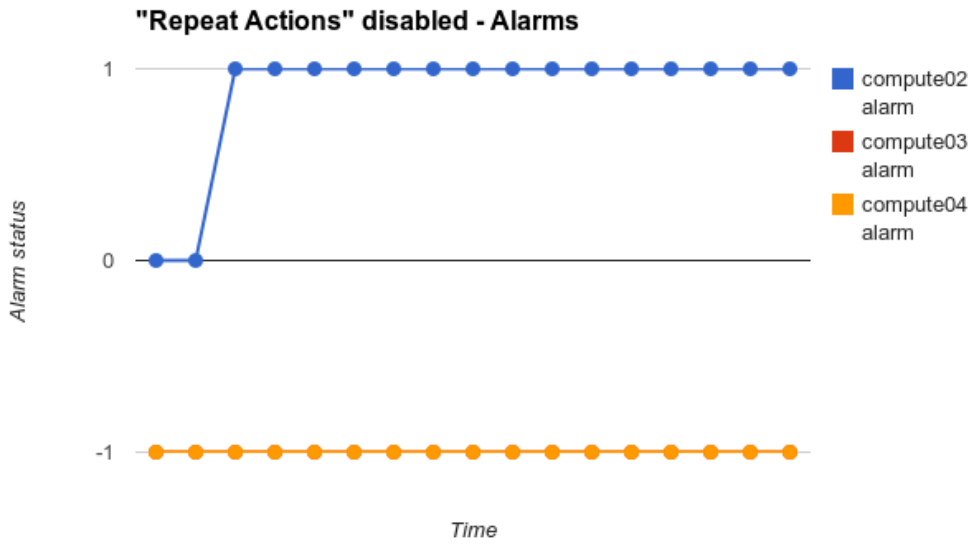

Figura 4.2: Repeat Actions disabilitato - Stato allarmi

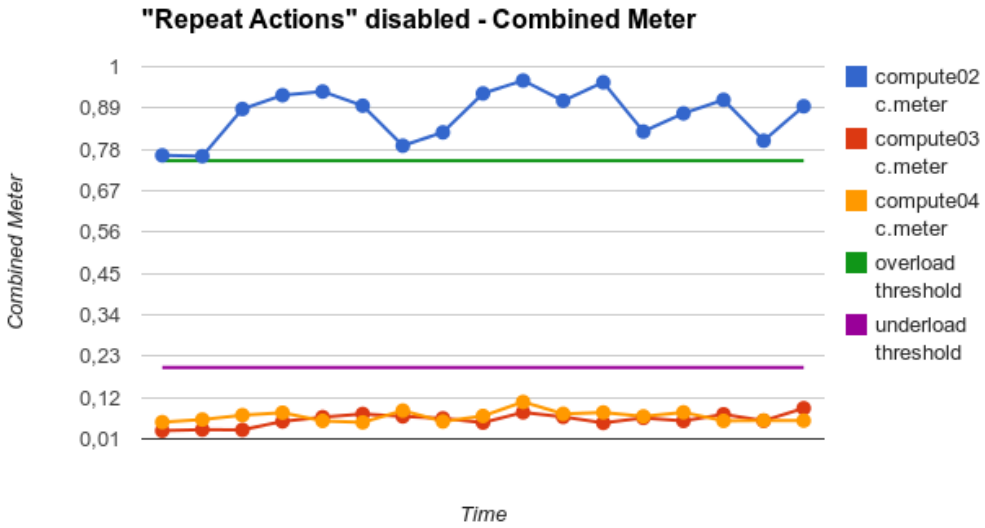

Figura 4.3: Repeat Actions disabilitato - Metrica combinata (CPU%,RAM%)

Test 2 - Importanza di *Evaluation Interval*

I parametri utilizzati per questo test sono riportati in tab. 4.7. Il numero di VM per

Num VM	20
Num VM each type	4 3 6 2 3 2
Data Collection Interval	150s
Evaluation Interval	60s
Initial VMs Distribution	Start on 3 node
Repeat Actions	True

Tabella 4.7: Parametri Test 2

ogni tipo è stato estratto casualmente. Questa distribuzione verrà utilizzata anche in test successivi. A differenza del test precedente, le VM sono distribuite inizialmente su tutti i tre nodi di computazione. La disposizione è stata definita dallo scheduler di Nova. In questo test si vuole evidenziare l'importanza della scelta dell'intervallo di valutazione degli allarmi; in particolare, utilizzare un valore inferiore all'intervallo di collezione dei campioni comporta la valutazione di un allarme sulla base degli stessi campioni dell'intervallo di valutazione precedente, ossia su dati passati; un host verrebbe visto quindi consecutivamente in stato di overload/underload, senza aspettare la collezione di nuovi dati.

In fig. 4.4 è riportata la disposizione nel tempo delle VM per ogni host. È possibile notare come non si raggiunga uno stato stazionario, ma le VM continuano a migrare da un host ad un altro. Questa situazione è conseguenza di un valore del parametro *evaluation_interval* minore del valore di *collection_interval*, che comporta l'invio multiplo di richieste per la gestione di eventi di overload/underload. Questo comportamento è ben evidenziato dallo stato degli allarmi di ogni host, mostrato in fig. 4.5, in cui gli host si alternano nello stato di overload. Il valore dei campioni della metrica combinata sono mostrati in fig. 4.6. La soglia di overload viene oltrepassata in maniera alternata dai campioni di ciascun host. I risultati di questo test sottolineano come

