
SAPIENZA
UNIVERSITÀ DI ROMA

FACOLTA' DI INGEGNERIA DELL'INFORMAZIONE,
INFORMATICA E STATISTICA

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

Tesi di Laurea Magistrale

# Tool per il benchmarking e la gestione autonomica di piattaforme datagrid transazionali

**Laureando**

Fabio Perfetti

**Relatore**

Prof. Francesco Quaglia

Anno Accademico 2012/2013

*Alla mia famiglia*

# RINGRAZIAMENTI

Ad un passo dalla sperata Laurea Magistrale, credo che sia d'obbligo spendere qualche parola per ringraziare le persone che da sempre sanno starmi affianco ed aiutarmi nelle scelte.
Innanzitutto, vorrei ringraziare tutti i membri del gruppo di ricerca di Sistemi Distribuiti del INESC-ID di Lisbona ed in particolare il Prof. Paolo Romano e Diego Didona i quali mi hanno aiutato quotidianamente a portare a termine il lavoro; allo stesso modo, ringrazio il Prof. Francesco Quaglia, mio relatore, per avermi accettato come suo laureando e per essersi mostrato sempre disponibile nei miei confronti. Grazie a loro ho potuto trascorrere una utile esperienza all'estero.
Ringrazio nuovamente Paolo per aver svolto in quest'ultimo anno il ruolo di amico, oltre a quello di supervisore, condividendo insieme avventurosi [1] surf trip e *gostose* surf session, accompagnate da dolci venti off-shore.
Senza ombra di dubbio, un caloroso ringraziamento và alla mia famiglia, a cui dedico questo lavoro e alla quale devo molti sacrifici, tutti per il mio bene. È grazie ai loro se sono riuscito a raggiungere questa importante tappa, (spero) fondamentale per il mio futuro. Ovviamente, non mi limito al nucleo familiare, bensì tutte quelle persone che quotidianamente passano nel mio *studio* solo per darmi un abbraccio o per invitarmi a cena.
Non possono mancare gli amici, i miei amici, genitori e non, per essermi stati sempre affianco, anche solo con un messaggio (ancora meglio se con

[1] perdere ripetutamente le chiavi della macchina (e di casa) a più di 100 Km da casa!!

un paio di birre!!).
Chiudo subito e vi lascio alla mia commedia, divina ovviamente, ma prima voglio e devo ringraziare colei che ha saputo aspettarmi portando (forse) fin troppa pazienza: la mia ragazza!

# ABSTRACT

*Il Cloud Computing è una tecnologia emergente e sempre più diffusa. La sua natura elastica, impattando drasticamente sugli attuali schemi di fornitura delle risorse, rende ragione dell'ampio consenso ottenuto: le risorse sono distribuite elasticamente, su richiesta, con una apparentemente illimitata quantità di potenza computazionale e di storage disponibile. Alla natura elastica delle piattaforme cloud, è associato un modello di pricing di tipo pay-only-for-what-you-use; ciò garantisce la riduzione dei costi ed efficienza. Allo stesso modo, anche le memorie transazionali distribuite hanno guadagnato un crescente interesse, destando l'attenzione di gruppi di ricerca in tutto il mondo, i quali hanno iniziato a disegnare algoritmi distribuiti e innovativi, capaci di assicurare semantica di consistenza transazionale in maniera efficace e scalabile. Aspetti di attuale interesse, nell'ambito delle memorie transazionali distribuite, sono come supportare il ridimensionamento dinamico, noto in letteratura come elastic scaling, di una piattaforma e il self-tuning dei suoi schemi di replicazione; questi ultimi giocano un ruolo essenziale per garantire la durabilità dei dati a fronte di guasti. Il lavoro di tesi è stato svolto sotto tali aspetti, ed ha permesso di raggiungere due fondamentali obiettivi: è stato disegnato e sviluppato* ELASTIC DTM BENCH, *un framework per il benchmarking di piattaforme transazionali distribuite elastiche; è stato sviluppato il componente Autonomic Manager, presente all'interno dell'architettura del progetto europeo Cloud-TM, capace di intraprendere, autonomamente, e gestire attività di riconfigurazione di piattaforme transazionali distribuite, con lo scopo di massimizzare le prestazioni e ridurre i costi operazionali della piattaforma.*

# INDICE

# CAPITOLO 1

# INTRODUZIONE

Il Cloud Computing è una tecnologia emergente e sempre più diffusa. A dimostrarlo è, non solo l'adesione ad esso da parte di grandi e piccole aziende, ma anche l'uso diffuso tra gli utenti per svolgere semplici attività quotidiane come ad esempio la scrittura di un documento di testo.
La natura elastica del cloud computing, impattando drasticamente sugli attuali schemi di fornitura delle risorse, rende ragione dell'ampio consenso ottenuto: le risorse sono distribuite elasticamente, su richiesta, con una apparentemente illimitata quantità di potenza computazionale e di storage disponibile. Alla natura elastica delle piattaforme cloud, è associato un modello di *pricing* di tipo *pay-only-for-what-you-use*; ciò garantisce la riduzione dei costi, rilasciando le risorse quando non utilizzate, ed efficienza.
Dal punto di vista del programmatore, l'adesione al cloud computing ha richiesto lo sviluppo di applicazioni distribuite ideate appositamente per funzionare in ambienti dinamici, elastici e altamente *fault-prone*.
In risposta a tali esigenze, le piattaforme transazionali distribuite *in-memory*, quali *data-grid NoSql* e memorie transazionali software, hanno guadagnato una forte rilevanza. Esse appaiono, nell'ambito della comunità di ricerca, un paradigma di programmazione particolarmente interessante per facilitare lo sviluppo di applicazioni distribuite e concorrenti, fornendo l'accesso a stati condivisi attraverso transazioni. In tale modo, è possibile liberare il programmatore dalla gestione di problematiche introdotte dall'uso di

primitive esplicite di sincronizzazione tra *thread* basate su *lock*, quali *deadlock* distribuiti, e permettendo di incrementare la produzione, ridurre i tempi di sviluppo e aumentare l'affidabilità del codice in applicazioni concorrenti complesse.
L'evoluzione distribuita delle memorie transazionali ha inoltre abilitato lo sviluppatore a delegare, ad un componente *middleware* distribuito, la gestione e risoluzione di problematiche riguardanti la concorrenza, *fault-tolerance*, trasferimento dello stato tra nodi, ecc. .
Aspetti di attuale interesse, nell'ambito delle memorie transazionali distribuite, sono come supportare il ridimensionamento dinamico, noto in letteratura come *elastic scaling*, di una piattaforma e il *self-tuning* dei suoi schemi di replicazione, questi hanno un ruolo essenziale come mezzo principale per garantire la durabilità dei dati a fronte di guasti. Entrambi gli aspetti aiutano a garantire le qualità del servizio e, allo stesso tempo, a minimizzare i costi operazionali.
Effettuare elasting scaling, all'interno di ambienti cloud, significa determinare la configurazione della piattaforma più *cost-effective*, in termini di numero di nodi computazionali e loro tipo (configurazione hardware adottata dai nodi). Raggiungere questo scopo è però tutt'altro che facile poichè richiede di affrontare alcuni problemi di difficile risoluzione, quali:

- predirre le performance di una applicazione basata su memoria transazionale distribuita in funzione su un insieme di nodi e/o nodi con caratteristiche computazionali eterogenee;
- minimizzare gli *overhead* associati alle riconfigurazioni del sistema, i quali, nel caso di piattaforme transazionali distribuite, possono richiedere onerose fasi per trasferire lo stato tra i nodi, per garantire la coerenza del sistema;
- identificare adeguati *trade-off* tra reattività e robustezza di controllori responsabili di determinare la dimensione della piattaforma, in modo da raggiungere un bilancio tra efficienza del sistema (data dall'abilità del controllore di rispondere in tempo a variazioni del carico di lavoro), e sua stabilità (tenendo in conto che politiche eccessivamente reattive potrebbero condurre il sistema in uno stato di *thrash*, se le riconfigurazioni venissero innescate frequentemente).

Effettuare self-tuning di schemi di replicazione, all'interno di ambienti cloud, significa invece determinare il grado, cioè il numero di repliche presenti nella piattaforma, e il protocollo di replicazione più adatti per raggiungere un livello prestazionale ottimale. I problemi della replicazione transazionale sono stati largamente studiati, sia nell'ambito dei database classici, sia nei sistemi di memorie transazionali. Come risultato, sono stati proposti un largo numero di protocolli di replicazione basati su diversi principi di design, come schemi di gestione delle transazioni di update single-master vs multi-master, serializzazioni di transazioni lock-based vs atomic broadcast-based, schemi di rilevamento di conflitti ottimistici vs pessimistici. Purtroppo, non esiste uno schema di replicazione capace di fornire prestazioni ottimali con un ampio range di carichi e scala del sistema. Le migliori performance del sistema possono essere raggiunte solamente selezionando attentamente e dinamicamente un'appropriato protocollo di replicazione, in funzione delle caratteristiche dell'infrastruttura e del carico di lavoro.

## 1.1 Lavoro svolto

Il lavoro di tesi è stato svolto nell'ambito del progetto europeo Cloud-TM ed ha avuto durata di 12 mesi. Il progetto Cloud-TM è stato realizzato con la collaborazione dei gruppi di ricerca appartenenti a INESC-ID, CINI, ALGORITHMICA e RedHat, e si pone l'obiettivo di risolvere le problematiche inerenti gli aspetti evidenziati precedentemente.
Il contributo fornito dal lavoro svolto al progetto Cloud-TM, ha permesso di:

- disegnare e sviluppare un framework di benchmarking per piattaforme transazionali distribuite, unificato, atto a valutare l'efficacia e l'efficienza di tecniche di *elasting scaling*;
- disegnare ed implementare il componente Autonomic Manager, proprio dell'architettura di Cloud-TM, responsabile di effettuare il self-tuning e lo elastic scaling della piattaforma dati transazionale sottostante.

Il framework di benchmarking prodotto, chiamato ELASTIC DTM BENCH, offre le seguenti caratteristiche:

- fornisce un insieme di benchmark eterogenei per sistemi transazionali distribuiti, ispirati e riadattati da popolari suite di benchmark sviluppati in area dei sistemi di database;
- facilita la portabilità delle applicazioni di benchmarking tra differenti piattaforme di memorie transazionali distribuite, astraendo sulle implementazioni per mezzo di una interfaccia, semplice e generica, di un *key/value store* transazionale;
- offre differenti strategie per la generazione di carichi di lavoro variabili nel tempo, specificatamente pensate per stressare l'elasticità della piattaforma transazionale distribuita. Sono stati inclusi sia generatori di carico basati su tracce (passate in ingresso attraverso file) sia basati su funzioni analitiche componibili (gradino, rampa e funzioni periodiche);
- permette il *plug-and-play* di tecniche di predizione di carico e di prestazione, attraverso, anche in questo caso, la definizione di interfacce astratte rivolte a nascondere l'eterogeneità delle implementazioni dei due componenti;
- assicura la portabilità tra differenti piattaforme cloud Infrastructure-as-a-Service (IaaS) attraverso un livello astratto che media l'interazione tra il fornitore della IaaS e del controllore, responsabile di automatizzare il processo di elastic scaling.

Il componente Autonomic Manager rappresenta un elemento di fondamentale importanza all'interno del progetto Cloud-TM in quanto permette di soddisfare Service Level Agreement prestabiliti con l'utente e garantire quindi Quality of Service al variare del volume e del profilo del carico di lavoro. Lo Autonomic Manager è stato disegnato in modo da disporre sia degli strumenti necessari per intervenire nella gestione delle risorse computazionali, cioè dello elastic scaling, richiedendo e rilasciando le risorse ad un provider cloud IaaS, sia degli strumenti necessari per adattare, autonomamente, la piattaforma transazionale al workload in ingresso.

# CAPITOLO 2

# BACKGROUND

Lo scopo di questa sezione è di introdurre i concetti di base sulle Memorie Transazionali e le principali tecnologie coinvolte durante lo sviluppo del progetto di tesi

## 2.1 Memorie Transazionali

L'introduzione dei multi processori, o multi core, ha marcato l'inizio dell'era della programmazione parallela permettendo di superare il limite intrinseco imposto alle performance dei processori sequenziali da importanti aspetti quali consumo di energia e dissipazione di calore. L'avanzare della tecnologia ha reso possibile l'installazione di un elevato numero di core su chip di ridotte dimensioni, superando di gran lunga le capacità degli sviluppatori software nel creare applicazioni parallele capaci di sfruttare in todo la corrispettiva potenza di calcolo. Il disegno, la scrittura e il debug di algoritmi paralleli, a differenza dei sequenziali, sono attività di fatto molto più ardue, sia a causa della natura umana, la quale ha difficoltà nel tenere traccia di eventi simultanei, sia per la mancanza di meccanismi utili ad astrarre e comporre [11], attività fondamentali per gestire la complessità. Threads e locks, meccanismi espliciti di sincronizzazione a basso livello largamente utilizzati nella programmazione parallela, non sono in grado di garantire tali attività e richiedono attenzione da parte degli sviluppatori durante il disegno e

l'implementazione delle applicazioni per non incorrere in problematiche quali deadlocks o data races. Tali meccanismi introducono inoltre, limiti nelle performance limitando la scalabilità e aumentando la complessità dei programmi stessi. Nell'ambito della programmazione parallela, le Memorie Transazionali (TM, Transactional Memory) nascono durante la ricerca di soluzioni innovative capaci di semplificare lo sviluppo di applicazioni parallele. Esse introducono la transazione, astrazione già nota nel mondo dei database, come parte integrante dei linguaggi di programmazione, proponendosi come alternativa ai tipici schemi lock-based per l'accesso alle sezioni critiche all'interno delle applicazioni parallele/distribuite.
Una transazione è un sequenza di operazioni, delimitata da appositi marker, la cui esecuzione rispetta determinate proprietà note come ACID, di seguito presentate:

**Atomicità** Le operazioni che compongono la transazione devono essere eseguite in todo (noto come commit), altrimenti tutte le operazioni della transazione devono essere scartate, come se non fossero mai state eseguite (noto come abort).

**Consistenza** Una transazione che effettua commit accedendo ad uno stato dei dati consistente, dove il concetto di consistenza dipende interamente dall'applicazione e dal dominio dei suoi dati (nel dominio bancario, ad esempio, se i dati rappresentano conti bancari, una operazione di prelievo su un conto deve garantire che il saldo sia maggiore o uguale alla quantità da prelevare), deve lasciare l'applicazione in un nuovo stato consistente.

**Isolamento** L'esecuzione di una transazione non deve interferire con l'esecuzione di una eventuale altra. Definisce quindi quando e come le modifiche allo stato dei dati apportate da una transazione devono essere rese visibile.

**Durabilità** Richiede che le modifiche apportate da una transazione che esegue il commit, siano rese persistenti nello stato anche a fronte di guasti.

Varie implementazioni di sistemi di memorie transazionali sono state proposte, sia attraverso il supporto hardware (Hardware Transactional Memory,

HTM), sia interamente software (Software Transactional Memory, STM). Le prime implementazioni di memorie transazionali hardware mantenevano le modifiche dello stato fatte da un transazione nella cache ed usavano i protocolli di coerenza della cache per rilevare conflitti. Differentemente, le implementazioni più recenti usano un buffer di scrittura associato con il processore per mantenere gli aggiornamenti.
Entrambe le implementazioni presentano alcuni punti a favore: da un lato, essendo il software più flessibile, permette l'implementazione di vari e sofisticati algoritmi, facili da modificare e ad estendere; inoltre permette di essere integrato facilmente nei sistemi già esistenti. Dall'altro lato, le memorie transazionali hardware riescono ad eseguire le applicazioni con degli overheads ridotti e non richiedono quindi alcuna ottimizzazione per raggiungere le performance. Inoltre, una implementazione hardware risulta essere meno invasiva poichè ogni accesso alla memoria è trattato come una transazione implicita.

### 2.1.1 Classificazione

È possibile classificare le possibili implementazioni di un sistema basato su memoria transazionale, sia esso software, sia esso hardware, in base ad alcune scelte di design fatte sui meccanismi che lo compongono, quali controllo della concorrenza, controllo delle versioni e per la rilevazione dei conflitti.
Prima di proseguire nella descrizione di tali scelte, definiamo il concetto di conflitto tra transazioni: due transazioni si dicono in conflitto se due operazioni, rispettivamente appartenenti ad esse, sullo stesso dato, sono o due scritture concorrenti oppure una operazione è di scrittura e l'altra di lettura.

#### 2.1.1.1 Controllo della concorrenza

A seconda degli istanti in cui si verificano rispettivamente gli eventi di verificazione, rilevazione e risoluzione di un conflitto tra transazioni, distinguiamo due tecniche di controllo della concorrenza: una pessimistica, nella quale tutti i tre eventi si verificano nello stesso istante, ed una ottimistica nella quale, rilevazione e risoluzione possono verificarsi successivamente all'evento di verificazione. Il controllo della concorrenza pessimistico richiede

che la transazione sia la proprietaria di un dato nel momento stesso in cui vi accede, impedendo quindi l'accesso a transazioni concorrenti. Normalmente questo comportamento è realizzato per mezzo di lock sul dato. Particolare attenzione è richiesta durante l'implementazione di questa tecnica affinchè non si verifichino dei deadlock, le transazioni si bloccano a vicenda aspettando che si liberi una risorsa necessaria all'altra transazione e viceversa. D'altra parte, il controllo della concorrenza ottimistico permette a transazioni di accedere ai dati concorrentemente, eseguire nonostante la presenza di azioni conflittuanti fintanto che il sistema di TM non individua e risolve tale conflitto abortendo una transazione o, ad esempio, ritardandola. Anche in questo caso, l'implementazione di tale tecnica all'interno dei sistemi di TM richiede attenzione per non incorrere in livelock, situazioni dove non vi è progresso; come esempio, se due transazioni sono in conflitto e la seconda venga fatta abortire dal sistema di controllo di concorrenza, questa potrebbe rieseguire e provocare l'abort della prima. Di norma, il controllo pessimistico è utile in quei casi dove il numero di conflitti è elevato; d'altra parte, in scenari in cui i conflitti sono rari, un controllo ottimistico permette di evitare i costi dei lock incrementando la concorrenza tra le transazioni. Soluzioni ibride vengono applicate cercando di ottenere benefici da entrambi le parti

#### 2.1.1.2 Gestione delle versioni

Un sistema di TM deve disporre di un meccanismo capace di gestire i tentativi di scrittura tra transazioni concorrenti; in letteratura, è possibile trovare due soluzioni note rispettivamente come *eager version management* e *lazy version management*.
Adottando la soluzione *eager version management*, durante l'esecuzione di una transazione le modifiche ai dati vengono effettuate accedendo direttamente alla memoria. L'adozione di un registro *undo-log* contente gli ultimi valori scritti, permette di ripristinare i valori originali dei dati nel caso in cui il commit della transazione non andasse a buon fine. L'uso di tale tecnica richiede obbligatoriamente l'adozione di un meccanismo di concorrenza pessimistico.
Differentemente, la seconda soluzione, *lazy version management*, permette di ritardare l'aggiornamento delle locazioni di memoria dei dati all'istante in cui la transazione effettua il commit. Ciò è possibile associando, ad ogni transazione, un registro *redo-log*, all'interno del quale essa scrive i suoi

aggiornamenti; tale registro sarà acceduto dalle successive operazioni di lettura presenti nella transazione. Infine, i dati presenti all'interno del *redo-log* saranno usati per aggiornare le locazioni di memoria nel caso in cui il commit, da parte della transazione, sia andato a buon fine.

#### 2.1.1.3 Rilevazione dei conflitti

Nel caso di controllo di concorrenza pessimistico, la rilevazione dei conflitti è banale poichè un lock, usato nell'implementazione, può essere acquisito solo nel caso in cui nessun altro thread lo abbia precedentemente ottenuto. Nel caso in cui il sistema di TM utilizzi un controllo della concorrenza ottimistico, varie tecniche, basate sull'esecuzione di un'operazione di validazione, sono state studiate e sono classificate secondo tre dimensioni tra esse ortogonali:

- granularità della rilevazione del conflitto (ad esempio a livello di completa linea di cache per i sistemi HTM o, a livello di oggetto, per STM).
- tempo di rilevamento del conflitto
  - eager conflict detection, se effettuato all'instante di acquisizione dell'accesso o alla prima referenza al dato.
  - al tempo di validazione, controllando la collezione di locazioni precedentemente lette o aggiornate per vedere se qualche altra trnasazione le ha modificate. L'operazione di validazione può essere eseguita varie volte ed in qualsiasi istante durante l'esecuzione della transazione
  - lazy conflict detection, se la rilevazione dei conflitti è fatta quando giusto prima che la transazione effettui il commit
- tipi di accessi conflittuali
  - tentative conflict detection, rileva il conflitto tra due transazioni concorrenti prima che una delle due esegue il commit.
  - committed conflict detection, rileva il conflitto tra transazioni attive e quelle che già hanno effettuato il commit

Tecniche di eager conflict detection sono, di norma, accoppiate con quelle di tentative conflict detection; d'altra parte, tecniche di lazy conflict detection sono usa in congiunto a tecniche di committed conflict detection. Vari sistemi di TM hanno proposto l'uso di soluzioni ibride.

## 2.2 Memorie Transazionali Distribuite

Il grande interesse riscosso nella comunità scientifica dal nuovo paradigma di programmazione orientato alle transazioni si è ben presto diffuso anche all'interno delle organizzazioni produttive, senza alcuna distinzione sulla loro dimensione.
Un sistema reale, messo in un ambiente produttivo, è esposto ad un numero variabile di richieste entranti, difficilmente prevedibile in toto: affinchè sia possibile processare il maggior numero possibile di richieste, i requisiti di alta disponibilità, scalabilità e performance acquisiscono importanza primaria.
Alcune tecniche di replicazione, già disegnate precedentemente, si sono rivelate particolarmente efficaci per migliorare la disponibilità dei sistemi di calcolo; tali tecniche sono state largamente utilizzate nel contesto dei sistemi di database. Le affinità tra i sistemi di database e quelli di memorie transazionali hanno perciò rinnovato l'interesse per tali soluzioni, riadattandole e integrandole nei sistemi di TM.
Poichè lo stato dei dati di un tale sistema è distribuito/replicato tra vari nodi, connessi attraverso una rete, in letteratura si usa il termine Memoria Transazionale Distribuita (DTM) per riferirsi ad una generica implementazione di tali sistemi.
Quando una transazione viene eseguita su un sistema distribuito, le proprietà ACID, precedentemente introdotte, devono valere anche al di fuori del nodo sulla quale è eseguita.

### 2.2.1 Classificazione

In un sistema di memoria transazionale distribuito, vari aspetti permettono di effettuare delle scelte di design

#### 2.2.1.1 Grado di replicazione

Un sistema può essere totalmente o parzialmente replicato per garantire resistenza ai guasti e raggiungere il requisito di scalabilità. L'adozione di uno schema di replicazione totale, in cui ogni nodo conserva una copia locale degli oggetti, rende il sistema robusto e capace di garantire alta affidabilità. Tuttavia, specialmente a fronte di *workload* prevalentemente formati da operazioni di aggiornamento, richiede di contattare tutti i nodi nella fase di *commit* vincolando la scalabilità del sistema.
Contrariamente, l'uso di schemi di replicazione parziale permettono di rendere altamente scalabile un sistema. Solamente un sottoinsieme dello stato applicativo è replicato su un numero predeterminato di repliche, le quali, a fine di una transazione di scrittura (cioè in cui almeno una operazione è di aggiornamento), dovranno essere coordinate affinchè sia garantita la consistenza tra esse. Dividendo il carico del sistema tra i vari nodi, l'uso di tecniche di replicazione parziale permette quindi di incrementare il *throughput*. Il partizionamento dello stato dei dati tra le varie repliche richiede però il pagamento di un costo da parte dei nodi che vogliono accedere ad un oggetto remoto, non replicato localmente.

#### 2.2.1.2 Protocolli di replicazione

All'interno di un sistema replicato, un protocollo di coerenza deve assicurare che tutti i nodi che partecipano ad una transazione, applichino sulle repliche le stesse operazioni, nello stesso ordine. Affinchè lo stato di ogni replica non diverga, è richiesto che le operazioni applicate siano deterministiche. In [16] è proposto un protocollo basato su primitive di *Total Order Multicast* (TOM) per disseminare l'ordine di serializzazione delle transazioni tra i nodi contenenti le repliche degli oggetti coinvolti. Tali primitive, specializzazioni del *Atomic Broadcast* (AB), risultano però relativamente lente poichè richiedono consenso tra i nodi. Una soluzione, proposta in [13], introduce la primitiva *Optimistic Atomic Broadcast*, promettendo di abbassare i ritardi medi di consegna dei messaggi. La consegna delle richieste avviene in maniera ottimistica, eseguendo la coordinazione tra i nodi solamente in una seconda fase, nel caso in cui l'ordine di consegna non sia comune ai partecipanti. Aggro [14] è un altro esempio di implementazione che usa OAB.

#### 2.2.1.3 Modello di Programmazione: Control e Data Flow

Possiamo identificare in letteratura due principali modelli di programmazione per DTM: il Control Flow Model, dove gli oggetti sono immobili e le operazioni su tali oggetti sono eseguite nei nodi proprietari dei dati, e il Data Flow, nel quale invece l'esecuzione della transazione è localizzata all'interno di un singolo nodo, e gli oggetti migrano tra i nodi.
Nel control flow model, più in dettaglio, la computazione si sposta da nodo a nodo attraverso delle chiamate a procedure remote (RPC), poichè gli oggetti sono assegnati staticamente ai nodi della DTM. La sincronizzazione tra transazioni è garantita attraverso l'uso two-phase locking e meccanismi di rivelazione di deadlock; l'atomicità di una transazione è garantita dal protocollo two-phase commit.
L'implementazione Snake-DSTM, proposta in [18], adotta il modello data flow; la chiamata sull'oggetto remoto è effettuata per mezzo del meccanismo di invocazione di metodi remoti (RMI); eventuali e conseguenti chiamate ad ulteriori metodi, su oggetti appartenenti a differenti nodi, danno luogo ad un grafo di chiamate. Quest'ultimo è usato durante il processo decisionale di commit.
Nel data flow model, la transazione esegue in un singolo nodo, recuperando i dati dagli altri nodi e adotta tecniche di sincronizzazione ottimistica, ritardando cioè la verifica di conflitti a fine esecuzione. Di norma, il modulo, chiamato Contention Manager, è incaricato di gestire la sincronizzazione ed evitare deadlock e livelock. Le DTM che optano per tale modello, non presentano alcun protocollo di commit distribuito, potendo di fatto eseguire il commit se giunge al termine senza essere state interrotte. Le implementazioni più comuni di DTM che adottano il data flow model, usano un approccio directory-based, nel quale l'ultima locazione di un oggetto è salvata in una directory distribuita per minimizzare i costi di lookup.
Ballistic [12], Relay[20] e Combine[2] sono alcuni esempi di implementazioni, proposte nella letteratura scientifica le quali adottano il modello data flow.
Ballistic è un protocollo che tiene conto della distribuisce gli oggetti per mezzo di una suddivisione dei nodi in gerarchie di cluster. Ogni gerarchia designa un leader in grado di comunicare con le gerarchie adiacenti, superiore e inferiore. La richiesta per un oggetto viene analizzata dai leader i quali la reindirizzano al livello superiore o inferiore corretto fintanto che la

copia cache dell'oggetto non viene trovata.
Il protocollo Relay è basato su un albero ricoprente fisso e sposta l'oggetto remoto nel nodo dove la transazione è in esecuzione ripercorrendo al contrario il cammino, mantenuto in una struttura di dati, effettuato dal messaggio per raggiungere il nodo richiesto.
A differenza dei due protocolli precedenti, il protocollo Combine non pone assunzioni sull'ordine dei messaggi scambiati nei link di comunicazione e combina le richieste concorrenti ad oggetti nello stesso messaggio, scambiato tra i nodi, organizzati in un albero ricoprente.
Infine, un compromesso tra i due modelli precedentemente descritti è fornito da soluzioni ibride, che cercano di sfruttare vantaggi di entrambi i precedenti modelli. L'implementazione proposta in [3], per esempio, viene demandata al programmatore la scelta del modello da usare, in accordo con le caratteristiche della transazione. Un'altro esempio di implementazione ibrida è fornita in [17] dove è presentato HyFlow: ogni oggetto ha un nodo proprietario soggetto a variazione se utilizzato il modello data flow. In tale caso, il nuovo proprietario è notificato agli altri nodi con un messaggio di broadcast.

## 2.3 Self-Tuning di DSTM

Attraverso l'uso di meccanismi di *self-tuning* (anche detto di *auto-tuning*), un generico sistema può decidere se variare la propria configurazione, modificando, a tempo d'esecuzione, specifici parametri interni, con lo scopo di massimizzare o minimizzare una specifica funzione obiettivo, come ad esempio massimizzare l'efficienza o minimizzare gli errori.
Per quanto riguarda i sistemi di memorie transazionali distribuite, l'introduzione di schemi di *self-tuning* permette di massimizzare le performance dell'applicazione in funzionamento, espressa con il numero di richieste correttamente gestite. Tali performance dipendono però da una varietà di parametri rilevanti, presenti su differenti livelli dell'architettura.

**Infrastruttura Hardware** Le capacità computazionali dell'infrastruttura che ospita la piattaforma hanno un forte impatto sul *response time* di una transazione. Oltre alla velocità di processamento della CPU e delle periferiche di I/O, anche la larghezza di banda della rete gioca un fattore importante per le prestazioni; infatti, all'interno delle piattaforme

distribuite transazionali, predominano le fasi di sincronizzazione per garantire le proprietà ACID.

**Data management platform** A livello di piattaforma, le prestazioni sono affetta da parametri di varia natura. Per garantire l'isolamento e l'atomicità, ad esempio, vengono introdotti *overhead* e costose fasi di sincronizzazione, limitando allo stesso tempo il grado di concorrenza dell'applicazione. Le prestazioni degli schemi di controllo di concorrenza o i protocolli di *commitment* sono invece determinate da altri paramentri, come il protocollo e il grado di replicazione, definendo, in particolare, un *tradeoff* tra località dei dati (probabilità di co-localizzazione di transazione e dati da essa acceduti) ed estensione della fase di commit (intensa come numero di nodi contattati per determinare l'esito della transazione).

**Logica di Business** Infine, le performance dipendono anche dal carico di lavoro presentato dall'applicazione stessa. Ad esempio, a seconda della natura del carico generato dall'applicazione (CPU o rete *dependent*), saranno generate prevalentemente transazioni di tipo *read-only* o *update*, le quali inducono più o meno contesa sui dati (influenzando le performance della piattaforma).

Creare dei meccanismi per predirre gli effetti di tutti questi fattori, spesso intrecciati tra essi, sulle prestazioni non è affatto banale. Ciò nonostante, questo è un requisito fondamentale per svolgere *capacity planning* e per progettare degli schemi di *self-optimizing* per la piattaforma.
Per tale motivo, tali meccanismi normalmente si concentrano esclusivamente su due livelli:

- auto-riconfigurazione delle risorse a livello di infrastruttura, affinchè sia possibile allocare la quantità minima di risorse necessare per soddisfare il carico attuale con le qualità del servizio prestabilite.
- auto-riconfigurazione degli schemi e dei parametri adottati dalla piattaforma, affinchè sia possibile massimizzare le performance.

È bene notare che i meccanismi di *self-tuning* che agiscono sui due livelli appena descritti non sono ortogonali: uno specifico protocollo di replicazione, ad esempio, potrebbe risultare ottimo in piattaforme di piccole dimensioni,

e, al contrario, un altro protocollo potrebbe presentare migliori performance quando adottato su una piattaforma di grandi dimensioni. Determinare quindi la configurazione ottima per una piattaforma transazionale richiede l'uso di modelli in grado di considerare entrambi gli aspetti per poterne catturare le interdipendenze.

### 2.3.1 Metodologie di *Self-tuning*

In questa sezione verranno presentate le principali tecniche adottate dalle soluzioni di self-tuning all'avanguardia: Teoria delle Code, Machine Learning e Teoria dei Controlli.

**Modellazione analitica basata sulla teoria delle code** La Teoria delle Code permette di modellare un sistema attraverso linee di attesa, cioè code. Ogni coda rappresenta un fornitore di servizio (*server*) responsabile di processare le richieste in ingresso, generate dai clienti (*client*). Le richieste entrano nel sistema con un *rate* $\lambda$. Per descrivere una coda si usano alcuni parametri espressi attraverso la notazione di Kendall: A/B/C/K/N/D.

**A** distribuzione di probabilità del rate di arrivo (*arrival rate*) delle richieste

**B** distribuzione di probabilità caratterizzante la richiesta del servizio (*service time*), cioè il tempo necessario per servire una richiesta, senza tenere in conto il tempo d'attesa nella coda.

I paramentri A e B tipicamente assumono i valori di M, D e G. Il valore M sta per Markovian e si riferisce ad un processo di Poisson caratterizzato da un paramentro $\lambda$ che indica il numero di arrivi (richieste) per unità di tempo. In tale caso, *arrival rate* e *service time* seguiranno la distribuzione esponenziale. Nel caso in cui il valore assunto da A/B sia D, significa che le distribuzioni sono deterministiche o costanti. Infine, il valore G sta a significare che la distribuzione è normale o Gaussiana con media e varianza noti.

**C** numero di server che processano elementi nella coda

**K** capacità della coda, cioè il numero totale di richieste che la coda può contenere

**N** numero di clienti che generano richieste verso la coda. Un valore di N finito permette di definire il sistema come chiuso; normalmente tale sistema è caratterizzato da un *think time*, cioè il tempo che un cliente attende dopo il completamento di una richiesta prima di generarne una nuova. Contrariamente, un valore di N infinito sta ad indicare che il sistema gestisce un numero infinito di clienti; tale sistema è detto aperto e il suo *arrival rate* non dipende dal numero di richieste già presenti nel sistema.

**D** politica adottata per gestire le richieste. Valori tipici sono First/Last Come First Serve, in cui l'ordine di processamento dipende dall'ordine di arrivo nella coda delle richieste.

I parametri precedentemente descritti caratterizzano completamente il comportamento di una coda ed è possibile calcolare analiticamente gli indicatori di performance come il numero medio di richieste nel sistema, *throughput* massimo raggiungibile e *response time* di una richiesta (inteso come somma del tempo trascorso nella coda più il tempo necessaro per il processamento da parte del server).

La teoria delle code fornisce uno strumento molto utile poichè permette, combinando multiple code in reti di code, di modellare grandi sistemi complessi. Immaginiamo ad esempio un sistema *n-tier*; esso può essere modellato attraverso n code, nelle quali le richieste scorrono fino a completamento.
Tuttavia, modellando un sistema attraverso la teoria delle code, si introducono delle assunzioni ed approssimazioni che potrebbero non essere valide nel sistema reale, compromettendo l'accurateza dei valori stimati analiticamente.

**Machine learning** Il Machine Learning (ML) rappresenta un'area fondamentale dell'intelligenza artificiale la quale si occupa della costruzione e dello studio di sistemi capaci di sintetizzare nuova conoscenza a partire da osservazioni fatte su un insieme di dati.
Vari meccanismi di previsione delle performance e *self tuning* basano il loro funzionamento su tecniche di ML e, in particolare, le tipologie *supervised*, *instance-based*, *unsupervised* e *reinforcement learning*.

**Supervised Learning** Il *supervised learning* cerca di apprendere la relazione esistente tra un insieme di paramentri in ingresso e uno in uscita. Più formalmente, un meccanismo di apprendimento supervisionato cerca di inferire una funzione $\phi : X \rightarrow Y$ , basandosi sulla risposta del sistema quando viene dato in ingresso un determinato insieme di valori $\widetilde{X} \subset X$ detto *training set*. La funzione può quindi essere usata per predirre l'uscita $y$ di un insieme di valori $(x \in X)$, non appartenenti al training set. Se la funzione $\phi$ ha codominio continuo, il machine learner è detto *regressor*; nel caso il codominio è discreto, è detto *classifier*. In maniera del tutto ortogonale, possiamo distinguere machine learner *offline trained*, se i dati appartenenti al training set sono campionati prima della messa in esercizio del sistema, o *online trained*, se il training set è costruito incrementalmente. Alcuni strumenti utilizzati dai supervised machine learner sono:

**Albero di decisione** basa il suo funzionamento su un albero di decisione, attraverso il quale mappa le osservazioni su un elemento a determinate conclusioni. Ogni nodo interno dell'albero rappresenta una variabile, un arco verso un nodo figlio rappresenta un possibile valore per tale proprietà e una foglia rappresenta il valore predetto per la variabile obiettivo.

**Rete neurale artificiale** la cui struttura ricorda quella della mente umana, dove la conoscenza è catturata dall'esperienza grazie all'interconnesione di semplici neuroni. In modo del tutto simile, una rete neurale artificiale si basa su neuroni artificiali. Nella loro forma più semplice, un neurono rappresenta un classificatore il quale effettua una somma pesata dei parametri di ingresso i quali sono così mappati in un insieme binario attraverso una funzione sigmoidea. I pesi assegnati dai ogni neurono vengono calibrati nella fase iniziale di training, eseguita sul training set. Una rete neurale è normalmente strutturata in livelli.

**Instance-based** Nel *instance-based learning* il machine learner non effettua una generalizzazione esplicita della relazione tra ingresso e uscita, come nel *supervisioned learning*, ma la risposta di un sistema ad un ingresso $x$, è determinata cercando nel training set, un ingresso che, data una funzione di somiglianza, è più simile a $x$. Uno tipico esempio

di questo tipo di learning è dato dall'algoritmo K-Nearest-Neighbor, dove l'uscita rispetto ad un ingresso $x$ è calcolata come somma pesata dei K campioni nel training set più simili a $x$.

**Unsupervised learning** L'obiettivo che lo *unsupervised leaning* (apprendimento non supervisionato) si pone è quello di classificare e organizzare, in base a caratteristiche comuni, i dati in ingresso e ricercare al loro interno dei pattern. A differenza dell'apprendimento supervisionato prima descritto, durante l'apprendimento vengono forniti al learner solo esempi non annotati, poichè nessun pattern è noto in tale fase; tali pattern devono essere riconosciuti automaticamente dal learner. Le tecniche di apprendimento non supervisionato lavorano confrontando i dati e ricercando similarità o differenze. Sono molto efficienti con elementi di tipo numerico, dato che possono utilizzare tutte le tecniche derivate dalla statistica, ma risultano essere meno efficienti con dati non numerici.

Il principale punto di forza delle tecniche di Machine Learning è che esse non richiedono una caratterizzazione analitica del sistema, rendondole particolarmente interessanti per risolvere problemi di ottimizzazione in sistemi troppo complessi per essere modellati analiticamente. Inoltre, se i valori in ingresso al sistema sono simili a quelli utilizzati per effettuare il training del *learner*, i risultati ottenuti attraverso l'uso di tecniche di apprendimento automatico possono essere molto accurati. Contrariamente, una mancanza di accuratezza si potrebbe verificare quando gli ingressi al sistema si discostano dal training set, portando quindi il machine learner a lavorare in estrapolazione. Un ulteriore limite che affetta le tecniche di machine learning è dato dalla crescita esponenziale del numero delle combinazioni causato dall'introduzione di nuove *feature*, richiedendo tempi di training insostenibili.