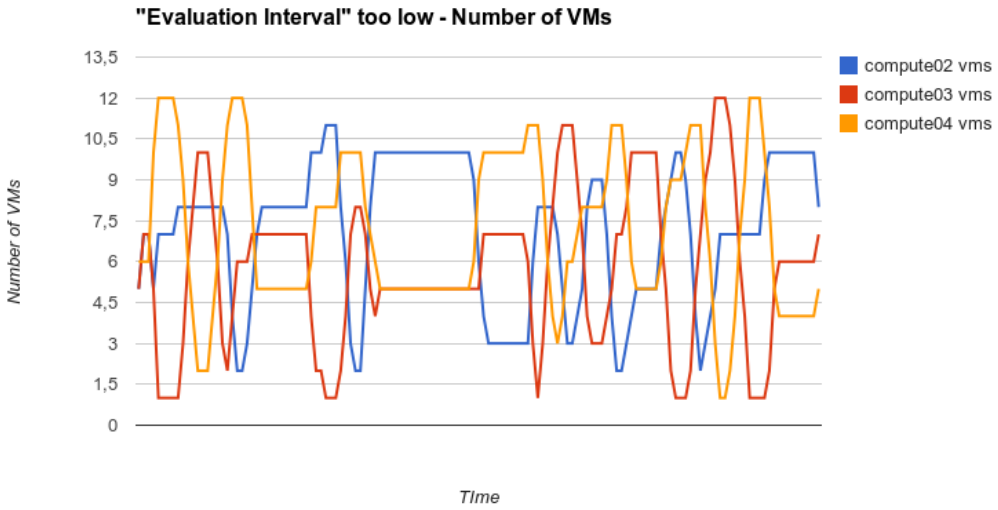

Figura 4.4: Evaluation Interval troppo basso - Numero di VM

la scelta di un valore di *evaluation_interval* minore del valore di *collection_interval*, porti il sistema ad essere totalmente instabile.

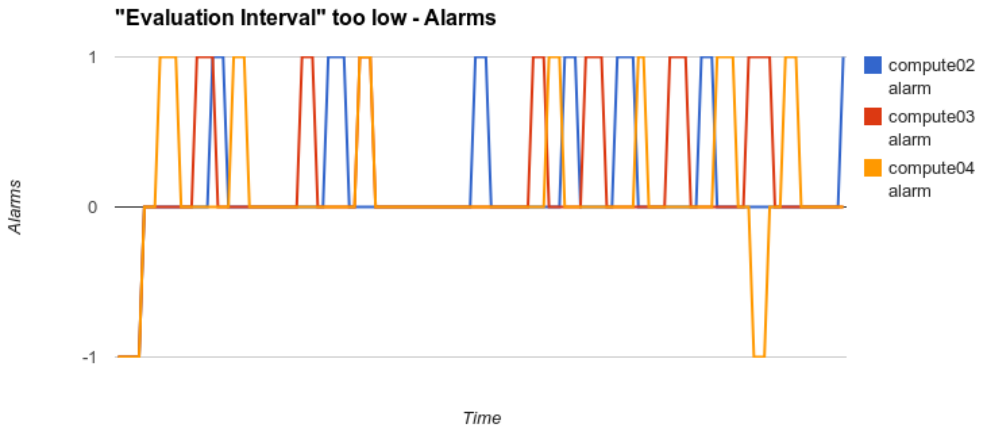

Figura 4.5: Evaluation Interval troppo basso - Stato allarmi

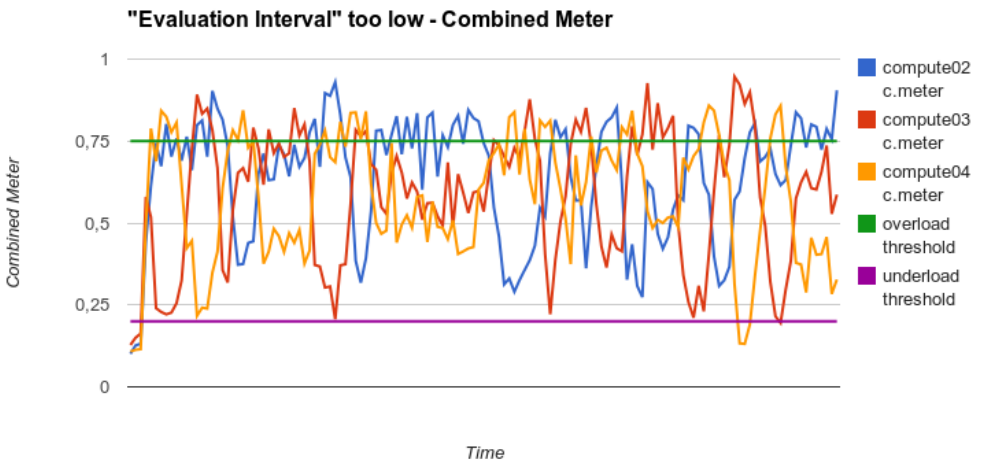

Figura 4.6: Evaluation Interval troppo basso - Metrica combinata (CPU%,RAM%)

Test 3 - Confronto *Best Fit Decreasing* Normale e Modificato

In questo test si mettono a confronto i risultati ottenuti con la versione di base e quella modificata dell'euristica per il *Bin Packing*, l'algoritmo per il posizionamento delle VM selezionate per la migrazione. I parametri utilizzati sono riportati in tab. 4.8. All'avvio del test le VM sono state posizionate tutte su un singolo host. È stato

Evaluation Interval	200s
Num VM	17
Num VM each type	6 1 4 4 2 0
Initial VMs Distribution	Start on 1 node
Repeat Actions	True
Best Fit Decreasing	Modified vs Normal

Tabella 4.8: Parametri Test 3

quindi analizzata la distribuzione sui restanti nodi.

In relazione al numero di VM per host, in fig. 4.7 è mostrato il confronto fra i due algoritmi. I grafici mettono in evidenza un problema legato all'algoritmo originale; gli host vengono valutati unicamente sulla base della relativa disponibilità di CPU e RAM, senza prendere in considerazione la possibilità che un ulteriore migrazione sull'host potrebbe portarlo in stato di overload. Per questo motivo, il numero di migrazioni è maggiore durante un'esecuzione dell'algoritmo base in confronto all'algoritmo modificato.

La maggiore stabilità dell'algoritmo modificato è ben visibile anche nella fig. 4.8, in cui il confronto avviene sul piano dei campioni della metrica combinata. In particolare la soglia di overload viene oltrepassata più volte nell'esecuzione dell'algoritmo base, causando un maggior numero di migrazioni.

È importante sottolineare come il terzo nodo non venga mai utilizzato in questo caso d'uso. Questo perchè il Best Fit Decreasing viene utilizzato con lo scopo del **Consolidamento di VM**, e non del bilanciamento. Poichè due nodi riescono a contenere

tutte le VM del cluster, senza entrare entrambi in stato di overload, il terzo nodo viene lasciato in stato inattivo.
Confrontando i risultati dei due algoritmi in relazione alla percentuale di tempo passata da ogni nodo in stato di overload e underload, è possibile concludere che:

- Il nodo *compute02* viene rilevato in overload per il 29% del tempo dall'algoritmo base. L'algoritmo modificato rileva un percentuale di overload pari al 10%. Essendo questo il nodo su cui tutte le VM sono posizionato all'avvio del test, non viene mai rilevato in stato di underload.

- Il nodo *compute03* viene rilevato in overload per il 17% del tempo e in underload per il 12% del tempo dall'algoritmo base. L'algoritmo modificato rileva una percentuale del 8% per entrambi gli stati.

- Il nodo *compute04* si trova perennemente in stato di underload durante l'esecuzione di entrambi gli algoritmi.