**Teoria dei controlli** La teoria dei controlli è un campo dell'ingegneria che studia come controllare il comportamento dei sistemi dinamici. Dato un valore di referenza per un parametro di interesse del sistema sotto controllo, un modulo, detto *controller*, è responsabile di alterare lo stato del sistema facendo variare gli ingressi del sistema, fintato da raggiungere il valore di referenza.

Per determinare come deve essere alterato lo stato del sistema controllato, il *controller* sfrutta un modello capace di catturare le relazioni tra ingresso e uscita del sistema, attraverso una funzione di trasferimento.
Possiamo individuare 3 tipi di sistemi di controllo noti in letteratura come *open-loop*, *feedback* e *feed-forward*. Essi differiscono l'uno dall'altro a seconda di come il valore di ingresso è alterato per raggiungere l'uscita desiderata.

**Open-loop controller** prende in considerazione solamente lo stato corrente del sistema e il modello adottato.

**Feedback controller** oltre allo stato del sistema e alla funzione di trasferimento tiene in considerazione la deviazione dell'uscita ottenuta dal valore di referenza desiderato.

**Feedforward controller** utilizza un modello per predirre il comportamento del sistema e quindi agisce prima che l'errore si verifica. Poichè la predizione potrebbe verificarsi errata, l'uso del feedforward controllor è spesso combinato con uno di tipo feedback.

Le soluzioni basate sulla teoria dei controlli sono caratterizzate da un alto grado di robustezza. Tuttavia in alcuni casi la raccolta di informazioni di feedback utili può ridurre la reattività del controllore; gli schemi di feedforward aiutano, tuttavia, a superare questa limitazione. D'altra parte, sia la funzione di trasferimento, sia gli schemi di predizione, tipici degli schemi feedforward, sono basati su modelli, presentando quindi gli stessi limiti già evidenziati per gli approcci di self-tuning basati su teoria delle code e machine learning.

**Metodologie miste** Alcuni lavori proposti implementano entrambe le tecniche di *machine learning* e di modellazione analitica, cercando di trovare un equilibrio tra i due approcci.

### 2.3.2 Parametri di interesse e stato dell'arte

Come già stato introdotto, per semplificare lo sviluppo dei meccanismi di *self-tuning* di sistemi di DTM, è possibile concentrarsi su determinati livelli della piattaforma. Per ognuno di questi, è di interesse identificare specifici paramentri la cui variazione comporti un cambiamento significativo delle performance dell'applicazione. Tali parametri sono presentati in questa

sezione, oltre ad una classificazione delle soluzioni finora presentate in letteratura; una maggiore desccrizione è svolta per quelle tecniche direttamente interessate nel lavoro di tesi.

#### 2.3.2.1 Protocollo di replicazione

La scelta del protocollo di replicazione è una delle responsabilità di cui i meccanismi di *self-tuning* si fanno carico.

**Stato dell'arte** Tra le varie soluzioni proposte in letteratura troviamo *PolyCert*, in *Couceiro et al.* [5], dove è presentato un protocollo polimorfico per il *self-tuning* del protocollo di replicazione, in grado di supportare simultaneamente i tre schemi di replicazione *Certification Based* (non-voting, voting and Bloom Filter Certification), presentati nella sezione precedente. *PolyCert* permette di determinare, per ogni transazione, la strategia di replicazione da utilizzare in base ad un oracolo, modulare, del quale è fornita sia una implementazione off-line, sia una on-line capace di rinforzare le tecniche di apprendimento. Un altra soluzione, sempre sviluppata da *Couceiro et al.*, basata su machine learning, è *MorphR* [6], framework in grado di cambiare, a livello di piattaforma transazionale, il protocollo di replicazione. *MorphR* formalizza un insieme di interfacce, implementabili da un generico protocollo di replicazione affinchè sia possibile attuare la riconfigurazione online. *MorphR* presenta, al suo interno, l'implementazione di tre noti protocolli di replicazione (two-phase commit, primary-backup and total order).

#### 2.3.2.2 Elasting Scaling

Affidandosi a modelli di dato semplici, come key/value, e usando meccanismi di replicazioni, le memorie transazionali permetto di scalare con facilità su un numero di nodi variabile. Un sistema transazionale in grado di fare *elasting scaling* riesce ad adattare, al variare del workload, il numero dei nodi e dei thread (per nodo) attivi, cercando di raggiungere il livello di concorrenza giusto. In tale scenario però, al crescere del numero dei nodi, le *performance* presentano un comportamento fortemente non lineare attribuibile agli effetti, simultanei e interdipendenti, della *contention* che si verifica sia sulle risorse fisiche (cpu, memoria, *network*) sia su quelle logiche (dati, acceduti da transazioni concorrenti).

**Stato dell'arte** *Rughetti et al.* in [15], presentano una soluzione basata su una rete neurale, utilizzando un algoritmo di controllo *on-line* per attivare o disattivare il numero di thread, affinchè possa essere raggiunto il giusto livello di concorrenza.
*Transactional Auto Scaler*(TAS) [7], è uno strumento capace di predire le performance di un data grid transazionale distribuito, quando in funzione su un numero di nodi variabile.
Le predizioni svolte da TAS beneficiano della sinergia tra le tecniche analitiche e machine learning. Esse seguono un approccio *divide and conquer* secondo il quale, la tecnica più appropriata è adottata per determinare l'impatto sulle prestazioni su specifici livelli della piattaforma transazionale.
TAS è sia in grado di effettuare predizioni offline del tipo *what-if analysis*, sia di guidare l'elasting scaling della piattaforma sul quale è messo in funzione attraverso l'integrazione con un controllore online.
Basandosi sulla tecnica *Mean Value Analysis* (MVA), TAS è in grado di predirre i valori medi per gli indicatori chiave delle prestazioni (KPI) quali throughput, response time e probabilità di abort per transazione.
In TAS vengono definiti tre modelli ognuno dei quali agisce su un livello differente della piattaforma:

**Contesa dei dati** Un modello a scatola aperta (*white-box*) sfrutta la conoscenza degli schemi di controllo di concorrenza adottati in Infinispan 2.4.1 per disegnare una rappresentazione analitica della contesa dei dati. TAS fornisce i modelli per i protocolli di replicazioni Two-Phase Commit e Primary-Backup, con sistema full-replicated e livello di replicazione Repeatable Read.

**Contesa sulle risorse di CPU** Un modello cattura gli effetti dell'uso delle CPU sul response time delle transazioni. La versione di TAS di referenza, permette di adottare o un modello statistico (costruito offline) o un modello analitico nel quale una CPU è modellata come una coda M/M/K, con K pari al numero di core di una CPU.

**Contesa sulle risorse di rete** Un modello statistico è in grado di predire la durata delle interazioni tra i nodi di un sistema, cioè i tempi di commit e i tempi di accesso ai dati remoti. I tempi predetti da questo modello tengono in conto sia le latenze introdotte dai collegamenti, sia

dei tempi di processamento degli overhead introdotti dai vari livelli software. Alcuni di questi costi sono specifici per una data piattaforma (ad esempio, tempi di packing/unpacking dei messaggi e tempi per garantire la consegna di quest'ultimi), altri infrastrutturali, dipendenti cioè dai protocolli adottati a livello di trasporto, rete e collegamento dello stack OSI. Disegnare dei modelli analitici per derivare tali tempi è una attività molto ardua e vincolante alla piattaforma e allo stack dei protocolli adottati. Per queste ragioni, il modello per la predizione delle risorse di rete in TAS è costruito attraverso l'uso di Cubist.

TAS introduce un valore scalare, detto *Application Contention Factor* (ACF), il quale permette di caratterizzare il patter di accesso ai dati e quindi misurare il grado di contesa di una applicazione.
L'introduzione di tale ACF distingue TAS dal resto dei modelli analitici già proposti in letteratura, i quali modellano pattern di accesso ai dati non uniformi attraverso una combinazione di distribuzioni uniformi. In TAS, invece, l'uso dello ACF permette di modellare tali pattern non uniformi attraverso un pattern, equivalente, uniforme.
Il calcolo del valore dello ACF è facilmente ottenibile a run-time attraverso la formula seguente:

$$ACF = \frac{P_{lock}}{\lambda_{lock} T_H} \tag{2.1}$$

Il valore $\frac{1}{ACF}$ rappresenta l'approssimazione della dimensione del database tale che, se l'applicazione avesse generato un patter di accesso uniforme sui dati di tale database, sarebbe incorso nella stessa probabilità di contesa della attuale esecuzione (nella quale l'accesso ai dati sia casuale).
L'indicatore ACF presenta, inoltre, tre ulteriori caratteristiche chiavi:

- è costante rispetto al numero di nodi che compongono il sistema;
- è insensibile al protocollo di replicazione
- è indipendente dalla piattaforma (sia essa una cloud privata, sia pubblica).

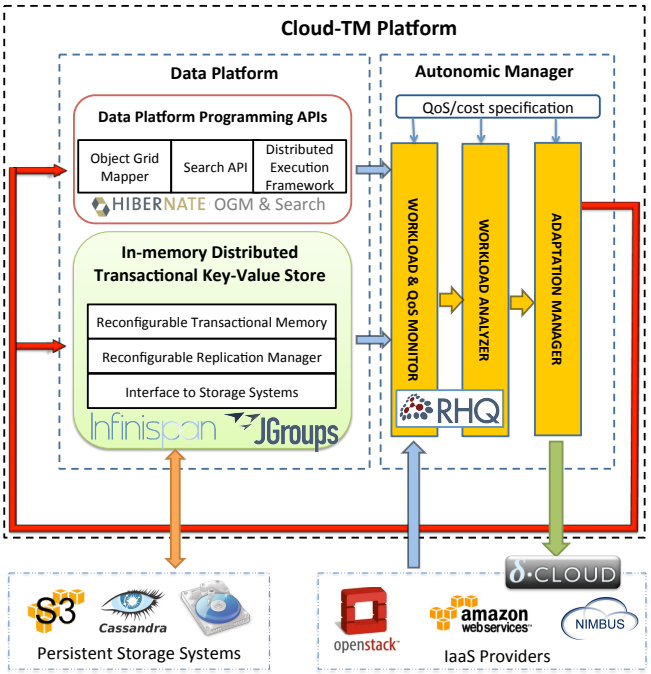


**Fig. 2.1: Architettura del progetto Cloud-TM**

## 2.4 Cloud-TM

Il progetto europeo Cloud-TM presenta una piattaforma transazionale, innovatica e data-centric, capace di facilitare lo sviluppo e ridurre i costi operazionali e di amministrazione associati con applicazioni cloud.
A partire dai requisiti di scalabilità e dinamicità, tipici delle infrastrutture cloud, il progetto Cloud-TM fornisce intuitive e potenti astrazioni attraverso le quali è possibile mascherare la complessita e quindi di sfruttare appieno le potenzialità delle piattaforme cloud.
Inoltre, il progetto integra schemi di self-tuning capaci di adottare, in maniera sinergica, le diverse metodologie precedentemente presentate, quali la modellazione analitica, schemi simulativi e machine learning, affinchè sia possibile raggiungere ottima l'efficienza sotto ogni scala e per ogni carico di lavoro.
Nella Figura 2.1 è presentata l'architettura ad alto livello della piattaforma Cloud-TM; come evindeniziato, la piattaforma si compone di due parti principali: il Data Platform e lo Autonomic Manager.

### 2.4.1 Data Platform

Il Data Platform è responsabile di memorizzare, recuperare e manipolare i dati presenti all'interno di un insieme di nodi distribuiti, acquisiti elasticamente tramite piattaforme IaaS Cloud. Le API esposte dal Data Platform sono state disegnate per semplificare lo sviluppo di grandi applicazioni data-centric, sviluppate per funzionare su infrastrutture cloud. Per tale motivo, le interfacce offerte permettono di:

1. memorizzare/interrogare dati nel/dal Data Platform utilizzando le familiari e comode astrazioni fornite dal paradigma di programmazione orientato agli oggetti, quali ereditarietà, polimorfismo, associazioni.

2. trarre beneficio dalla potenza di processamento della piattaforma Cloud-TM per mezzo dell'insieme di astrazioni che nascondono la complessità associata con la programmazione parallela/distribuita, quali sincronizzazione e scheduling di thread, e tolleranza ai guasti.

3. garantire il raggiungimento dei requisiti di alta scalabilità e consistenza utilizzando protocolli di gestione dei dati decentralizzati, tecniche di replicazione parziale e meccanismi di bilanciamento del carico *locality aware*.

**Infinispan: in-memory *key-value store*** Alla base del Data Platform troviamo una piattaforma transazionale distribuita, altamente scalabile, la quale rappresenta la backbone dell'intero progetto; tale ruolo è svolto dal *key-value store* Infinispan, il quale è stato integrato ed esteso, all'interno del progetto Cloud-TM, introducendo algoritmi innovativi per la replicazione dei dati e meccanismi attraverso i quali è possibile riconfigurazione in real-time il sistema.
Infinispan è un *in-memory* key-value store distribuito, transazionale, altamente scalabile, sviluppato in linguaggio Java, rilasciato sotto licenza open-source e sponsorizzato da Red Hat.
L'architettura di Infinispan è estremamente flessibile e supporta due modalità di funzionamento: standalone (non-clustered) e distribuita. Nella modalità standalone, Infinispan si comporta come una memoria transazionale software, in cui è possibile interagire con una memoria condivisa attraverso il supporto di transazioni e secondo il modello chiave-valore. Tale modalità

di funzionamento è indicata per ambienti di sviluppo e di test. Tuttavia, solamente attraverso l'uso della modalità distribuita, si riesce a sfruttare tutte le potenzialità di Infinispan. Infatti, quando in funzionamento in tale modalità, la piattaforma è capace di sfruttare le risorse computazionali di un insieme di nodi distinti.
Infinispan fa uso di un meccanismo di controllo di concorrenza multi-versioned, il quale offre vantaggi rispetto ai classici meccanismi di controllo di concorrenza basati su lock, quali:

- letture e scritture possono essere eseguite concorrentemente;
- le operazioni di lettura e scrittura, non bloccano operazioni successive;
- anomalie di tipo write skew sono facilmente rilevabile e gestibili;

Infinispan offre due livelli di isolamento, READ COMMITTED (RC), usato di default, e REPEATABLE READ (RR), configurabile attraverso file di configurazione. La scelta del livello di isolamento determina quando le letture vedono il risultato di una scrittura concorrente.
A livello di nodo, l'architettura di Infinispan prevede i seguenti componenti:

***Transaction Manager*** responsabile per l'esecuzione della transazione, locale o remota.

***Lock Manager*** responsabile della gestione dei *lock* acquisiti dalle transazioni e della rivelazione di *deadlock distribuiti*. In tal caso, una transazione viene fatta abortire.

***Replication Manager*** responsabile della coerenza tra le repliche degli oggetti. Certifica l'esecuzione delle transazioni attraverso l'uso del protocollo *two-phase commit*.

***Persistent Storing*** responsabile della persistenza e successivo caricamento su/da disco duro.

***Group Communication System*** responsabile di mantenere informazioni sui membri di un gruppo, incaricandosi di rilevare guasti e offrire supporto per la comunicazione tra repliche.

Durante il progetto Cloud-TM, Infinispan è stato arricchito con un gran numero di feature riguardanti replicazione dei dati, meccanismi di posizionamento dei dati e state transfer. A seguire si presentano aspetti principali delle innovazioni introdotte. Per maggiori dettagli si rimanda alla documentazione ufficiale rilasciata.

**Protocollo GMU** *Genuine Multiversion Update serializability* (GMU) è un innovativo protocollo di replicazione transazionale capace di garantire che le transazioni di tipo read-only non siano mai abortite o sottoposte a un fase di validazione addizionale. GMU si basa su un algoritmo di concorrenza multiversione distribuito, il quale fà uso di innovativi meccanismi di sincronizzazione basati sui vector clock per mantenere traccia, in maniera decentralizzata, sia i dati, sia le relazioni causali tra le transazioni.
GMU assicura il criterio di consistenza Extended Update Serializability (EUS); tale criterio presenta garanzie analoghe a quelle offerte dal criterio classico, e più forte, 1-Copy Serializabilit (1CS) per le transazioni di tipo update, assicurando l'evoluzione dello stato del sistema consistente. EUS garantisce che le transazioni read-only osservano uno snapshot uquivalente a una esecuzione serializzabile delle transazioni di update. EUS estende le garanzie di osservare snapshot consistenti anche quando transazioni di update devono, eventualmente, abortire.
Il protocollo GMU è il primo protocollo di replicazione parziale genuino, che sfrutta la semantica di consistenza EUS per implementare un schema di controllo di concorrenza che non introduce punti di sincronizzazione globali, non richiedendo fasi di validazioni costose per effettuare il commit di transazioni read-only.

**Non Blocking State Transfer** Non Blocking State Transfer è un innovativo protocollo capace di prestare efficiente supporto alle attività incaricate di far variare la dimensione della piattaforma e il grado di replicazione di ogni copia dei dati. Grazie a questo protocollo, tali attività possono essere eseguite senza interrompere l'esecuzione delle transazioni attive (includendo quelle iniziate durante la fase di riconfigurazione).

**Replicazione Polimorfica** Affinche la piattaforma sia in grado di adattarsi e raggiungere, quindi, livelli prestazionali ottimali, è stato deciso di estende-

re Infinispan integrando il framework MorphR. Tale framework, come gia descritto precedentemente, permette, attraverso la definizione di interfacce, di utilizzare un protocollo di replicazione, rispetto ad un altro, a tempo d'esecuzione. MorphR si fa carico di gestire i vari protocolli e il loro switch; è a carico del componente Autonomic Manager, invece, la decisione di quando effettuare un cambio di protocollo e quale protocollo utilizzare.
Infispan è stato arricchito con l'implementazione di tre protocolli di replicazione, varianti del protocollo GMU:

**2PC** Questa versione, multi-master, permette ad ogni nodo di processare sia transazioni di tipo read-only, sia transazioni di update; essa è basata su una variante del classico schema Two Phase Commit per certificare le transazioni e determinare il vector clock da attribuire a una transazione che effettua commit. Questo protocollo si comporta bene in scenari write intensive, dove però vi è una contesa tra dati limitati. In tale scenario, tutti i nodi possono processare transazioni di update, senza incorrere in un eccessivo numero di abort. Il principale punto negativo di questo protocollo è che è prono ai deadlock distribuiti al crescere della contesa;

**PB** Questa variante permette solamente ad un singolo nodo, detto master o primary, di processare le transazioni di tipo update; i nodi di backup possono esclusivamente processare le transazioni di tipo read-only. In tale modo, il nodo master regola la concorrenza tra transazioni locali, attraverso strategie di lock prive di deadlock, e propaga gli aggiornamenti ai nodi di backup. Le transazioni read-only sono processate in maniera non bloccante sui nodi di backup. All'interno di tale protocollo, il nodo master tende a divenire un collo di bottiglia negli scenari dove il workload è prevalentemente write intensive; allo stesso tempo, limitando il numero di transazioni di update concorrenti, è possibile ripristinare il carico della rete e ridurre le latenze di commit delle transazioni di update.

**TO** Similmente al 2PC, è un protocollo multi-master che processa le transazioni senza l'uso di sincronizzazione durante la fase di esecuzione. Tuttavia, a differenza di 2PC, l'ordine di serializzazione non è determinato dall'ordine di acquisizione dei lock, bensì da servizi di Total Order Multicast, capaci di stabilire un ordine totale tra le transazioni

committanti. In tale variante, gli schemi di prepare sono disseminati tra i nodi, usando primitive TOM. Il lock dei dati viene acquisito dalla transazione non appena vi è la consegna di un messaggio di prepare, la quale segue l'ordine di consegna stabilito da TOM. L'uso di TOM assicura un accordo sull'ordine di acquisizione dei lock tra i nodi coinvolti nella fase di commit, e quindi evita la possibilità di incorrere in deadlock distribuiti. Tuttavia, adottando la versione basata su TOM, si introducono larghi overhead rispetto ai due protocolli precedenti. All'interno di scenari di workload write intensive, TO presenta maggiore scalabilità rispetto PB, ma presenta alto abort rate in scenari ad alta contesa.