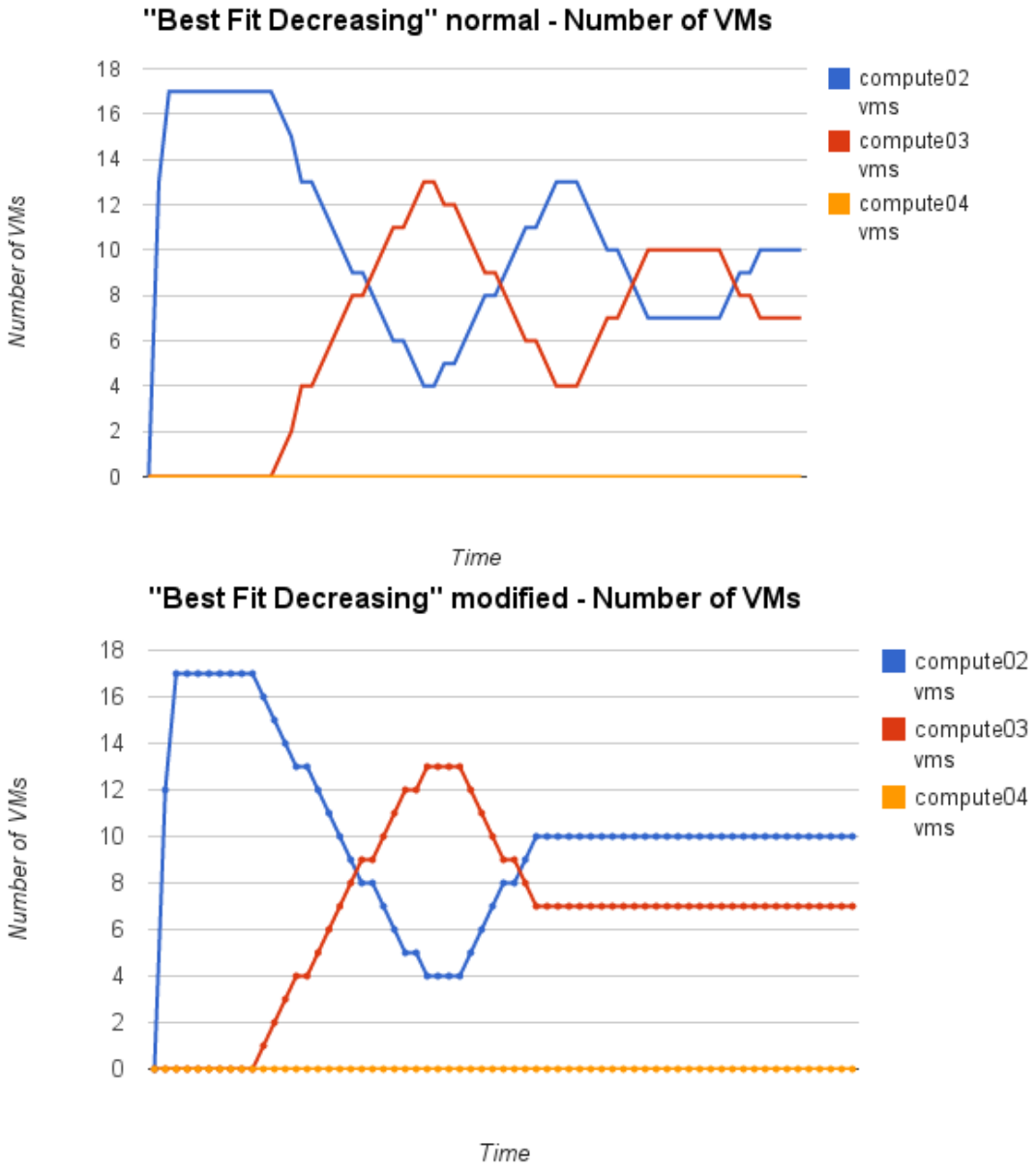

Figura 4.7: Best Fit Decreasing: confronto versioni - Numero di VM

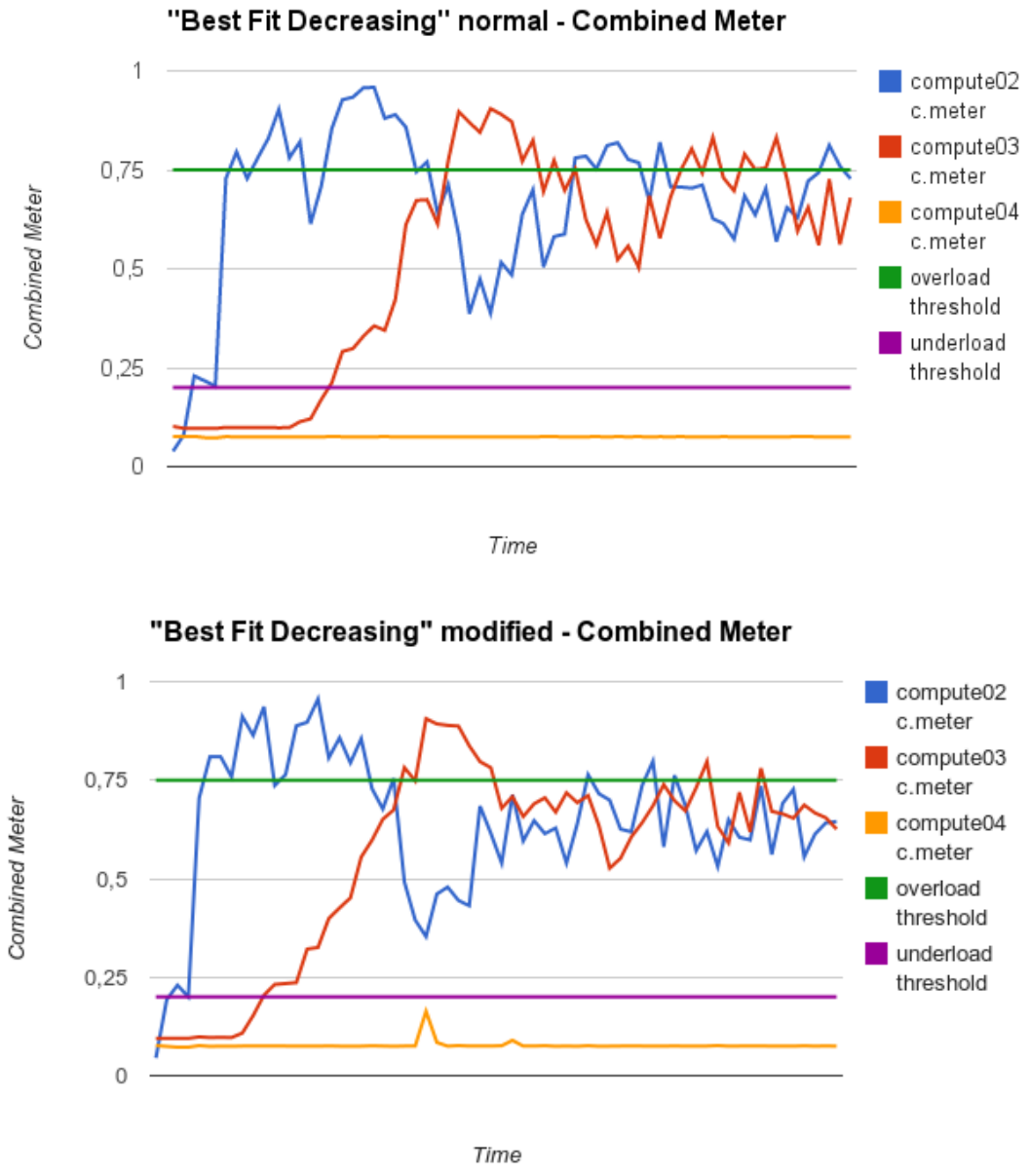

Figura 4.8: Best Fit Decreasing: confronto versioni - Metrica Combinata (CPU%,RAM%)

Test 4 - Esecuzione di OpenStack Neat con parametri ottimali

In questo test è stata effettuata un'esecuzione di OpenStack Neat utilizzando i parametri ottimali ricavati nei test precedenti, riportati in tab. 4.9. In fig. 4.9 è riportato

Evaluation Interval	200s
Num VM	20
Num VM each type	4 3 6 2 3 2
Initial VMs Distribution	Start on 3 node
Repeat Actions	True
Best Fit Decreasing	Modified

Tabella 4.9: Parametri Test 4

un grafico relativo alla disposizione delle VM sui vari host. È possibile notare come il sistema a regime tenda a stabilizzarsi. Nelle fig. 4.10, 4.11 e 4.12 sono riportati

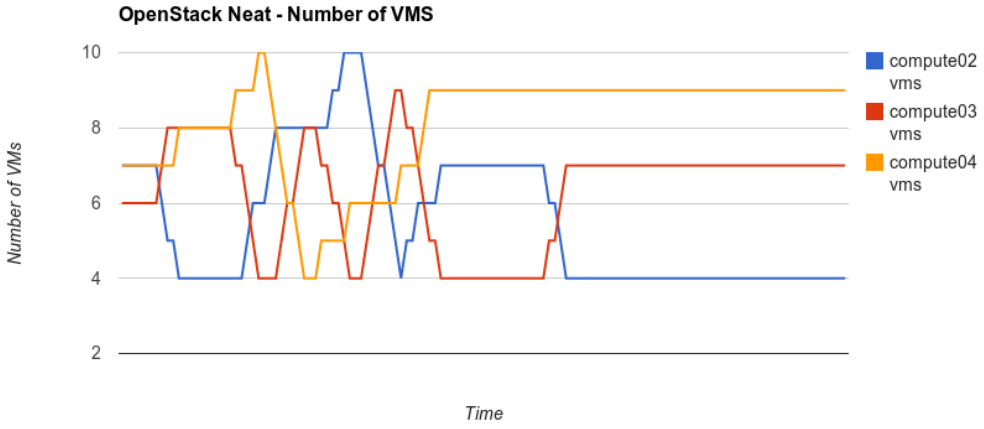

Figura 4.9: OpenStack Neat: parametri ottimali - Numero di VM

per ogni nodo di computazione i seguenti valori: numero di VM, stato degli allarmi, campioni della metrica combinata, campioni di utilizzo CPU e campioni di utilizzo RAM.