### 2.4.2 Autonomic Manager

Il componente Autonomic Manager è il responsabile di effettuare il *self-tuning* della Data Platform. All'interno della piattaforma Cloud-TM i meccanismi di auto-ottimizzazione si estendono tra i vari livelli della piattaforma. Specificamente, Cloud-TM si basa su un numero di meccanismi di self-tuning complementari che puntano ad ottimizzare automaticamente, sulla base delle qualità del servizio (o Quality of Service, QoS) e dei vincoli di costo, le seguenti funzonalità/parametri:

- la scala della sottostante piattaforma, cioè il numero e il tipo di macchine sulla quali il Data Platform è in esecuzione;
- il grado di replicazione dei dati, cioè il numero di repliche di ogni dato memorizzato nella piattaforma;
- il protocollo usato per assicurare la consistenza dei dati;
- la strategia di posizionamento dei dati e politica di distribuzione delle richieste, con l'obiettivo di massimizzare l'accesso ai dati in locale.

Come mostrato nel diagramma in Figura 2.1, lo Autonomic Manager espone delle API attraverso le quali è possibile specificare e negoziare i requisiti sulle QoS e i vincoli di costi.
Lo Autonomic Manager fa uso di meccanismi di monitor (WPM, Sezione 2.6.1) capaci di tracciare sia l'utilizzo delle risorse di sistema (come CPU,

memoria, rete, dischi) sia il carico dei singoli sottocomponenti della Data Platform (come il meccanismo di controllo di concorrenza, replicazione dei dati, meccanismi di distribuzione, ecc.) e la loro efficienza.
Il flusso di dati raccolti dal framework di monitoring WPM è filtrato ed aggregato dal WorkloadAnalyzer, il quale genera informazioni sul profilo del workload

## 2.5 Benchmark per memorie transazionali

Gli strumenti di benchmarking nascono dalla necessità di fornire una misura, veritiera, delle prestazioni di un sistema. Tali strumenti sono di fondamentale importanza sia per chi progetta ed implementa il sistema, permettendo di verificare l'esattezza dei modelli adottati, sia per l'utente, che può comparare i risultati di differenti prodotti e selezionare quello che ritiene migliore per le proprie necessità.
Anche nell'ambito delle memorie transazionali un discreto numero di benchmark sono stati proposti. *Microbench*, rappresenta una suite di microbenchmark, inizialmente utilizzata per comparare le performance di implementazioni rispettivamente *lock-based* (ad esempio, TinySTM [8]) e *lock-free*.
Un altra soluzione per il *benchmarking* di sistemi di TM è *STMBench7* [10], il quale si candida come promettente evoluzione dei microbenchmark, definiti *toy benchmark*, non sono sufficienti per effettuare una valutazione completa e realistica del sistema. *STMBench7* opera su un grafo composto da milioni di oggetti interconnessi tra loro. Si caratterizza da un elevato numero di operazioni, con vari scopi e complessità, ed è in grado di simulare un gran numero di scenari reali.
Della stessa idea, per quanto riguarda i *microbenchmark*, sono *Chi Cao Minh et al.* in [4]. Essi definiscono i microbenchmark strumenti non idonei a valutare le caratteristiche di un sistema di TM.
Come alternativa, gli autori introducono STAMP, una suite di benchmarking formata da 8 benchmark con 30 differenti insiemi di configurazioni e dati di input, in grado di generare differenti comportamenti.
La suite STAMP è stata creata seguendo 3 principi guida, riconosciuti dagli autori, per creare uno strumento di benchmarking efficace e completo:

**Breadth** STAMP è formato da una varietà di algoritmi e domini applicativi. In particolare, sono forniti quelli non facilmente parallelizzabili se non

attraverso l'uso di primitive di sincronizzazione. In questo modo, tali algoritmi/domini, possono beneficiare del controllo di concorrenza ottimistico tipico delle memorie transazionali.

**Depth** STAMP copre un vasto range di comportamenti transazionali come, per esempio, vari gradi di contesa, brevi e lunghe transazioni o insiemi di read/write di differenti dimensioni. Inoltre, sono presenti delle applicazioni che fanno uso di transazioni a grana grossa, trascorrendo significanti porzioni di tempo dentro la transazione.

**Portability** STAMP può essere fatto funzionare, facilmente, su varie classi di sistemi di memorie transazionali, sia esse hardware, software o ibride.

Come detto, STAMP presenta 8 differenti applicazioni, le quali coprono una grande varietà di domini e caratteristiche transazionali, variando la lunghezza delle transazioni, la grandezza degli insiemi di read/write e la *contention*.

**Bayes** implementa un algoritmo per apprendere la struttura di reti bayesiane a partire dai dati osservati. La rete è rappresentata sotto forma di grafo diretto aciclico (DAG), con un nodo per ogni variabile stocastica e un arco per ogni dipendenza condizionale tra le variabili. Inizialmente non sono presenti dipendenze; l'algoritmo, assegna ogni variabile ad un thread e incrementalmente apprende le dipendenze analizzando dati. Le transazioni vengono utilizzate per calcolare ed aggiungere nuove dipendenze.

**Vacation** emula un sistema di prenotazioni online per viaggi, attraverso un sistema di processamento di transazioni. L'implementazione è formata da un insieme di alberi che tengono traccia dei clienti e loro prenotazioni per vari viaggi. Durante l'esecuzione del *workload*, vari clienti (*thread*) effettuano un numero di operazioni, di vario tipo (riserva, cancellazione e aggiornamento), che agiscono sul *database* del sistema. Ogni azione viene racchiusa all'interno di una transazione, affinchè sia mantenuta la consistenza del database.

Un altro strumento per il *benchmarking* di TM è il *LeeBenchmark*, presentato in [1]. *LeeBenchmark* è una suite composta da varie implementazioni, sequenziale, *lock-based*, e transazionale, dell'algoritmo di Lee, permettendo

un confronto delle performace diretto ed immediato. L'algoritmo di Lee è uno dei primi che cerca di risolvere il problema di produrre automaticamente circuiti di connessione tra componenti elettronici. Il suo interesse, nell'ambito del *benchmarking*, deriva dall'elevato grado di parallelismo dovuto dalla presenza all'interno dei circuiti reali di migliaia di rotte, ognuna concorrentemente instradabile.

### 2.5.1 Benchmark per memorie transazionali distribuite

Nel contesto delle piattaforme distribuite, l'attività di *benchmarking* dei sistemi transazionali non è ancora ben supportata da strumenti di riferimento, capaci di valutare correttamente tutte le sfaccettature di un sistema.
I ricercatori attivi nell'area delle DTM hanno reagito a questa mancanza adottando tecniche differenti. Da una parte, si è cercato di adattare le soluzioni di *benchmarking* non distribuite, precedentemente descritte, al nuovo scenario, richiedendo però la risoluzione di inevitabili problematiche, di seguito illustrate.

**Identificazione degli oggetti** In un ambiente non distribuito, è possibile accedere ad un oggetto presente in memoria attraverso l'uso di puntatori. Contrariamente, all'interno di un ambiente distribuito, il caratterizzato dalla partizione e disseminazione dello stato applicativo, l'uso di puntatori non è più permesso. È necessario quindi modificare il benchmark per far in modo che ogni oggetto sia associato con un valore identificativo univoco.

**Serializzabilità degli oggetti** All'interno di piattaforme distribuite gli oggetti possono essere inviati, per mezzo della rete, da nodo a un altro, richiedendo l'utilizzo di meccanismi di serializzazione, non ben supportati in tutti i linguaggi di programmazione (ad esempio C, C++). Inoltre, anche in linguaggi ad alto livello (Java), la serializzazione di oggetti complessi non è una attività banale.

**Fase di popolazione coerente** In alcuni *benchmarking tool* sono previste delle fasi iniziali responsabili di poplare la memoria con dei dati fittizi, utilizzati nelle seguenti fasi per eseguire le operazioni proprie dell'applicazione di test. Spesso vengono utilizzate tecniche non deterministiche, basate su generazione di valori *random*, per determinare tali dati

fittizi. Per evitare quindi inconsistenza dello stato all'inizio del test, è necessario introdurre degli schemi deterministici per la generazione di dati fittizi su ogni nodo.

Da un'altra parte, si è cercato di riadattare soluzioni di benchmarking adottate nell'area dei *database management system*, sistemi affini con quelli di DTM, nei quali i concetti di replicazione e distribuzione sono stati già ampliamente analizzati.

**TPC-C [19]** è stato progettato per effettuare il benchmarking di *on-line transaction processing* (OLTP). Tali sistemi sono caratterizzati da transazioni brevi, capaci di registrare eventi di business i quali richiedono alta disponibilità, consistenza e brevi tempi di risposta. I sistemi OLTP garantiscono quindi che la richiesta di un servizio sia gestita in un periodo temporale prevedibile, il più possibile vicino al *real time*.
Date le molte affinità tra i sistemi OLTP e quelli di DTM, recentemente TPC-C è stato utilizzato per valutare le prestazioni di quest'ultimi.
TPC-C appartiene alla suite di benchmark fornita dal *Transaction Processing Performance Council* (TPC), il quale si pone l'obiettivo di definire un insieme di requisiti funzionali eseguibili su ogni sistema orientato alle transazioni, a prescindere dall'hardware o sistema operativo adottato. Ogni produttore di sistemi OLTP, sia esso *open* o proprietario, dovrà dimostrare che il proprio sistema è in grado di rispettare tali requisiti, permettendo all'utente finale di paragonare, 1-a-1, prodotti differenti.
Il benchmark TPC-C simula un ambiente dove un insieme di operatori terminali eseguono delle transazioni su un database per portare a termine le attività tipiche di un sistema di gestione di ordini.
Più in particolare, il modello di business di TPC-C riguarda un fornitore di pezzi all'ingrosso, il quale opera su un numero di magazzini e rispettive aree di vendita. Il benchmark è progettato per scalare in maniera proporzionale alla crescita del fornitore, aggiungendo nuovi magazzini. Quando nuovi magazzini sono aggiunti devono essere rispettati alcuni vincoli: ogni magazzino deve fornire 10 aree di vendita e ogni area deve servire 3000 clienti.
Un operatore può selezionare, da un'area di vendita, una delle 5 operazioni (o transazioni) offerte dal sistema di gestione degli ordini del fornitore. La

| Transazione | Descrizione |
|---|---|
| New-order | inserisce un nuovo ordine dal cliente |
| Payment | aggiorna il bilancio di un utente a fronte di un pagamento |
| Delivery | consegna gli ordini (svolto come una operazione batch) |
| Order-status | recupera lo stato degli ordini di un cliente più recenti |
| Stock-level | controlla l'inventario di un magazzino |

**Tabella 2.1: Tipi di transazioni disponibili all'interno del *tool* TPC-C**

frequenza con cui sono create nuove transazioni è modellata da scenari realistici.
Tali attività (sintetizzate nella Tabella 2.1) includono l'inserimento ed l'evasione di ordini, la registrazione dei pagamenti, il controllo dello stato degli ordini e l'analisi dei livelli di stock di un magazzino.
Le transazioni incaricate di inserire nuovi ordini sono le più frequenti; ogni nuovo ordine, in media, è composto da 10 pezzi. Nonostante ogni magazzino ha disponibilità di 100.000 pezzi in uno stock locale, alcuni ordini( 10%) devono essere evasi da un magazzino differente.
Anche le transazioni incaricate di registrare i pagamenti ricevuti da un cliente sono frequenti all'interno del modello.
Meno frequenti sono invece le transazioni per richiedere lo stato di un ordine precedentemente inserito, processare una partita di 10 ordini da evadere o per interrogare il sistema per esaminare i livelli di stock del magazzino.
La metrica di performance fornita da TPC-C misura il numero di ordini totalmente evasi al minuto, la quale rappresenta, non solo un mero numero di transazioni eseguite in un DBMS, ma rappresenta quante operazioni di business possono essere processate al minuto. Tale dato è in effetti quello di maggior interesse per l'utente finale.

## 2.6 Progetti software relazionati al lavoro svolto

Lo scopo di questa sezione è di introdurre i progetti software, di rilevante importanza, direttamente coinvolti durante lo sviluppo del progetto. A seguire mi limiterò a descriverne le funzionalità e le loro caratteristiche peculiari, riproponendomi di posizionarli all'interno del mio lavoro nella capitolo successivo.

### 2.6.1 Workload and Performance Monitor

*Workload and Performance Monitor* (WPM) è un sottosistema software, rilasciato sotto licenza open source, sviluppato nel contesto del progetto europeo Cloud-TM 2.4. Il suo compito è di raccogliere informazioni statistiche sul carico e sulle performace dei vari componenti distribuiti su vari livelli della piattaforma Cloud-TM.

**Lattice monitoring framework** La dorsale del framework WPM è rappresentata da Lattice, un framework di recente sviluppo, disegnato specificamente per raggiungere alta scalabilità, elasticità e adattabilità, requisti chiave delle infrastrutture di larga scala. Lattice fà uso di un ridotto numero di componenti, ognuno responsabile di portar a termine uno specifico compito.
Le interazioni tra i componenti del framework Lattice si basano sullo schema producer-consumer; un producer accede a sorgenti di dati per mezzo di sonde capaci di campionare valori di di grandezze monitorate, incapsularli all'interno di messaggi appositi e accodarli all'interno di una coda tra sorgente di dati e relativa sonda. Il campionamento dei valori può avvenire periodicamente o come conseguenza di qualche evento. I messaggi presenti all'interno delle code sono inviati al consumer. Producer e consumer, hanno la capacità di interagire per riconfigurare le modalità di operazione interne. Sono definiti tre canali logici per l'interazione tra i due componenti:

- data plane;
- info plane;
- control plane.

Il *data plane* è usato per trasferire i messaggi di dati, il cui *payload* è formato da un set di misurazioni, ognuna raccolta all'interno di uno specifico campo. Poichè la struttura dei messaggi, intesa come quantità di campi e loro significato, è predeterminata, è possibile trasmettere esclusivamente i valori dei dati campionati, eliminando overhead legato alle etichette. Tuttavia, il messaggio presenta un header, contenente informazioni di identificazione della sorgente e timestamping. Attraverso l'interazione supportata dallo *info plane*, è possibile riconfigurare dinamicamente tale struttura, permettendo frequenti scambi di messaggi di dati.

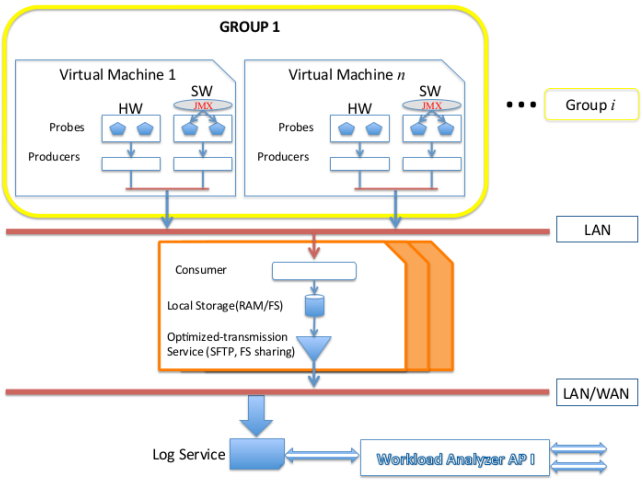


**Fig. 2.2: Architettura di WPM**

Infine, il *control plane* può essere usato per scatenare delle riconfigurazioni del componente producer, introducendo, ad esempio, un cambio nella frequenza con cui campionare.
Il meccanismo di trasporto, di supporto per i vari livelli, è disaccoppiato dalla architettura dei componenti producer/consumer. Specificatamente, i dati sono disseminati tra i componenti attraverso meccanismi di distribuzione configurabili, quali IP multicast o publish/subscribe, i quali possono essere cambiati, senza interessare le configurazioni dei restanti componenti.
Il framework è capace di supportare multipli componenti producer/consumer, fornendo la possibilità di gestire dinamicamente le configurazioni delle sorgenti di dati, attivazione/disattivazione delle sonde, frequenza di invio dei dati e ridondanza.

**Organizzazione architetturale di WPM** La Figura 2.2 mostra l'organizzazione architetturale di WPM, la quale è stata pensata per fornire supporto a due principali funzionalità:

- statistical data gathering (SDG);
- statistical data logging (SDL).

La funzionalità di raccolta di dati statistici (SDG) prevede una istanza del framework Lattice con il supporto di specifiche sonde per la piattaforma

Cloud-TM, la cui infrastruttura prevede delle macchine virtuali. È possibile suddividere le risorse da monitorare in due gruppi, risorse hardware/virtualizzate e risorse logiche. Le statistiche per il primo gruppo di risorse dono raccolte e collezionate direttamente interagendo direttamente con il sistema operativo, o in alcuni casi, attraverso delle librerie disaccoppiate; per quanto riguarda le risorse logiche, relazionate con vari livelli della piattaforma Cloud-TM, sono presenti delle specifiche sonde capaci di collezionare statistiche attraverso il protocollo JMX. Le statistiche raccolte attraverso tali sonde sono inviate al componente producer, attraverso le facilities fornite nativamente dal framework Lattice. Ogni produttore è accoppiato con un o più sonde, ed è responsabile della loro gestione.
Il consumer è il componente del framework Lattice, il quale riceve i dati dai producer, attraverso vari sistemi di messaggistica.
Nella Figura 2.2 si ipotizza uno schema di funzionamento, dove il consumer si fa carico di gestiore uno o più gruppi di macchine appartenenti alla stessa rete LAN. Oltre a collezionare dati dai produttori, il consumer si fa carico di effettuare una elaborazione locale con l'obiettivo di produrre un flusso di dati idoneo per essere fornito come input al Log Service, capace di supportare la funzionalità di logging dei dati.
L'interazione tra i due blocchi SDG e SDL avviene attraverso il servizio detto *optimized-transmission*; tale servizio si affida a differenti soluzioni a seconda della dislocazione fisica dei due componenti. Inoltre, possono essere attivati schemi di compressione per garantire minori latenze e minor spazio di memorizzazione.
Il Log Service è il componente logico responsabile di aggregare, memorizzare e rendere disponbili i dati raccolti dall'insime di consumer presenti nella piattaforma. Il Log Service è in grado di memorizzare i dati in ingressi in vari tipi di storage, quali file system o key-value store distribuiti.

**Log Service** Il Log Service è il componente, interlo all'architettura del framework WPM, è capace di raccogliere i flussi di dati prodotti dai componenti consumer del framework Lattice, e svolge due principali compiti:

- assicura che il dati raccolti siano resi persistenti e archiviati per il processamento e l'analisi off-line;

- espone i dati ricevuti per una eventuale analisi e processamento online.

È possibile selezionare la modalità di persistenza dei dati; il Log Service offre due principali meccanismi, il più semplice, basato su file system e il più affidabile e scalabile basato su Infinispan (Sezione 2.4.1). La prima soluzione permette di memorizzare all'interno di file, compressi e all'occorrenza crittografati, i messaggi individualmente collezionati. La seconda soluzione, invece, richiede il parsing dei messaggi ricevuti e la successiva memorizzazione dei singoli valori all'interno di coppie chiave-valore, memorizzate in una istanza di Infinispan; attraverso i suoi meccanismi di replicazione e distribuzione, massimizzano la scalabilità e la tolleranza ai guasti di tale schema di persistanza.
Inoltre, WPM espone un insieme di API che permettono di notifica, in maniera efficiente, applicazioni esterne riguardo la disponibilità di nuove informazioni all'interno del Log Service. Per tale scopo, sono esposte un insieme di interfacce, aderenti al pattern Observer, il quale permette alle applicazioni esterne di registrare vari tipi di *listener*, cioè metodi invocati al verificarsi di specifici eventi, quali ad esempio il join/leave/failure di nodi all'interno della piattaforma o eventi di notifica di nuovi dati relativi o ad un singolo nodo o a tutti i nodi della piattaforma. Le interfacce sono disponibili sia localmente, cioè su applicazioni in funzione sulla stessa JVM del LogService, sia remotamente, esposte attraverso Java RMI; per tale obiettivo WPM dispone di un plugin, chiamato WPM Connector, il quale, una volta importato all'interno della applicazione esterna, nasconde le complessità di comunicazione.