In ogni grafico è possibile notare l'occorenza degli allarmi di overload al superamento della soglia di 0.75 in relazione ai campioni della metrica combinata. Allo scattare

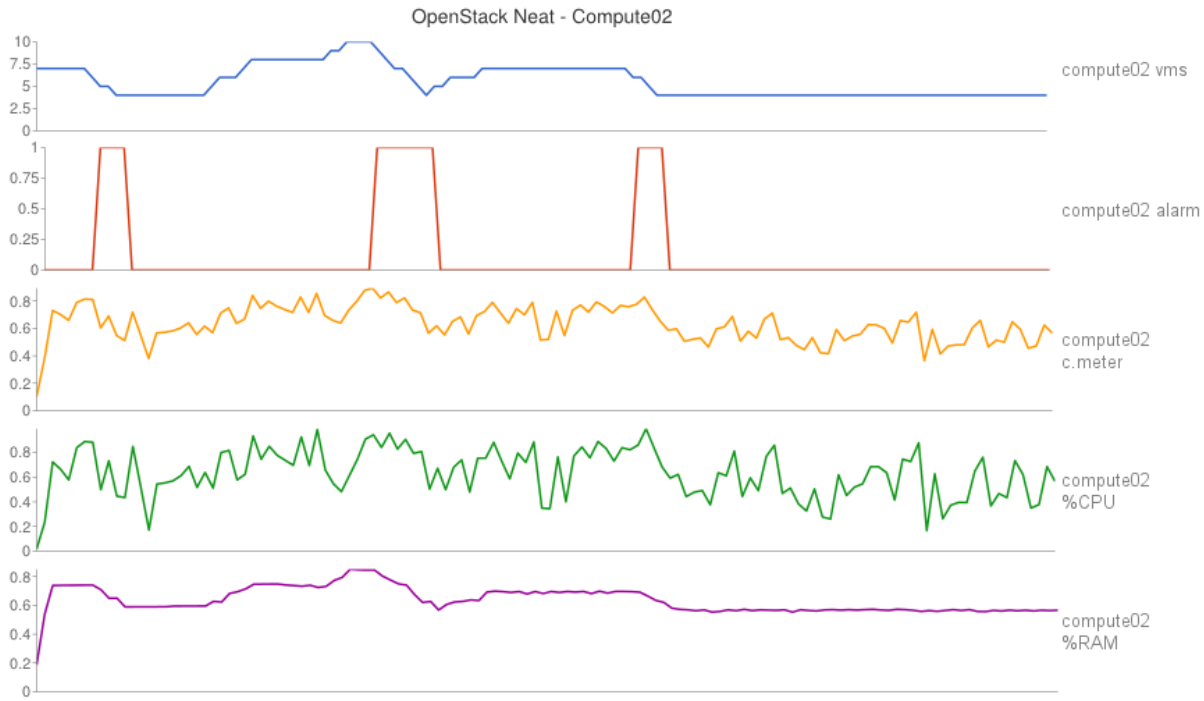

Figura 4.10: OpenStack Neat: parametri ottimali - Risultati Compute02

di un allarme di overload il numero di VM diminuisce. Si sottolinea inoltre come il grafico dell'utilizzo di RAM sia del tutto analogo a quello del numero di VM, in quanto l'utilizzazione di RAM è stata posta staticamente al valore di 85%, per le considerazione fatte nel par. 4.3.

Dai risultati si possono estrapolare i seguenti dati:

- L'host *Compute02* è stato rilevato in overload per il 12% del tempo totale di test e mai in underload.
- L'host *Compute03* è stato rilevato in overload per il 14% del tempo totale di test e mai in underload.
- L'host *Compute04* è stato rilevato in overload per il 7% del tempo totale di test e mai in underload.