### 2.6.2 OpenStack

OpenStack è una piattaforma *Infrastructure-as-a-Service* di cloud computing, rilasciata sotto licenza *open-source*, la quale permette di fornire risorse computazionali *on-demand* attraverso la fornitura e la gestione di reti di macchine virtuali.
L'obiettivo che OpenStack si pone è quello di fornire infrastrutture di cloud computing elastiche e scalabili, di tipo pubbliche e private, di dimensioni variabili. Non ponendo vincoli su hardware o software proprietario, Open-

Stack è altamente flessibile e capace di integrarsi con sistemi *legacy* e/o con tecnologie di terze parti.
OpenStack è composto da un insieme di tecnologie software open-source attraverso le quali è possibile gestire la propria infrastruttura cloud per l'elaborazione e lo storage dei dati in maniera scalabile. Tra i principali progetti che aiutano ad assolvere le sue funzionalità principali troviamo:

**OpenStack Compute** utilizzato per effettuare computing, quindi utile per creare cloud di macchine virtuali

**OpenStack Networking** necessario per gestire la rete, in ottica cloud

**OpenStack Storage** per lo storage, con un servizio ad oggetti, del tutto simile ad Amazon S3

Inoltre, OpenStack inoltre fornisce un utile componenete per la gestione del cloud: una applicazione web, la *Dashboard*, altamente *user-friendly* grazie all'uso di tecnologie d'avanguardia, integrata con le API esposte dai progetti sottostanti.
Il successo di OpenStack è dovuto al grande numero di organizzazioni che hanno sin da subito aderito e supportato il suo sviluppo, tra le quali HP, Intel, AMD, VMware, IBM, ecc., oltre ad avere una vasta comunità in grado di aiutare le organizzazioni ad eseguire i processi di installazione e configurazione delle cloud per l'elaborazione virtuale o l'archiviazione dei dati.
Per quanto riguarda il lavoro di tesi, nella prima fase del progetto si è provveduto ad installare un ambiente cloud, basato su OpenStack, all'interno del cluster disponibile, composto da 10 macchine. Durante tutto il resto del progetto è stato, OpenStack è stato utilizzato per effettuare lunghe ed intense sessioni di test.

### 2.6.3 DeltaCloud

Deltacloud è un progetto *open-source*, sviluppato da Red Hat e Apache Software Foundation, in grado di fornire un livello software attraverso il quale è possibile interagire, in maniera standardizzata, con un gran numero di piattaforme cloud Infrastructure-as-a-Service, per mezzo di astrazioni.

Il problema che Deltacloud si propone di limitare è la proliferare di *Application Programming Interface* (API) proprietarie, scritte in modo da permettere l'interazione con un ristretto numero di soluzioni IaaS.
Deltacloud dispone di un *front-end*, REST-*based*, composto da una serie di API in grado di gestire risorse presenti all'interno di varie *cloud*.
L'interazione con la specifica IaaS, avviene, in Deltacloud, attraverso un modulo di *back-end* composto da *driver* specifici capaci di comunicare con le API native del *cloud provider*. Per incentivare l'uso di Deltacloud e il processo di standardizzazione, all'interno di *Deltacloud Core Framework* sono forniti i componenti basilari utili per l'implementazione di *driver* per comunicare con nuove piattaforme cloud.
Durante lo sviluppo del progetto di tesi, abbiamo provveduto a configurare Deltacloud per l'interazione con la piattaforma IaaS fornita da OpenStack, affinchè fosse possibile gestire le risorse attraverso delle chiamate REST effettuate da moduli attuatori, descritti nel prossimo capitolo

# CAPITOLO 3

# ELASTIC DTM BENCH

L'introduzione di efficienti tecniche di *elasting scaling* all'interno delle piattaforme transazionali distribuite permette di raggiungere un duplice obiettivo: da una parte aiuta a garantire le qualità del servizio (QoS) predeterminate con l'utente e, allo stesso tempo, permette di minimizzare i costi operazionali dovuti all'uso di risorse computazionali.
Di recente, vari gruppi di ricerca hanno centrato il loro interesse sul disegno di specifiche soluzioni per fare *elasting scaling*, ognuna delle quali presenta caratteristiche più o meno promettenti.
Purtroppo però, come testimoniato dall'analisi svolta nel capitolo precedente, al giorno d'oggi, la comunità scientifica ancora non dispone di uno strumento in grado di valutare sperimentalmente le tecniche di *elasting scaling* finora sviluppate, integrate all'interno di un sistema di DTM elastico.
In questo capitolo è presentata una parte del lavoro svolto nel progetto di tesi, il quale cerca di colmare tale mancanza attraverso il disegno e l'implementazione di ELASTIC DTM BENCH, un framework di benchmarking per sistemi di memorie transazionali distribuite elastiche Java-based.

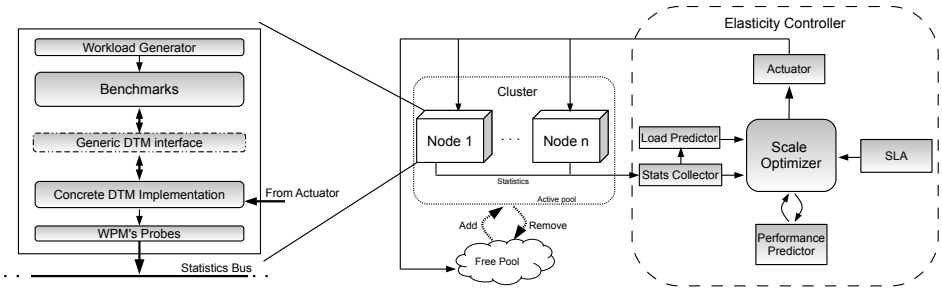


**Fig. 3.1: Architettura globale di ELASTIC DTM BENCH**

## 3.1 Requisiti

Lo sviluppo di ELASTIC DTM BENCH ha richiesto una fase iniziale di analisi dei requisiti, atta a identificare le caratteristiche che ELASTIC DTM BENCH debba possedere. Tale fase è stata portata a termine attraverso l'analisi di strumenti di benchmarking già presenti nella comunità scientifica, sia destinati a valutare le performance di piattaforme transazionali distribuite, sia per valutare *Data Base Management System* (DBMS).
I requisiti individuati a conclusione di tale fase, che ELASTIC DTM BENCH deve presentare, sono di seguito presentati:

1. deve disporre di un insieme di benchmark, per piattaforme di DTM, eterogenei e permettere l'aggiunta di altri;
2. deve assicurare e facilitare la portabilità delle applicazioni di benchmarking tra varie piattaforme di DTM;
3. deve fornire varie strategie per la generazione di carichi di lavoro variabili in funzione del tempo;
4. deve permettere il *plug-and-play* di meccanismi di *elasting scaling* forniti dall'utente;
5. deve garantire e facilitare la portabilità tra differenti *provider* di piattaforme cloud IaaS.

## 3.2 Architettura

Il diagramma in Figura 3.1 rappresenta una visione ad alto livello del framework ELASTIC DTM BENCH. Tale diagramma è di aiuto sia per identifica-

re la dislocazione del singolo componente, sia le relazioni che esso instaura con il resto dell'architettura.
Possiamo individuare due componenti distinti principali all'interno dell'architettura di ELASTIC DTM BENCH:

- *DTM Benchmarking Framework*
- *Elasticity Controller*

Il primo componente, *DTM Benchmarking Framework*, è responsabile di valutare le performance della piattaforma transazionale eseguendo specifiche applicazioni di benchmark; ricordiamo che, come evidenziato nel Capitolo 2, le prestazioni di una piattaforma transazionale sono strettamente relazionate alla dimensione della piattaforma e al profilo del carico di lavoro in ingresso nella piattaforma.
Il secondo componente identificato, *Elasticity Controller*, si incarica invece di automatizzare lo scaling automatico della piattaforma. Il suo disegno, altamente modulare, è stato voluto per dare la possibilità all'utente di integrare e valutare algoritmi di auto-scaling propri.
Nel resto della sezione verranno analizzati nel dettaglio i componenti finora individuati. Per quanto riguarda lo *Elasticity Controller*, la descrizione si limiterà a presentare i compiti e gli obiettivi che i vari sotto-moduli tentano di raggiungere all'interno del framework; nel Capitolo 4 saranno invece presentate delle implementazioni concrete di tali moduli, le quali sono state integrate all'interno del componente *Autonomic Manager* del progetto Cloud-TM.

## 3.3 DTM Benchmarking Framework

Il componente *DTM Benchmarking Framework* presente all'interno di ELASTIC DTM BENCH permette di effettuare una valutazione delle prestazioni di una piattaforma transazionale distribuita, espresse con il numero di transazioni correttamente eseguite (*throughput*), eseguendo su ogni nodo della piattaforma una specifica applicazione di benchmark decisa dall'utente.
Per effettuare il benchmarking di piattaforme distribuite, *DTM Benchmarking Framework* utilizza una architettura di tipo Client/Server, la quale permette di controllare vari nodi, detti Slave, attraverso un singolo nodo di

controllo, detto Master. Tale architettura è stata ereditata dall'integrazione del progetto open-source Radargun, progetto software, creato, sviluppato e mantenuto da RedHat per il benchmarking di cache/datagrid distribuite transazionali Java-based. Tuttavia, il componente *DTM Benchmarking Framework* introduce delle nuove caratteristiche all'intero di Radargun, le quali saranno presentate a seguire.
Nel resto dell'elaborato, spesso è usato il termine *cache distribuita* (semplicemente *cache*, o più precisamente, *istanza di cache*), valido nel contesto di piattaforme cache/datagrid distribuite, per indicare una generica memoria transazionale distribuita.

### 3.3.1 Elastic Master

In una istanza di benchmark, il componente Master inizializza e gestisce gli Slave durante tutta la durata del test. Generalmente, una tipica esecuzione di benchmark distribuito presenta le seguenti fasi:

- un numero di nodi vengono avviati; un nodo corrisponde ad una istanza di memoria transazionale distribuita;
- si attende che tutti i nodi siano capaci di comunicare, formando quindi un cluster;
- viene eseguita una fase di *warm-up*, con il fine di riprodurre, all'interno della piattaforma, un ambiente il più simile possibile a quello di produzione;
- viene eseguita l'applicazione di benchmark: il codice applicativo sarà eseguito su ogni nodo presente nel cluster. Durante l'esecuzione del test, ogni nodo registra informazioni relative alle performace;
- al termine dell'esecuzione del benchmark, tutte le statistiche relative alle performance su singolo nodo vengono raccolte e quindi aggregate.

In *DTM Benchmarking Framework*, il componente master è stato arricchito fornendogli gli strumenti adatti per garantire e gestire lo scaling up/down dei nodi a tempo di esecuzione; per tale motivo è stato chiamato *ElasticMaster* ed estente il modulo *Master*, presente all'interno del framework Radargun. Tra i tanti compiti svolti dal master, possiamo evidenziare quelli di:

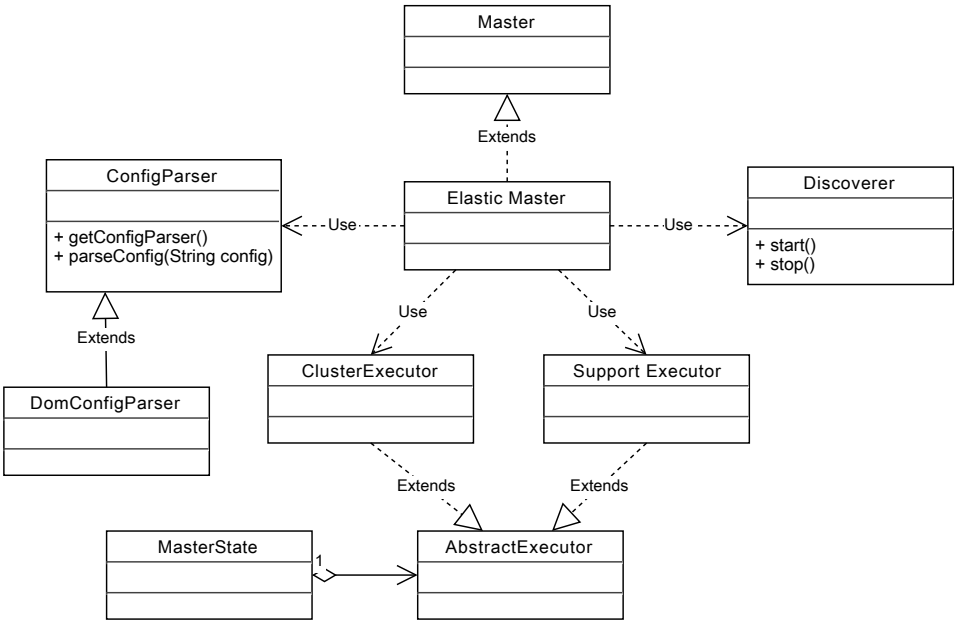


**Fig. 3.2: Diagramma delle classi per il componente Master**

- effettuare il parsing del file di configurazione del framework;
- attendere un numero di connessioni minimo da parte di slave;
- inviare agli slave, in broadcast, unità di lavoro, dette stage;
- sincronizzare l'esecuzione degli stage tra gli slave;
- fornire supporto per operazioni di scaling up/down

Nel diagramma delle classi in Figura 3.2 è presentata la suddivisione in moduli del componente Master. Il modulo *ConfigParser* espone l'interfaccia utile per leggere la configurazione, fornita dall'utente tramite file esterno, ed inizializzare le strutture dati necessarie per completare una run di benchmark. Attualmente il framework dispone di una implementazione del parser, *DomConfigParser*, capace di leggere un file XML e rappresentarlo in memoria, secondo il modello Document Object Model (DOM).
Per mezzo del modulo *Discoverer*, il Master si mette in attesa di un numero minimo, configurabile, di connessioni da parte di Slave. Al raggiungimento di tale soglia il master inizia ad inviare, agli slave connessi, singole unità di lavoro, dette stage. L'invio di ogni *Stage* avviene in broadcast sincrono,

**Listing 3.1: Semplificazione dell'algoritmo svolto dal componente Master**

```
for (Stage stage: List<Stage>) {
    if (!isDistributedStage(stage)){
      stage.executeOnMaster();
      continue;
    }
    stage.initOnMaster();
    broadcast(stage);
    Set<Resp> responses = readRespFromAllSlaves();
    boolean shouldContinue = stage.processResponses(
  responses);
    if (!shouldCountinue) break;
  }
```

cioè il flusso d'esecuzione sul master è bloccato finchè tutti gli stage non notificano la fine dello stage stesso inviando un messaggio di *Acknowledge*. I moduli *Executor* sono i principale responsabili della gestione e dell'inoltro degli stage sugli slave. Come evidenziato in figura, distinguiamo due tipi: il primo, detto *ClusterExecutor*, si fa carico di gestire l'insieme di slave già appartenenti al cluster; il secondo, detto *SupportExecutor*, presta supporto al primo, gestendo gli slave che si collegano al master dopo il raggiungimento del numero minimo di slave, in corrispondenza di una azione di scaling up. Essi forniscono supporto selezionando ed inviando solamente gli stage, decisi dall'utente, etichettati come *non-skippable*. Il tempo di vita del *ClusterExecutor* coincide con quello del benchmark; stesso discorso vale per l'istanza del *Discoverer*, il quale, durante tutta l'esecuzione del benchmark acconsente, o meno, il join di nuovi nodi valutando l'etichetta *scalable* presente sullo stage in esecuzione dal ClusterExecutor; differentemente, le molteplici istanze del *SupportExecutor* vengono create su richiesta quando uno slave effettua il join a runtime e vengono eliminate non appena tali slave ricevono lo stesso stage in esecuzione sul *ClusterExecutor*.

### 3.3.2 Slave

Il ruolo che il componente *Slave* assume all'interno del framework è prevalentemente passivo: esso si limita ad stabilire la connessione con il nodo master, deserializzare gli stage ricevuti tramite connessione di rete, eseguire la logica applicativa presente al loro interno e inviare il messaggio di *Ac-*

**Listing 3.2: Semplificazione dell'algoritmo svolto dal componente Slave**

```
while (masterRunning) {
   Stage stage = readStage();
   Resp resp = stage.executeOnSlave();
   serializeRespToServer(resp);
}
```

*knowledge*. Ogni slave è eseguito come un processo indipendente il quale gestisce un singolo nodo tra quelli del cluster. Nel Listato 3.2 è presentato l'algoritmo, semplificato, applicato dagli slave.

### 3.3.3 Stage

Uno Stage è un modulo preposto a portare a termine uno specifico compito durante l'esecuzione del test di benchmark. L'utente può progettare l'andamento del benchmark specificando quali moduli devono essere eseguiti durante l'esecuzione, specificandoli all'interno del file di configurazione del framework.
Possiamo distinguere due tipi di moduli di tipo stage: i primi implementano semplicemente l'interfaccia *Stage* e sono destinati ad essere eseguiti solamente all'interno del nodo Master. Fanno parte di questa categoria gli stage capaci di generare report statistici alla fine di un benchmark. Un secondo tipo di stage implementano invece l'interfaccia *DistStage*, parzialmente presentata nel Listato 3.3. Tali stage sono destinati ad essere serializzati ed inviati, dal Master allo Slave, e quindi essere eseguiti all'interno del nodo Slave. In tali casi, lo slave deve comunicare la fine dell'esecuzione del benchmark, inviando al master un messaggio di *Acknowledge*. In alcuni casi, tale messaggio può contenere ulteriori informazioni, come ad esempio i dati campionati sulle prestazioni.
È possibile etichettare uno Stage come *skippable*, rendendolo opzionale durante l'esecuzione del test, e *scalable*, specificando che altri nodi possono effettuare il join all'interno del cluster durante l'esecuzione di tale stage. Normalmente, tutti gli stage che implementano funzionalità di supporto dovrebbero essere *non-scalable*, lasciando tale possibilità solamente agli stage responsabili di eseguire la logica applicativa capace di stressare la memoria transazionale; tali benchmark sono indicati con il nome di *Benchmark Stage*.