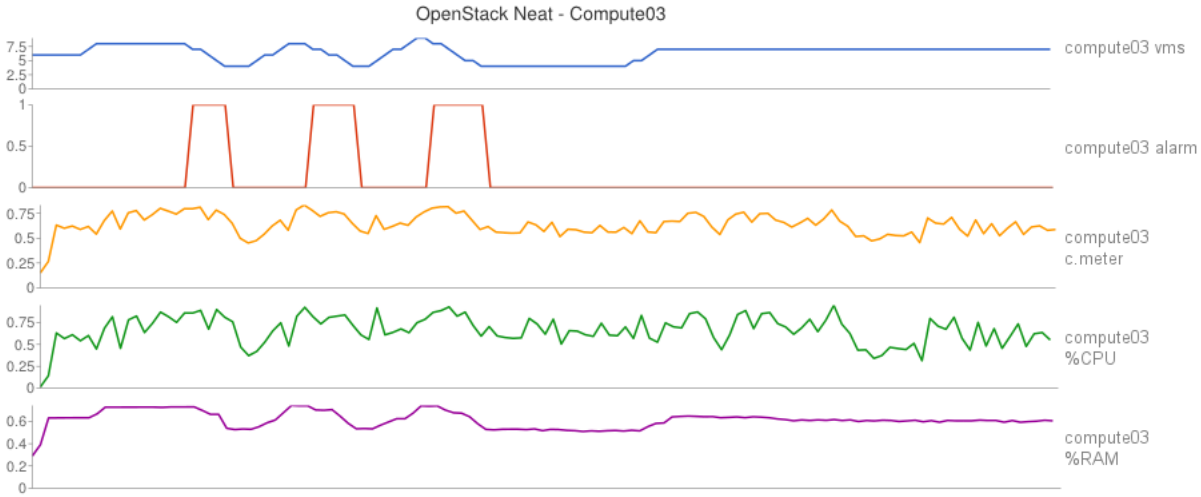

Figura 4.11: OpenStack Neat: parametri ottimali - Risultati Compute03

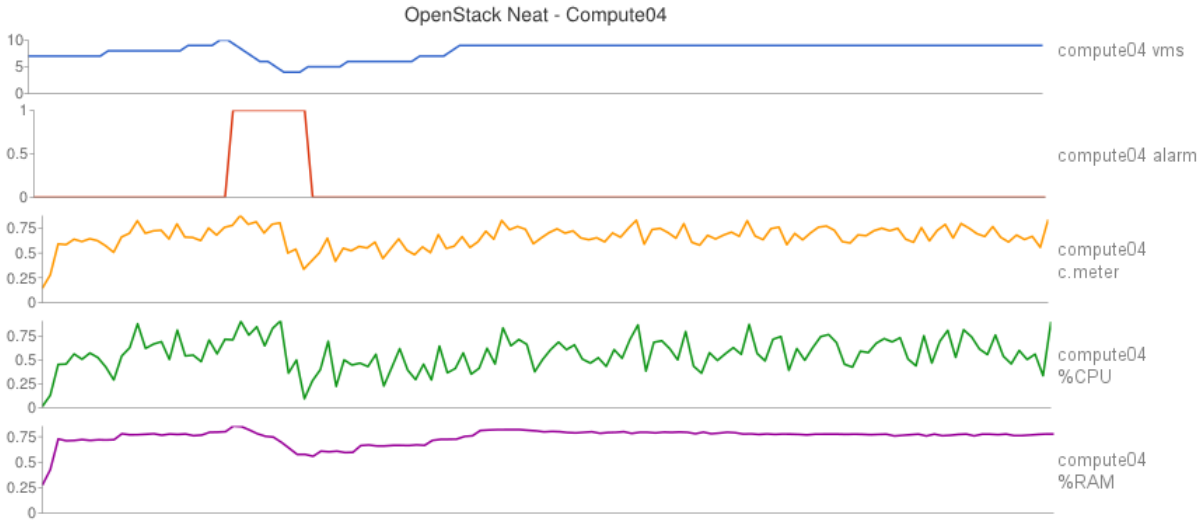

Figura 4.12: OpenStack Neat: parametri ottimali - Risultati Compute04

Test 5 - Esecuzione con OpenStack Neat disabilitato

Come ultimo test, è stata effettuata un'esecuzione disabilitando totalmente OpenStack Neat. Gli allarmi sono stati mantenuti, ma alla rilevazione di uno stato di overload/underload nessun componente era in azione ad eseguire politiche di migrazione. I parametri sono mostrati in tab. 4.10. Il numero e il tipo di VM per ogni

Num VM	20
Num VM each type	4 3 6 2 3 2
Initial VMs Distribution	Start on 3 node
VMs on Compute02	8 (types: 6 5 5 4 3 3 2 1)
VMs on Compute03	6 (types: 6 4 3 3 2 1)
VMs on Compute04	6 (types: 5 3 3 2 1 1)

Tabella 4.10: Parametri Test 5

host sono fissati inizialmente dallo scheduler di Nova e non vengono mai modificati per l'assenza di migrazioni.

In fig. 4.13, sono riportati i valori dei campioni della metrica combinata. L'host *com-*

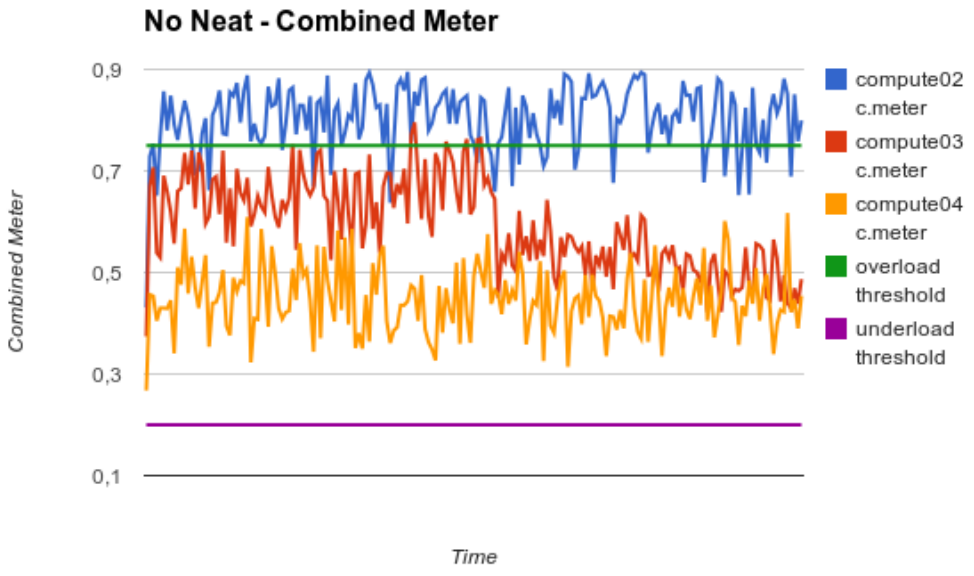

Figura 4.13: OpenStack Neat disattivato - Metrica Combinata (CPU%,RAM%)

pute02 si trova perennemente in stato di overload, essendo i campioni della metrica derivata superiori alla soglia di overload.
Dai risultati è possibile estrapolare i seguenti dati:

- L'host *Compute02* è stato rilevato in overload per l'89% del tempo totale di test e mai in underload.
- L'host *Compute03* non viene mai rilevato in stato di overload o underload.
- L'host *Compute04* non viene mai rilevato in stato di overload o underload.