**Listing 3.3: Interfaccia del modulo Stage**

```
public interface DistStage extends Stage, Serializable {

    public String getId();

    public void setId(String id);

    public boolean isSkippable();

    public boolean isScalable();

    /**
     * After un-marshalling on the slave, this method will
  be called to setUp the stage with slave's state.
     */
    public void initOnSlave(SlaveState slaveState);

    /**
     * Do whatever on the slave. This will only be called
  after {@link #initOnSlave(org.radargun.state.SlaveState)}
  is called.
     *
     * @return an response that will be serialized and send
   back to the master.
     */
    DistStageAck executeOnSlave();

    /**
     * Called on master. Master state should not be passed
  to the slaves.
     */
    public void initOnMaster(MasterState masterState, int
  slaveIndex);

    /**
     * After all slaves replied through {@link #
  executeOnSlave()}, this method will be called on the
  master.
     *
     * @return returning false will cause the benchmark to
  stop.
     */
    boolean processAckOnMaster(List<DistStageAck> acks,
  MasterState masterState);

        /* ...altri metodi omessi... */
}
```

**Benchmark Stage** Tra i *DistStage* disponibili all'interno del framework di benchmarking, alcuni, chiamati Benchmark Stage, implementano la logica applicativa di benchmark e sono capaci di mettere sotto stress la memoria transazionale distribuita con il fine di valutarne le effettive performance.
Attualmente il *DTM Benchmarking Framework* dispone delle seguenti applicazioni:

- TPC-C
- Vacation
- Synthetic Benchmark

TPC-C e Vacation, già introdotti nella Sezione 2.5, non sono presenti all'interno del framework Radargun ma sono stati implementati, all'interno del *DTM Benchmarking Framework*, seguendo le rispettive specifiche.
Il *Syntethic Benchmark*, ereditato da Radargun, rappresenta una applicazione di benchmark sintetica, caratterizzata dalla presenza di transazioni (di due tipi, read-only, composta solo da operazioni di lettura, e update, dove si alternano operazioni di lettura e scrittura sui dati), generate attraverso l'uso di funzioni randomiche, che non rappresentano scenari realistici. Tuttavia, il benchmark sintetico presente all'interno del *DTM Benchmarking Framework* presenta una caratteristica in più rispetto all'originale: fornisce maggior controllo sulla composizione delle transazioni, permettendo di specificare il numero di operazioni *read/write* che le compongono.
*DTM Benchmarking Framework* fornisce, inoltre, un insieme di interfacce / classi astratte capaci di favorire la portabilità delle nuove caratteristiche introdotte, utile agli utenti interessati in sviluppare applicazioni di benchmark proprie.
Il diagramma delle classi in Figura 3.3 è di aiuto per comprendere l'organizzazione dei moduli presenti nel framework. La classe astratta *AbstractBenchmarkStage*, estendibile dall'utente per implementare nuove applicazioni di benchmark, implementa tecniche capaci di offrire il supporto per:

- simulare un modello di sistema chiuso/aperto (vedi Sezione 2.3.1);
- integrare un generatore di carico, capace di generare curve di carico specifiche, personalizzabili, per analizzare il comportamento della piattaforma sotto scenari di particolare interesse;

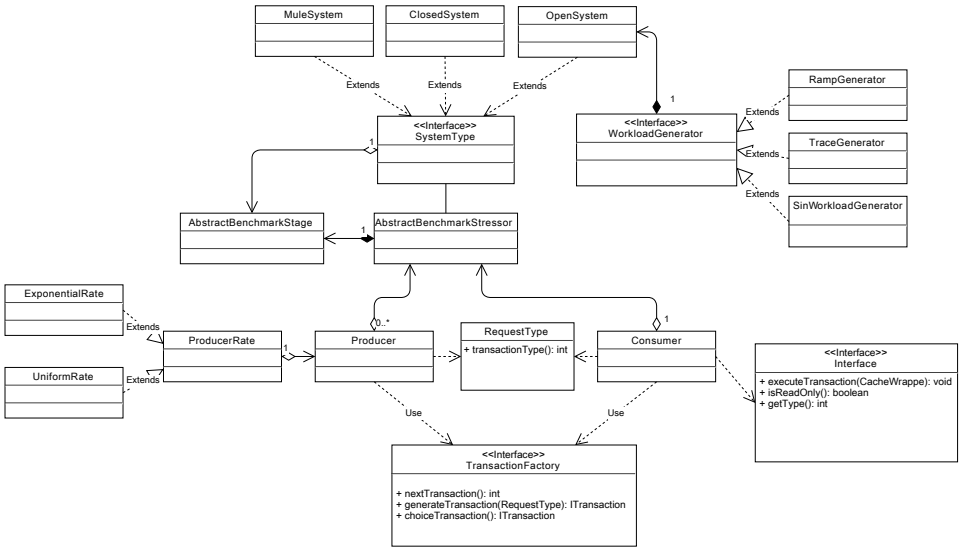


**Fig. 3.3: Diagramma delle classi**

- sincronizzare slave, aggiunti a run-time, con il resto del cluster;
- raccogliere e processare le statistiche sulle performance campionate durante l'esecuzione del benchmark;
- terminare anticipatamente l'esecuzione dell'applicazione di benchmark su un nodo, simulando una azione di *scaling down*;
- notificare i nodi del cluster attivi del completamento di una azione di *scaling up/down*;
- variare, a run-time, parametri del benchmark, come ad esempio grado di multiprogrammazione (numero di thread).

### 3.3.4 Sistemi di code e generatori di carico

Tramite il file di configurazione, l'utente ha la possibilità di effettuare il benchmark di sistemi di code sia a modello aperto, sia chiuso. L'implementazione di questa feature è realizzata all'interno della classe astratta *AbstractStressorBenchmark*, la quale è capace di istanziare e gestire insiemi di *Producer* e di *Consumer* a livello di ogni singolo nodo.
Un producer rappresenta un cliente capace di generare ed immettere nel sistema, in una struttura dati di tipo coda, delle richieste, ognuna delle quali è mappata con una specifica transazione. L'insieme dei Producer costituisce

la popolazione dei clienti; un sistema a modello aperto è caratterizzato dalla presenza di una popolazione infinita dove non è possibile definire un think-time tra una richiesta e un'altra; in un sistema chiuso, al contrario, la popolazione dei clienti è finita ed esiste un *think-time* del cliente tra due richieste successive.
All'interno del framework di benchmarking è possibile generare *arrival rate* fissi/variabili, facendo variare o la dimensione della popolazione dei producer, o la frequenza con la quale il singolo producer genera nuove richieste. Inoltre, il framework dispone di meccanismi per l'auto-regolazione dei su detti parametri al variare o dello *arrival rate* o del numero di nodi presenti nella piattaforma.
È necessario regolare la dimensione e la frequenza dei producer al variare del numero di nodi presenti all'interno della piattaforma poichè il componente che determina il carico attuale (*arrival rate*), detto *WorkloadGenerator*, è interno ai singoli nodi. In caso di join/leave di un nodo è quindi necessario ridistribuire il carico in ingresso su tutti i nodi presenti.
Il *WorkloadGenerator* è quindi il componente, interno ad ogni singolo nodo, che si fà carico di decidere quale è l'attuale *arrival rate* della piattaforma. Adottando tale scelta, è stato possibile semplificare l'architettura e quindi ridurne la complessità, evitando di introdurre componenti di poco interesse per il nostro obiettivo, quali *Load Balancer*.
Attualmente, il framework dispone di generatori di forme di carico che seguono funzioni matematica di tipo:

- gradino;
- rampa;
- sinusoidale.

Tuttavia, è possibile comporre curve di carico personalizzabili, alternando stage di benchmark associati a differenti generatori di carico. Inoltre, il framework dispone di un ulteriore generatore di carico capace di leggere delle tracce di carico da file esterno.
I Consumer, invece, rappresentano un singolo server capace di processare una richiesta/transazione. Il framework di benchmarking permette di variare, a livello di singolo nodo, il numero di consumer, per valutare il comportamento del sistema al variare del grado di multiprogrammazione.

All'interno dei consumer è implementata la logica per eseguire la transazione; inoltre essi presentano la logica per la cui è possibile, in caso di abort, tentare di rieseguire la transazione. Ovviamente, tale *feature* è attivabile o meno tramite file di configurazione.
Inoltre, è di valore menzionare un'ulteriore modalità di funzionamento, offerta dal framework, la quale permette di simulare un sistema chiuso all'interno del quale il numero di clienti può variare proporzionalmente al numero di servitori presenti nella piattaforma. Questa modalità di funzionamento è stata realizzata fornendo ai Consumer gli strumenti per autogenerare le transazioni da eseguire; è possibile configurare i Consumer in modo da simulare un think-time dei clienti, attendendo quindi uno specifico intervallo di tempo prima di procedere nel generare ed eseguire una nuova transazione. L'uso di tale modalità di funzionamento è particolarmente indicata quando si vuole valutare le performance di una memoria transazionale, piuttosto che valutare il comportamento di quest'ultima sotto specifici scenari. Sia all'interno del codice sorgente, sia nel resto dell'elaborato, ci si riferisce a questa modalità di funzionamento come *modalità mula*.

### 3.3.5 Generic DTM Interface

Il framework di benchmarking permette di nascondere i dettagli implementativi di una specifica piattaforma transazionale adottando una interfaccia generica ed astratta. In tale modo è di possibile garantire la portabilità del benchmark su un vasto numero di piattaforme transazionali eterogenee.
In dettaglio, l'interazione tra l'applicazione di benchmark e la piattaforma distribuita sottostante è mediata per mezzo di una interfaccia, la quale espone metodi standard per demarcare una transazione e gestire l'accesso ai dati.
L'interfaccia assume che la rappresentazione dei dati, interna alla piattaforma, segua il modello *key/value*, il quale ha recentemente riscosso forte successo grazie alla sua semplicità.

## 3.4 Elasticity Controller

Il componente *Elasticity Controller* permette di integrare, all'interno di ELASTIC DTM BENCH, le tecniche di *elasting scaling* progettate, dagli utenti, per

gestire la riconfigurazione della piattaforma, in risposta ad una variazione del carico di lavoro.
Attraverso una attenta analisi iniziale, è stato possibile suddividere il componente nei seguenti moduli interdipendenti:

- Load Predictor;
- Performance Predictor;
- Scale Optimizer;
- SLA Manager;
- Actuator.

Proprio per favorire e facilitare l'estensione del componente, sono state inoltre definite chiare e precise interfacce, attraverso le quali i moduli si relazionano. Nel resto della sezione, verranno presentati quali sono i compiti ed i ruoli che ogni singolo modulo svolge. Nel Capitolo 4, sarà invece presentato il componente *Autonomic Manager*, il quale estende, parzialmente, lo *Elasticity Controller*. Saranno quindi forniti dettagli su una specifica implementazione.

### 3.4.1 Load Predictor

Il *Load Predictor* è il modulo responsabile di prevedere il profilo del carico di lavoro futuro a partire dal flusso di informazioni raccolte a run-time nella piattaforma. Attraverso la sua stima è possibile riconfigurare il sistema secondo un approccio proattivo, cioè, percependo in anticipo, la variazione del carico in ingresso nel sistema, e prevenire tale cambiambento, adattando la piattaforma al carico futuro.

### 3.4.2 Performance Predictor

Il *Performance Predictor* permette di astrarre su un *oracolo*, entità capace di, dato il profilo del workload attuale, predirre il comportamento della piattaforma, quando in funzione su un determinato numero di nodi. Il comportamento è, ovviamente, espresso con indici prestazionali, come ad esempio il *throughput*, cioè il numero di transazioni correttamente eseguite nell'unità di tempo.

### 3.4.3 SLA Manager

Il modulo *SLA Manager* è responsabile di raccogliere e gestire le specifiche dei *Service Level Agreement* (SLA), strumenti contrattuali attraverso i quali vengono definite le metriche di servizio tra fornitore e cliente. Il fornitore del servizio (*provider*) dovrà rispettare tali specifiche, poichè, una volta stipulato il contratto, esse assumono un obbligo contrattuale.

### 3.4.4 Scale Optimizer

All'interno del diagramma in figura 3.1, la scelta della posizione centrale dello scale optimizer, non è stata una mera scelta artistica, ma è stata voluta per evidenziare la sua importanza all'interno dello Elasticity Controller e la sua capacità di interagire con gli altri moduli.
Il ruolo dello Scale Optimizer è di determinare il momento in cui è necessario iniziare una attività di riconfigurazione e la relativa configurazione da adottare, definendo un *trade-off* tra reattività e robustezza, affinchè sia possibile ottenere un bilancio tra efficienza e stabilità del sistema. L'efficienza dipende dall'abilità dello Scale Optimizer di rispondere in tempo alla variazione del carico, mentre la stabilità deve tener conto che politiche eccessivamente reattive potrebbero portare la piattaforma in trash.

### 3.4.5 Actuator

Tramite il modulo *Actuator*, ELASTIC DTM BENCH si interfaccia con un provider delle risorse, tipicamente piattaforme IaaS, per aggiungerne o rilasciarne.
Tipicamente è possibile far variare due aspetti di una piattaforma:

- numero di nodi;
- grado di multiprogrammazione, inteso come numero di *thread* attivi per nodo che concorrono a completare richieste;

I parametri in ingresso al modulo *Actuator*, rappresentanti i valori assunti da ogni singolo aspetto su considerato, sono decisi dal *Performance Predictor* in congiunto con lo *Scale Optimizer*.
Il compito portato a termine da questo modulo non si limita semplicemente a richiedere o rilasciare risorse alla piattaforma *cloud* sottostante, bensì lo

*Actuator* deve essere in grado di garantire coerenza tra la configurazione adottata da ogni singolo nodo e il resto della piattaforma.
Lo *Actuator* deve essere in grado di interagire con un vasto numero di piattaforme *cloud* IaaS, come ad esempio OpenStack o Amazon Elastic Compute Cloud (EC2), per ottenere le risorse computazionali. Progetti come ad esempio Deltacloud (Sezione 2.6.3), favorendo la standardizzazione attraverso astrazioni di pubblico dominio, permettono di limitare il numero di implementazioni di attuatori.

# CAPITOLO 4

# CLOUD-TM AUTONOMIC MANAGER

Il secondo obiettivo portato a termine durante lo sviluppo del progetto di tesi è stato l'implementazione e l'integrazione di vari componenti, di seguito introdotti e descritti, costituenti il modulo Autonomic Manager del progetto europeo Cloud-TM. Tale modulo è responsabile del monitoraggio e pilotaggio dello auto-tuning e scaling della relativa Data Platform di Cloud-TM.

## 4.1 Architettura

Il diagramma in Figura 4.1 fornisce una prima vista ad alto livello dell'architettura dello Autonomic Manager (AM) e mette in evidenza l'insieme di meccanismi di self-optimizing da esso supportati.
I meccanismi di self-optimization supportati dall'AM cercano di identificare i valori ottimi per un insieme di parametri/protocolli chiave della Data Platform di Cloud-TM quali:

- scala della piattaforma sottostante, cioè il numero e il tipo di nodi sui quali la piattaforma è in funzionamento;

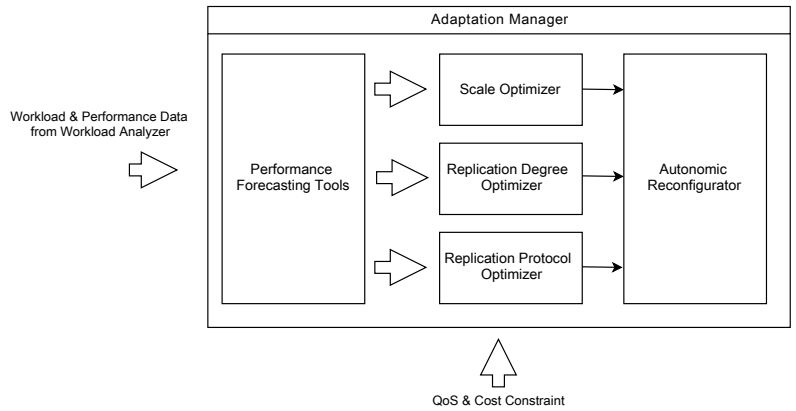


**Fig. 4.1: Diagramma architetturale del componente Workload Analyzer**

- numero di repliche di ogni dato memorizzato nella piattaforma (grado di replicazione);
- protocollo di replicazione.

Alla base dei processi di self-optimization, lo AM adotta un insieme di modelli di previsione delle performance, già presentati in dettaglio nella Sezione 2.3.1. Attraverso una loro interrogazione, lo AM può stimare le prestazioni, presentate da una specifica applicazione quando si adotta una certa configurazione, con l'obiettivo di selezionare quella che massimizzi l'efficienza.
È bene notare che i meccanismi appena descritti, possono essere adottati sia individualmente, sia in sinergia, a seconda del grado di automazione che lo sviluppare/amministratore di una applicazione punta a raggiungere.
Per quanto concerne il controllo della piattaforma, lo Autonomic Manager offre allo sviluppatore/amministratore sia una classica *Command Line Interface* (CLI), sia una interfaccia RESTful, sulla quale è stata disegnata ed implementata un'applicazione web in grado di:

- specificare e negoziare le QoS tra lo sviluppatore e i fornitori della piattaforma;
- monitorare in real-time le statistiche/informazioni sulle prestazioni e stato dell'intera piattaforma;

- configurare e monitorare il comportamento dei vari aspetti di self-tuning supportati;
- effettuare analisi *what-if* off-line, capace di stimare l'impatto delle variazioni del profilo/intensità del workload sulle performance e sulla richiesta di risorse.

In seguito verranno analizzati nel dettaglio i componenti *Workload Analyzer* ed *Adaptation Manager*, presenti all'interno dello AM, mentre per quanto riguarda il *Workload and Performance Monitor*, si rimanda alla Sezione 2.6.1 in quanto non prodotto durante il progetto di tesi.

## 4.2 Workload Analyzer

Il modulo Workload Analyzer (WA) trova posizione tra il framework di monitoring WPM e il modulo Adaptation Manager. I compiti soddisfatti dal WA all'interno della piattaforma Cloud-TM sono:

**Aggregazione dei dati** il flusso di dati prodotto dai nodi distribuiti all'interno della piattaforma e collezionati per mezzo di WPM, sono raccolti dal WA e aggregati statisticamente.

**Filtraggio dei dati e detezione di variazione nel workload/KPI** il WA integra algoritmi capaci di rilevare variazioni statistiche rilevanti dei *Key Performance Indicator* (KPI) e/o caratteristiche di carico. Queste tecniche permettono di filtrare inevitabili fluttuazioni migliorando la stabilità e la robustezza dei meccanismi di self-tuning integrati nella piattaforma.

**Predizione della richiesta delle risorse** il WA implementa algoritmi di previsione basati su time-series analysis, la quale permette di predirre l'andamento del carico futuro dando la possibilità allo Adaptation Manager di attivare proattivamente gli schemi di self-tuning. Tale funzionalità rappresenta una caratteristica fondamentale per ogni schema di adattamento proattivo, cioè schemi capaci di anticipare la variazione del workload scatenando una riconfigurazione della piattaforma. L'adozione di tali schemi è altamente desiderabile all'interno di piattaforme dove le latenze di riconfigurazione non sono trascurabili.

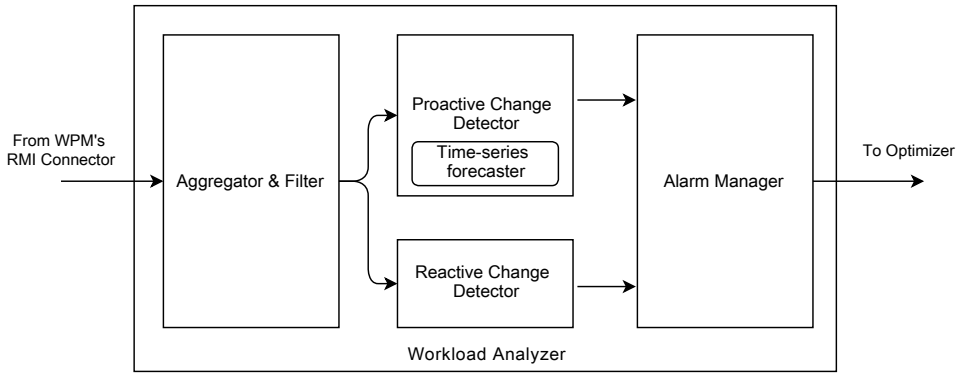


**Fig. 4.2: Diagramma architetturale del componente Workload Analyzer**

### 4.2.1 Architettura

Il diagramma in Figura 4.2 fornisce una vista d'alto livello del Workload Analyzer, il quale si compone di un sotto-modulo, detto Workload Filter and Predictor (WFF), capace di filtrare il flusso di dati raccolti dal framework di monitoring e rilevare, in maniera reattiva o proattiva (utilizzando metodi di forecasting basati su time-series), sia variazioni di workload che potrebbero causare delle inefficienze all'interno della piattaforma, sia inadempienze dei Service Level Agreement pre-accordati. Quando si rileva una variazione significativa nelle statistiche del workload corrente/predetto, si genera un evento gestito dal componente Alarm Manager, il quale può avviare un ciclo di ottimizzazione della piattaforma, basandosi sul relativo workload corrente/predetto.

**Workload Filter and Predictor** Il Workload Filter and Predictor (WFF) si incarica di individuare, statisticamente, variazioni del carico di lavoro in ingresso nella piattaforma e notificare di conseguenza il componente Adaptation Manager. Il WFF utilizza tecniche d'avanguardia sulla detezione e previsione delle performance, permettendo di:

- filtrare fluttuazioni dei segnali in ingresso non desiderabili, le quali potrebbero scatenare riconfigurazioni non necessarie o addirittura troppo frequenti, compromettendo la stabilità della piattaforma e dei meccanismi di self-tuning.
- predire gli andamenti del carico di lavoro futuro con lo scopo di adottare schemi di riconfigurazione proattivi, cioè schemi capaci di scatenare

una riconfigurazione con anticipo rispetto alla variazione del carico di lavoro, incrementando la probabilità che tale riconfigurazione sia già completata nell'istante in cui si verifica la variazione. Le strategie proattive sono di elevata importanza quando vengono effettuate delle riconfigurazioni, come ad esempio di elasting scaling, le quali richiedono la fornitura di nuove risorse e conseguente trasferimento di stato tra i nodi; tali operazioni soffrono, di fatto, di latenze non minime. In questi scenari, gli schemi di adattamento proattivi hanno mostrato prestazioni migliori rispetto ai più semplici schemi reattivi, generando un minor numero di violazioni delle QoS.

Come mostrato nel diagramma, il WFF raccoglie dati attraverso il sistema di notifica del framework di monitoring WPM, registrandosi come *listener* di eventi associati al cambio di vista della Data Platform o all'arrivo di nuovi campionamenti di statistiche.
I dati raccolti dal framework WPM sono quindi processati da un insieme di filtri estendibili, i quali permettono il calcolo di statistiche aggregate, di interesse per i meccanismi di self-opimization supportati dallo AM, quale lo Application Contention Factor (ACF), calcolato a partire dai dati raccolti all'interno della piattaforma.
A seguire, un insieme di KPIs e statistiche relative al carico di lavoro (selezionabili e configurabili per mezzo di file di configurazione), vengono passate in ingresso a due moduli: il *Reactive Change Detector* e *Time-series Forecaster*.
Il primo si incarica di riconoscere variazioni statistiche delle variabili in ingresso. Per portare a termine tale compito, integra un filtro che rileva la variazione dei valori medi su due finestre temporali consecutive. La dimensione di ciascuna finestra può essere facilmente configurando attraverso il file di configurazione, affinche sia possibile stabilire un trade-off tra il rate dei falsi allarmi, il rate di non-rivelazioni e il ritardo di rivelazione.
Il modulo Time-series Forecaster è invece utilizzato per predire l'andamento futuro del carico di lavoro, e quindi, anticipare il bisogno di future riconfigurazioni. Questo modulo può essere configurato per funzionare secondo due classici algoritmi di previsione time-series, basati rispettivamente su Polynomial fitting e filtri di Kalman.
Attraverso questi algoritmi, è possibili ottenere previsioni sui valori di vari KPI/indicatori di carico su una finestra temporale futura, detta *Look-ahead window*, calcolata tenendo in considerazione il tempo di riconfigura-

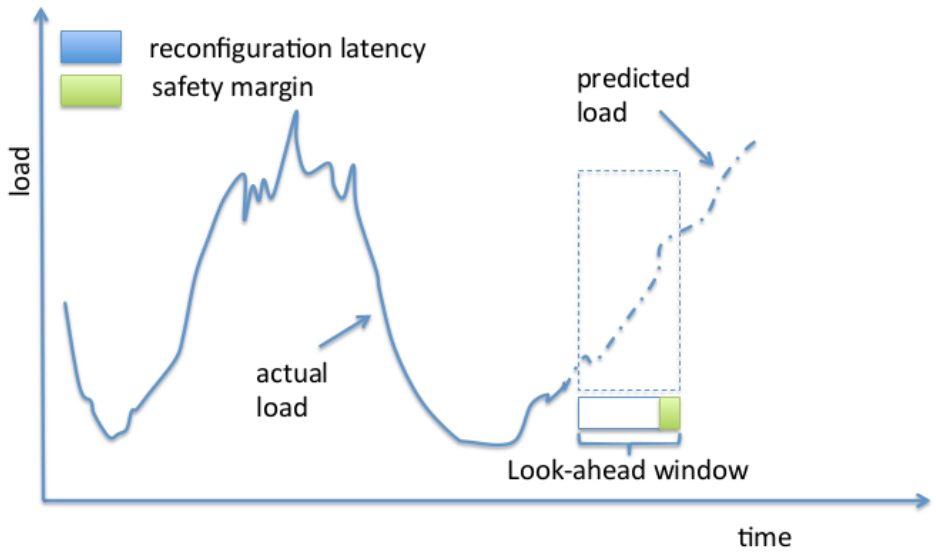


**Fig. 4.3: Rilevamento proattivo della variazione del workload.**

zione atteso, e il tempo di margine di sicurezza (mostrato nella Figura 4.3 in blue e verde).
È bene notare che il tempo di riconfigurazione atteso dipende dal tipo di riconfigurazione richiesta. Normalmente riconfigurazioni che interessano il cambio del protocollo di replicazione hanno tipicamente una durata limitata, nell'ordine di secondi. D'altra parte, una riconfigurazione che interessa il cambio del numero di nodi all'interno della piattaforma e/o il cambio del grado replicazione ha una durata variabile, dipendente da fattori quali la quantità di dati da trasferire tra i nodi, le caratteristiche hardware della piattaforma sottostante e il carico attuale gestito dalla piattaforma.
Le previsioni del carico di lavoro effettuate sulla Look-ahead window sono passate in ingresso a un rivelatore di cambio detto Proactive Change Detector: in questo modo è possibile rilevare se il workload subirà dei cambiamenti rilevanti nell'intervallo di tempo contenuto nella look-ahead window, e scatenare quindi una riconfigurazione proattiva.
Entrambi i Change Detector, reattivo e proattivo, sollevano degli eventi/allarmi, raccolti e gestiti da un Alarm Manager, il quale deciderà se sollecitare lo Adaptation Manager ad effettuare una ottimizzazione della piattaforma.

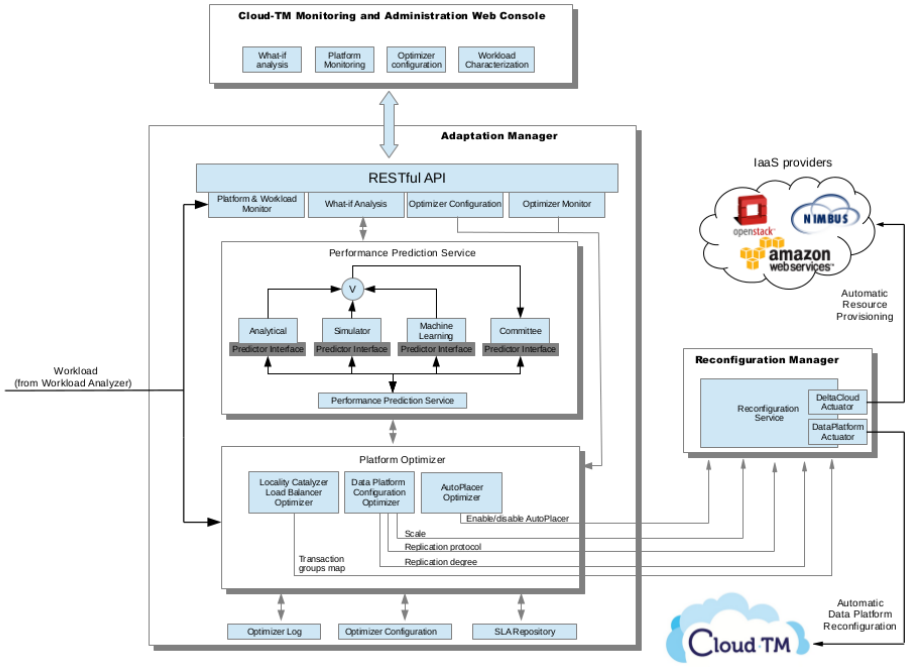


**Fig. 4.4: Architettura d'alto livello del componente Adaptation Manager.**

## 4.3 Adaptation Manager

Il componente Adaptation manager (AdM) è il componente chiave all'interno dello Autonomic Manager. Questo modulo si incarica di guidare il self-tuning di vari meccanismi della Data Platform di Cloud-TM e di automatizzare il processo di provisioning di risorse (acquisendo e rilasciandole in maniera trasparente da un provider IaaS) in base alle QoS.
Nella Figura 4.4 è mostrata l'architettura dello Adaptation Manager, evidenziandone i principali blocchi, e le interazioni con il resto della piattaforma Cloud-TM.
Possiamo distinguere due sotto-componenti principali:

- Performance predictor service;
- Platform optimizer.

Il Performance Prediction Service integra diversi meccanismi di previsione delle performance basato su differenti metodologie, le quali lavorano in sinergia per massimizzare l'accuratezza del sistema di predizione, e di conseguenza, dell'intero processo di self-tuning. In dettaglio, il Performance

Prediction Service combina sia tecniche white-box, adottando modelli analitici, sia black-box, attraverso l'uso di modelli machine learning, per cercare di raggiungere il meglio:

- alta accuratezza dei metodi statistici, tipici delle tecniche black-box, quando si ha a che fare con workload simili a quelli osservati nella fase di training;
- breve fasi di apprendimento di modelli white-box, e la loro potenza di estrapolazione, cioè la capacità di raggiungere alta accuratezza quando si ha a che fare con tipi di workload

Attraverso il Performance Prediction Service è possibile non solo guidare il processo di ottimizzazione, di cu si parlerà in seguito, ma permette all'utente finale di condurre analisi dettagliate *what-if*, utili per valutare quali sono le performance raggiungibili dalla piattaforma al variare della configurazione (scala, grado di replicazione, protocollo di replicazione) e/o del carico in ingresso. Tale strumento assume un importante valore sia per gli sviluppatori di applicazioni destinate ad essere eseguite nella piattaforma Cloud-TM, sia per i fornitori della piattaforma stessa (come ambiente PaaS). In questo modo, gli sviluppatori guadagnano forte conoscenza della scalabilità e efficienza della loro applicazione,
Dall'altra parte, i fornitori della piattaforma possono sfruttare l'analisi *what-if* per supportare il processo di gestione dei rischi alla base della specifica degli SLA con i clienti.
Il componente Platform Optimizer è incaricato di definire la strategia di riconfigurazione dei vari schemi di self-tuning integrati in Cloud-TM. Questo componente è stato disegnato affinchè sia flessibile, facilmente estendibile, attraverso la specifica di una catena di ottimizzatori responsabili dell'ottimizzazione dei singoli aspetti/parametri del Data Platform, ed altamente configurabile. Per raggiungere quest'ultima caratteristica, lo AdP espone delle API REST-based, largamente sfruttate dall'applicazione web *Monitoring and Administration Web Console* la quale permette all'utente finale di specificare quali meccanismi di self-tuning devono essere automatizzati dallo AdM. Inoltre, tramite la Console Web, è possibile effettuare e analizzare graficamente i risultati dell'analisi *what-if* attraverso un pannello web user-friendly.

Infine, l'eventuale riconfigurazione, formata da un insieme di adattazion decisa dal Platform Optimizer è passata in ingresso al *Reconfiguration Manager*, il quale:

- definisce l'ordine nel quale le varie riconfigurazioni saranno effettuate, tenendo in considerazione ogni possibile dipendenza che potrebbe riguardare efficienza/correttezza dell'intero processo di riconfigurazione;
- coordina un insieme di attuatori, i quali permettono di mettere in pratica le varie riconfigurazioni, incapsulando la complessità dell'interazione con differenti tipi di risorse/livelli della piattaforma.

Nel resto della sezione saranno descritti più in dettaglio i componenti finora introdotti.

### 4.3.1 Performance Prediction Service

Il *Performance Prediction Service* è un componente di particolare importanza all'interno dell'architettura dello AdP in quanto è suo compito guidare varie strategie di riconfigurazione e di supportare l'analisi *what-if*.
Come mostrato nel diagramma in Figura 4.4, il componente in questione dispone di diverse metodologie di predizione delle performace:

- metodi analitici;
- metodi machine learning;
- tecniche simulative.

Poichè queste tecniche sono del tutto complementari, il loro uso congiunto conferisce, allo AdP, grosse capacità di predizione. Come già detto, le tecniche machine learning, permettono di effetturare predizioni altamente affidabili per quelle configurazioni di sistema e profili di carico che rientrano nell'insieme di domini dello spazio dei paramentri già esplorati. L'esplorazione può essere portata a termine sia a a tempo di esecuzione, sia off-line, e quindi raffinata in fasi successive.
Quanto detto non vale per quell'insieme di domini, poichè le tecniche ML soffrono di povere capacità di estrapolazione, non precedentemente

osservati. In tale caso, le predizioni effettuate non possono considerarsi affidabili.
Attraverso l'uso di modelli simulativi e/o analitici, l'AM può tuttavia raggirare le limitazioni presentate dalle tecniche ML. I modelli analitici, rispetto quelli simulativi, richiedono meno potenza computazionale, ma richiedono sforzi maggiori durante la fase di disegno. Entrambi, poichè più complessi rispetto all'approccio ML, tendono ad introdurre un elevato numero di semplificazioni, le quali potrebbero degradare l'accuratezza della predizione.
Infine, la disponibilità di varie metodologie di predizione, ha permesso di implementare dei *meta-predittori*, i quali, in maniera del tutto trasparente, combinano le predizioni in uscita dai singoli predittori, e le ricombinano utilizzando uno schema di voting. In tale maniera, si cerca di compensare errori/debolezze degli individui predittori.

### 4.3.2 Platform Optimizer

Il *Platform Optimizer* (PO) è un componente altamente flessibile e configurabile, il quale orchestra l'esecuzione di singoli sotto-ottimizzatori, detti OptimizerComponent, responsabili di portare a termine l'ottimizzazione di specifici processi di self-tuning. Nel Listato 4.2 è riportata l'interfaccia comune a tutti gli OptimizerComponent.
Attualmente, lo AdM dispone di un *Data Platform Configuration Optimizer*, il quale si incarica di automatizzare il tuning della dimensione della piattaforma, il grado di replicazione dei dati e di scegliere il protocollo di replicazione più adatto.

**Data Platform Configuration Optimizer** La dimesione della piattaforma, il grado e il protocollo di replicazione, sono parametri che si intrecciano nella scelta della configurazione della piattaforma. Ad esempio, una piattaforma *full-replicated*, è spesso preferibile fintanto che il numero delle macchine è piccolo (4-8); vale il contrario invece in grosse piattaforme, dove un sistema parzialmente replicato è da preferire, fintanto che il workload in arrivo sia *read-intensive*. Per quanto riguarda il protocollo di replicazione, la scelta di un protocollo Primary-Backup in una grossa piattaforma non favorisce la scalabilità, in quanto tutte le transazioni di update saranno gestite da un singolo nodo. Differentemente, quando la piattaforma è piccola, e il profilo del

**Listing 4.1: Interfaccia dello Performance Prediction Service**

```
public interface OracleService {

    public PlatformConfiguration minimizeCosts(
  ProcessedSample sample,
                                                double
  arrivalRateToGuarantee,
                                                double
  abortRateToGuarantee,
                                                double
  responseTimeToGuarantee) throws OracleException;


    public PlatformConfiguration maximizeThroughput(
  ProcessedSample sample) throws OracleException;


    /**
     * What-if with Protocols on X-axis
     * @param sample
     * @param fixedNodes
     * @param fixedDegree
     * @return
     */
    public Map<PlatformConfiguration, OutputOracle> whatIf(
  ProcessedSample sample, int fixedNodes, int fixedDegree);


    /**
     * What-if with Nodes on X-axis
     * @param sample
     * @param fixedProtocol
     * @param fixedDegree
     * @return
     */
    public Map<PlatformConfiguration, OutputOracle> whatIf(
  ProcessedSample sample, int minNumNodes, int maxNumNodes,
  ReplicationProtocol fixedProtocol, int fixedDegree);


    /**
     * What-if with Degree on X-axis
     * @param sample
     * @param fixedNodes
     * @param fixedProtocol
     * @return
     */
    public Map<PlatformConfiguration, OutputOracle> whatIf(
  ProcessedSample sample, int minNumDegree, int maxNumDegree
  , int fixedNodes, ReplicationProtocol fixedProtocol);

}
```

**Listing 4.2: Interfaccia dello OptimizerComponent**

```
public interface OptimizerComponent<T>{

    public OptimizerType getType();

    public T doOptimize(ProcessedSample processedSample,
  boolean pureForecast);
}
```

workload è prevalentemente *write-intensive*, un protocollo Primary-Backup è più efficiente rispetto ad uno multi-master (2PC o TO **??**), il quale potrebbe incorrere in frequenti conflitti e generare elevato traffico di rete.
Per risolvere questo problema, il processo di self-tuning si affida su l'uscita dei vari predittori di performace.
Più in dettaglio l'ottimizzatore in questione determina i valori da assegnare a ciascun parametro, esplorando i vari stati possibili (scala x grado x protocollo), interrogando i predittori, attraverso il *Performance Prediction Service*, e selezionando la configurazione che massimizzi/minimizzi una funzione obiettivo:

- massimizzare i KPI usati per definire gli SLA
- minimizzare i costi operazionali della configurazione.

Attraverso un algoritmo di ricerca, l'ottimizzatore cerca di identificare la configurazione ottima che minimizzi i costi, assicurando che gli SLA definiti durante la fase di negoziazione siano rispettati.
Inoltre, l'ottimizzatore permette di esplorare esclusivamente un sottoinsieme dei tre parametri, rendendo i sistemi di self-tuning flessibili e permettendo all'amministratore della piattaforma di imporre dei vincoli solamente su alcuni parametri, assegnati staticamente.

**Ricerca esaustiva** Attualmente, l'ottimizzatore della configurazione della piattaforma dispone di due algoritmi di ricerca: un algoritmo esaustivo e uno basato su ricerca locale. Il primo, effettua una ricerca esaustiva in tutto lo spazio dei parametri per determinare la configurazione ottima, dato il workload corrente (e l'insieme di SLA da garantire). Questo algoritmo, oltre

**Listing 4.3: Pseudocodice per l'algoritmo di ricerca locale Hill Climbing**

```
Discrete Space Hill Climbing Algorithm
   currentNode = startNode;
   while do
      L = NEIGHBORS(currentNode);
      nextEval = -INF;
      nextNode = NULL;
      for each x in L
         if (OBJ_FUNC(x) > nextEval)
              nextNode = x;
              nextEval = OBJ_FUNC(x);
      if nextEval <= OBJ_FUNC(currentNode)
         return currentNode;
      currentNode = nextNode;
```

ad essere di facile implementazione, garantisce sempre l'identificazione della configurazione ottima.

**Ricerca locale: HillClimbing** Nel caso di piattaforme di grosse dimensioni (per numero di nodi), la crescita dello spazio dei parametri (la cui complessità asintotica cresce quadraticamente) impedirebbe l'uso di algoritmi di ricerca esaustivi. Di fatto, in tale caso, un ciclo di ottimizzazione richiederebbe lunghi tempi di ricerca, a discapito della reattività del sistema. Per superare queste problematiche, è stato implementato un algoritmo di ricerca locale *Hill Climbing*, attraverso il quale, a partire da uno stato iniziale, si esplorano gli stati adiacenti. Due stati si dicono adiacenti se e solo se è possibile passare da uno all'altro facendo variare solamente un parametro. Nel caso in cui la funzione obiettivo calcolata in uno stato adiacente presenti un valore superiore a quella dello stato iniziale, allora l'esplorazione continua, in maniera incrementale, esplorando gli stati adiacenti. Il Listato 4.3 presenta lo pseudocodice dell'algoritmo Hill Climbing implementato all'interno dell'ottimizzatore. Riducendo lo spazio degli stati esplorati si introduce un *trade-off* tra completezza ed efficienza dell'ottimizzatore.