Capitolo 5

Conclusioni e Sviluppi Futuri

In questo lavoro di tesi è stato realizzato un framework per la gestione di politiche di migrazione di macchine virtuali (VM), con particolare focus sul consolidamento. Questo framework è nato come estensione di OpenStack Neat, un progetto per il consolidamento di VM in OpenStack. Neat rileva lo stato di un host basandosi unicamente sui dati collezionati sull'utilizzo di CPU. Inoltre, per la raccolta dei dati utilizza un servizio realizzato ad hoc, anziché appoggiarsi a Ceilometer, il progetto di OpenStack con queste funzionalità.

Partendo da questa analisi, si è proceduto con l'obiettivo di realizzare politiche di migrazione più complesse, tramite il monitoraggio di un numero maggiore di metriche. Si è quindi deciso di integrare Ceilometer in Neat, aumentando notevolmente il numero di metriche disponibili. In seguito, si è proceduto con la definizione di una metrica composta da una combinazione lineare di utilizzazione della CPU e della RAM. Ceilometer nella versione Juno ha introdotto un meccanismo per la combinazione di metriche. Questo meccanismo presenta però dei limiti, rilevati durante il suo utilizzo. Innanzitutto, è stato identificato e segnalato alla community un bug, per cui è stato fornito supporto alla risoluzione. Inoltre, è stato necessario modificare in maniera importante il meccanismo di collezione, trasformazione e pubblicazione dei campioni

di metriche.

In parallelo alla realizzazione del framework, si è provveduto all'installazione e alla configurazione di un ambiente con OpenStack. Inizialmente, appoggiandosi su Amazon Web Services e Digital Ocean, è stato realizzato un cloud interamente virtuale. In seguito, si è potuto procedere su un cluster di macchine fisiche localizzato nell'Università di Tor Vergata. Sono stati quindi eseguiti una serie di test che hanno posto in risalto il funzionamento del framework, e i parametri ottimali nell'ottica del consolidamento di VM. Il confronto con un'esecuzione priva di Neat, ha messo in evidenza l'utilità di questo framework nella gestione di un'infrastruttura cloud.

Il framework realizzato è facilmente estendibile. In particolare, il framework si basa sulla risoluzione di quattro problemi: rilevamento di overload, rilevamento di underload, selezione delle VM da migrare e posizionamento delle stesse sugli host. Partendo da questo presupposto, un primo sviluppo potrebbe consistere nella definizione di algoritmi più complessi per la risoluzione dei suddetti problemi.

Per la rilevazione dello stato di un host, si potrebbe utilizzare una combinazione più ampia di metriche, in quanto il meccanismo realizzato non richiede modifiche per essere applicato in tal senso. Allo scopo, l'utilizzo del framework Kwapi, tramite l'installazione di Wattmetri nel cluster, fornirebbe informazioni interessanti sul consumo energetico dei nodi. Altro sviluppo interessante, sempre sul fronte del rilevamento dello stato di un host, consiste nello sfruttare il meccanismo dell'*overcommit*; di base OpenStack permette il posizionamento su una macchina fisica, di un numero di VM superiore alle risorse disponibili. Questo non risulta un problema se queste macchine non utilizzano per intero le risorse allocate. Tramite un monitoraggio delle risorse effettivamente utilizzate dalle singole VM, si potrebbero effettuare migrazioni al momento in cui le risorse fisiche fossero effettivamente per terminare. Per lo scopo, il

monitoraggio della memoria effettivamente utilizzata da una VM costituisce un problema la cui risoluzione è in atto nella versione di OpenStack attualmente in sviluppo, OpenStack *Kilo*.

Per quanto riguarda gli algoritmi di selezione delle VM, si potrebbero utilizzare anche in questo caso informazioni su un numero maggiore di metriche. Lo sfruttamento di tecniche di *Machine Learning* potrebbe fornire informazioni per la previsione del comportamento futuro di una VM. Queste informazioni potrebbero essere utilizzate sia nella selezione che nel posizionamento delle VM. Il posizionamento delle VM potrebbe avvenire anche tenendo conto della tipologia di risorse consumate maggiormente da una VM. Macchine virtuali che nella storico risultino essere Memory-Intensive, non dovrebbero essere migrate sullo stesso host, o l'host target dovrebbe utilizzare una memoria particolarmente veloce. Per lo scopo, il meccanismo di Nova degli *Host Aggregates*, che permette di raggruppare host secondo criteri arbitrari, tornerebbe sicuramente utile.

Un ulteriore interessante sviluppo è relativo alla scalabilità del framework. Mentre gli algoritmi per il rilevamento dello stato di un host e la selezione delle VM da migrare possono essere eseguiti su repliche in maniera indipendente, l'algoritmo di piazzamento delle VM è intrinsecamento centralizzato. L'esecuzione parallela su più repliche, senza collaborazione fra le stesse, porterebbe ad una visione non aggiornata sullo stato dei nodi. La realizzazione di un algoritmo di posizionamento totalmente distribuito, che preveda collaborazione fra le varie repliche, aumenterebbe la scalabilità del sistema.

Infine, un possibile sviluppo consiste nell'aumentare la collaborazione fra il framework e gli altri componenti di OpenStack. In particolare, la definizione di nuovi algoritmi di scheduling in Nova potrebbe fornire supporto al framework nel consolidamento di VM.

Elenco delle figure

Bibliografia

[1] S. Hand, T. L. Harris, E. Kotsovinos, I. Pratt.
"*Controlling the XenoServer Open Platform*"
in *Proc. of OPENARCH 2003*, 2003.

[2] R. Bradford, E. Kotsovinos, A. Feldmann, H. Schiöberg
"*Live Wide-Area Migration of Virtual Machines Including Local Persistent State*"
in *Proc. of VEE 2007*, pp.169-179, *ACM*, 207.

[3] D. Erickson, G. Gibb, B. Heller, D. Underhill et al.
"*A Demonstration of Virtual Machine Mobility in an OpenFlow Network*"
in *Proc of ACM SIGCOMM 2008 (Demo). ACM*, 2008.

[4] T. Wood, P. Shenoy, K.K. Ramakrishnan, J. Van der Merwe
"*CloudNet: Dynamic Pooling of Cloud Resources by Live WAN Migration of Virtual Machines*"
in *Proc. of VEE 2011*, pp.121-132, *ACM*, 2011.

[5] U. Kalim, M. K. Gardner, E. J. Brownent, W. Feng
"*Seamless Migration of Virtual Machines Across Networks*"
in *Proc. of IEEE ICCCN*, 2013.

[6] M. Mishra, A. Das, P. Kulkarni, A. Sahoo

"*Dynamic Resource Management Using Virtual Machine Migrations*"

in *IEEE Communications Magazine*, vol.50, no.9, pp.34-40, September 2012.

[7] A. Beloglazov, R. Buyya

"*OpenStack Neat: A Framework for Dynamic and Energy-Efficient Consolidation of Virtual Machines in OpenStack Clouds*"

in *Concurrency and Computation: Practice and Experience*, *accepted for publication*, 2014.

[8] F. Rossigneux, J.P. Gelas, L. Lefèvre, M.D. de Assunção

"A Generic and Extensible Framework for Monitoring Energy Consumption of OpenStack Clouds"

in *Proc. of 4th IEEE Int. Conference on Sustainable Computing and Communications*, 2014

[9] D. Meisner, B. Gold, T. Wenisch

"PowerNap: eliminating server idle power"

in *ACM SIGPLAN Notices-ASPLOS 2009*, Vol.44, No.3, March 2009.

[10] A. Mercanti

"Analisi delle prestazioni della piattaforma OpenStack"

Tesi di laurea in Ingegneria Informatica A.A. 2013/2014, Università di Roma Tor Vergata, 2014.

[11] A. Fuggetta, G. P. Picco, G. Vigna

"Understanding Code Mobility"

in *IEEE Transactions on Software*, Vol.24, No. 5, Maggio 1998

[12] JAIN, R.; Paul, S.

"Network virtualization and software defined networking for cloud computing: a survey,"

in *IEEE Communications Magazine*, vol.51, no.11, pp.24,31, November 2013

[13] Wood, T.; Ramakrishnan, K.K.; Shenoy, P.; van der Merwe, J.; Hwang, J.; Liu, G.; Chaufournier, L.

"CloudNet: Dynamic Pooling of Cloud Resources by Live WAN Migration of Virtual Machines"

in *IEEE/ACM Transactions on Networking*, *accepted for publication*, 2014

[14] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum

"Optimizing the Migration of Virtual Computers"

in *Proc. of USENIX OSDI*, 2002

[15] A. Corradi, M. Fanelli, L. Foschini

"VM consolidation: A real case based on OpenStack Cloud"

in *Future Generation Computer Systems*, Vol.32, pp. 118-127, March 2014

[16] Carl A. Waldspurger

"Memory resource management in VMware ESX server"

in *SIGOPS Oper. Syst. Rev. 36, SI*, 181-194, December 2002

[17] Hypervisor supportati da Openstack e relative caratteristiche

`http://docs.openstack.org/developer/nova/support-matrix.html`

[18] Elenco caratteristiche Hyper-V

`http://technet.microsoft.com/it-it/library/hh831531.aspx#BKMK_NEW`

[19] Supporto alla migrazione inter-cluster in VMware

http://www.vmware.com/products/vsphere/features/vmotion.html

[20] Segnalazione bug nella gestione degli arithmetic transformer da parte del Compute Agent

http://bugs.launchpad.net/ceilometer/+bug/1394228

[21] Elenco metriche disponibili in Ceilometer

http://docs.openstack.org/developer/ceilometer/measurements.html

[22] Documentazione Ceilometer

http://docs.openstack.org/developer/ceilometer/

[23] Documentazione Kwapi

http://kwapi.readthedocs.org

[24] Documentazione Openstack, versione Juno, per Ubuntu 14.04

http://docs.openstack.org/juno/install-guide/install/apt/content/

[25] Elenco immagini disponibili per macchine virtuali in OpenStack

http://docs.openstack.org/image-guide/content/ch_obtaining_images.html

[26] Configurare migrazione in OpenStack

http://docs.openstack.org/admin-guide-cloud/content/section_configuring-compute-migrations.html

[27] Tracce di utilizzo CPU di macchine virtuali in PlanetLab

http://github.com/beloglazov/planetlab-workload-traces

[28] Lookbusy, generatore di carico sintetico per CPU, RAM e Disco

`http://www.devin.com/lookbusy`

[29] OpenFlow

`http://archive.openflow.org`

[30] OpenStack

`http://www.openstack.org/`

[31] Openstack Neat

`http://openstack-neat.org/`

[32] Amazon Web Services

`http://aws.amazon.com/`

[33] Amazon Web Services - EC2 Instances

`http://aws.amazon.com/ec2/instance-types`

[34] Digital Ocean

`https://www.digitalocean.com`

[35] DevStack

`http://docs.openstack.org/developer/devstack/`