### 4.3.3 Reconfiguration Manager

Il Reconfiguration Manager è il componente che si incarica di mettere in pratica le riconfigurazioni decise dallo AdM, attraverso il *Platform Optimizer*. Questo modulo compie i seguenti passi:

**Listing 4.4: Interfaccia dello Actuator**

```
public interface Actuator {

    public void stopInstance() throws ActuatorException;

    public void startInstance() throws ActuatorException;

    public List<String> runningInstances();

    public void switchProtocol(ReplicationProtocol
  repProtocol) throws ActuatorException;

    public void switchDegree(int degree) throws
  ActuatorException;

    public void triggerRebalancing(boolean enabled) throws
  ActuatorException;

}
```

- decide l'ordine nel quale le varie modifiche devono essere attuate;
- coordina l'esecuzione di tali riconfigurazioni attraverso un insieme di attuatori che permettono di acquisire/rilasciare risorse da differenti fornitori IaaS, e di interagire con vari livelli della piattaforma Cloud-TM che supportano la riconfigurazione dinamica.

Per quanto riguarda il primo punto, si deve notare che l'ordine nel quale certe modifiche alla piattaforma vengono attuate impatta la latenza della riconfigurazione stessa e la sua effettività. Ad esempio, è altamente desiderabile che riconfigurazioni che coinvolgono modifiche sia al grado di replicazione, sia alla scala della piattaforma, siano eseguite contemporaneamente. Se tali modifiche fossero apportate separatamente, i dati presenti all'interno della piattaforma sarebbero trasferiti più di una volta tra i nodi. Il *Reconfiguration Manager* incapsula la logica per applicare le modifiche richieste dallo *Adaptation Manager*, basandosi su un insieme di regole che determinano le precedenze di riconfigurazione. Le modifiche vengono applicate alla piattaforma attraverso degli attuatori, la cui specifica, sotto forma di interfaccia, è riportata nel Listato 4.4. Ogni attuatore deve implementare tale interfaccia.

| Azione | Medoto | URL |
|---|---|---|
| Recupera informazioni sullo stato della piattaforma | GET | http://ip:1515/status |
| Permette di cambiare il predittore corrente | POST | http://ip:1515/tuning/forecaster |
| Permette di cambiare la dimensione della piattaforma | POST | http://ip:1515/tuning/scale |
| Permette di cambiare il grado di replicazione di ogni dato | POST | http://ip:1515/tuning/degree |
| Permette di cambiare il protocollo di replicazione | POST | http://ip:1515/tuning/protocol |
| Recupera valori correnti di alcuni parametri di interesse per what-if analisi | GET | http://ip:1515/whatif/values |
| Effettua una interrogazione ai predittori di tipo what-if | POST | http://ip:1515/whatif |
| Effettua una interrogazione al predittore corrente per ricercare la configurazione ottima | GET | http://ip:1515/whatif/forecast |

**Tabella 4.1: Interfaccia REST esposta dallo AdP per il controllo e monitoring dei meccanismi di self-tuning**

Ogni attuatore, a sua volta, può utilizzare dei client per portare a termine i propri sotto-task. Lo *Adaptation Manager*, ad esempio, dispone di due tipi di attuatori: il primo, fa uso di un client Java per effettuare chiamate remote a delle API RESTful, esposte e definite dal progetto $\delta$-cloud. Attraverso le richieste REST, l'attuatore riesce a acquisire/rilasciare/gestire le risorse fisiche.
Il secondo attuatore, invece, fa uso di un client SSH, scritto in Java, attraverso il quale esegue degli script bash su macchine remote, già pronte per l'uso, attraverso i quali avvia i processi applicativi.
Entrambi gli attuatori disponibili all'interno dello AdP fanno uso di un client Java, capace di effettuare delle richieste sulla Data Platform e scatenare riconfigurazioni quali cambio di grado di replicazione o protocollo di replicazione. Le richieste verso la piattaforma vengono effettuate attraverso il protocollo, standardizzato, JMX (Java Management Extensions).

### 4.3.4 REST API e Web Console

È possibile configurare e gestire il componente Adaptation Manager effettuando delle richieste HTTP a web-service di tipo RESTful, esposti per mezzo di un server HTTP integrato, fornito all'interno del progetto *Grizzly NIO framework*. L'implementazione dei servizi REST è stata realizzata, includendo nello AdP il framework *Jersey RESTful Web Services*, il quale esporta un insieme di API standard per lo sviluppo di servizi JAX-RS.
La configurazione di default dello Autonomic Manager prevede che il server web si metta in ascolto sulla porta TCP *1515*. In Tabella 4.1 sono riportate le interfacce esposte per il controllo dello AdP.

**Monitoring and Administration Web Console** L'applicazione *Monitoring and Administration Web Console* permette all'utente finale il controllo di ogni singolo aspetto del componente Autonomic Manager del progetto Cloud-TM, in modo semplice ed immediato attraverso l'uso di un browser di ultima generazione.
Gli amministratori di sistema, attraverso un pannello di controllo, possono:

- monitorare lo stato corrente della piattaforma, visualizzando lo stream delle statistiche raccolte, in tempo reale, dal framework di monitoring WPM (Figura 4.5, pagina di overview e Figura 4.8, pagina di monitoring real-time);
- effettuare delle interrogazioni al *Prediction Service* per valutare la qualità della configurazione attuale e confrontarla con la configurazione ottima, suggerita dai predittori, per il carico corrente (Figura 4.7);
- abilitare/disabilitare vari meccanismi di self-tuning offerti dalla piattaforma (Figura 4.6).

L'applicazione web è stata sviluppata utilizzando tecnologie d'avanguardia, basate su linguaggio di scripting client-side, JavaScript, quale, ad esempio, il framework *jQuery*. Attraverso delle richieste AJAX, essa interagisce con lo AdP attraverso le interfacce precedentemente descritte; gli elementi presenti nelle varie pagine html sono quindi aggiornati in tempo reale.

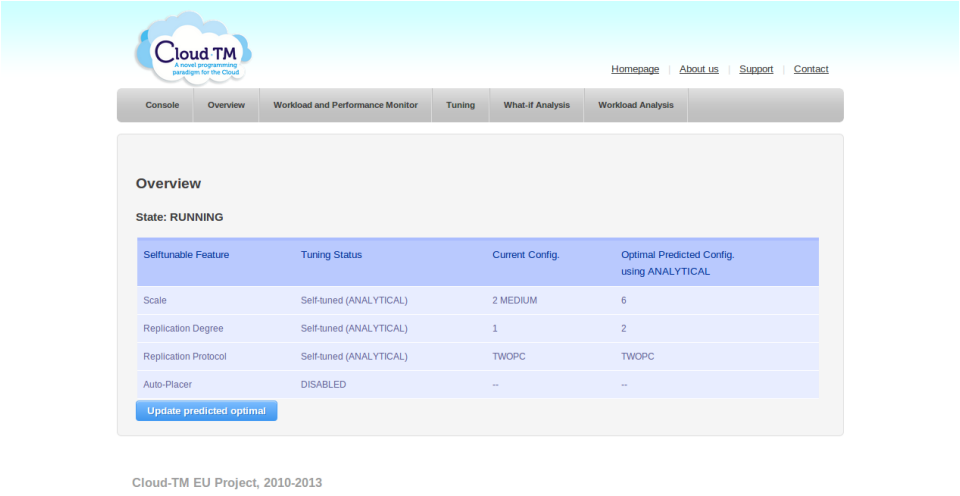


**Fig. 4.5: Monitoring and Administration Web Console: overview della configurazione corrente della piattaforma.**

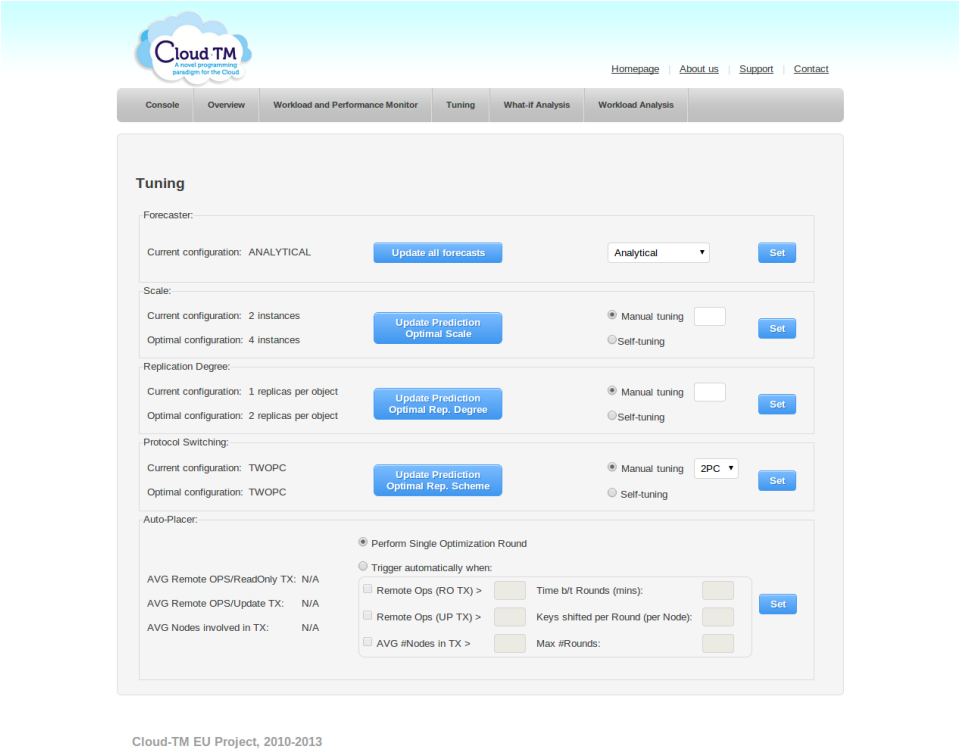


**Fig. 4.6: Monitoring and Administration Web Console: pannello per il tuning della piattaforma.**

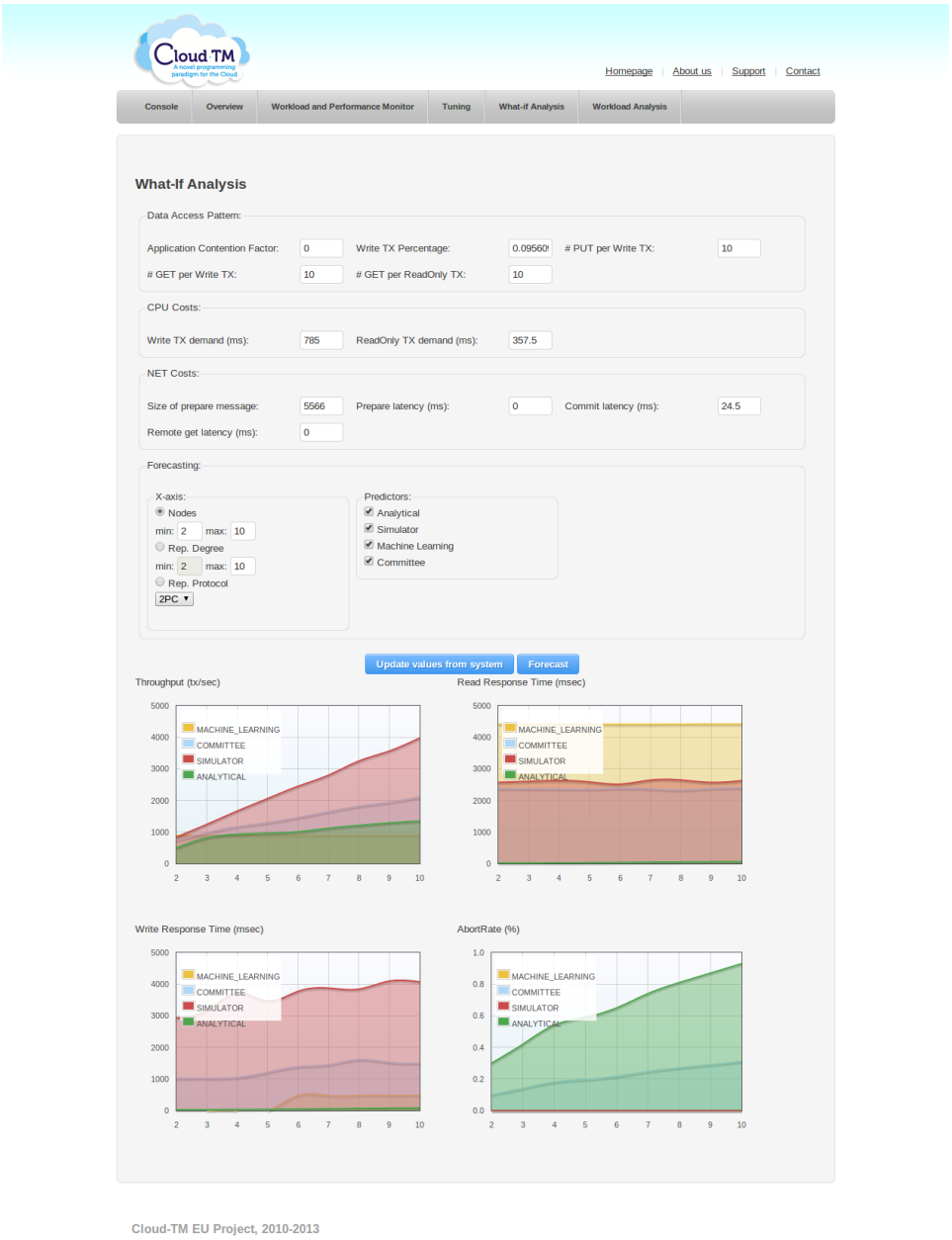


**Fig. 4.7: Monitoring and Administration Web Console: pannello per la What-if analisi.**

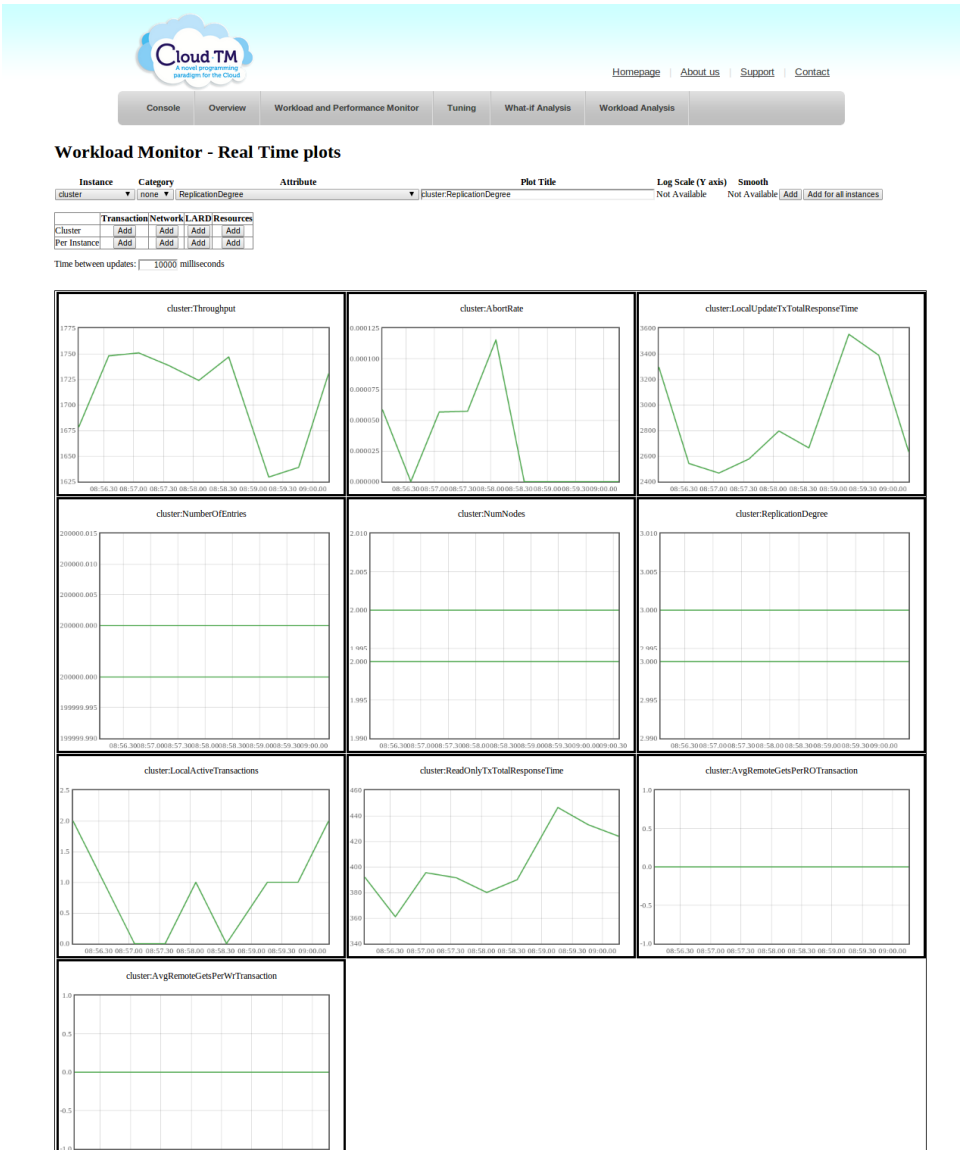


**Fig. 4.8: Monitoring and Administration Web Console: monitoring real-time della piattaforma.**

# CAPITOLO 5

# VALUTAZIONI

## 5.1 Elastic scaling e meccanismi di State Transfer

Scopo di questa sezione è di presentare i risultati ottenuti dall'esecuzione di esperimenti mirati a valutare efficacia ed efficienza di meccanismi utili per rispettare le QoS prestabilite con l'utente. In particolare, sarà valutato lo schema di *Non Blocking State Transfer* (NBST), progettato ed implementato all'interno della memoria transazionale del progetto Cloud-TM, ed un semplice predittore di performace *CPU-based*.

**Ambiente di testing** Gli esperimenti sono stati effettuati utilizzando le risorse offerte dalla piattaforma cloud FuturGrid. Il cluster utilizzato è composto da 20 server virtuali ognuno dei quali dispone di 1 CPU Intel(R) Xeon(R) X5570 @2.93GHz, 4 GB di Memoria RAM; le macchine virtuali sono interconnessi tramite Gigabit Ethernet. Su ogni singolo server è stata installata una immagine con sistema operativo CentOS 5.9.

### 5.1.1 Non Blocking State Transfer

Il NBST è considerato parte integrante del processo di *provisioning* quando attività quali cambio di scala della piattaforma e/o cambio di grado di replicazione hanno luogo. Poichè non è di interesse analizzare l'efficienza, espressa in termini di latenze, del provisioning di risorse computazionali

da parte del provider IaaS, i test sono stati effettuati in uno scenario dove le risorse sono già disponibile nella piattaforma, in modo da isolare la valutazione del NBST.
Una prima valutazione è stata effettuata utilizzando *Synthetic* benchmark con un data-set composto da 100 K chiavi, ciascuna di dimensione 1 KB. Il profilo del carico di lavoro generato è composto da due classi transazionali: la prima effettua 1000 accessi read-only sulle chiavi, la seconda, invece, effettua 10 accessi read-only e 10 accessi di update sulle chiavi.
Una seconda valutazione è stata effettuata utilizzando TPC-C benchmark, con un numero di warehouse pari a 3 e conseguente data-set composto da $\sim$1 M di chiavi.
Il framework di benchmarking è stato configurato per funzionare in modalità mula (Sezione 3.3.4) in modo tale da poter misurare il valore di throughput massimo per tale configurazione. Su ogni macchina è stato utilizzato un singolo thread, per generare ed eseguire transazioni, ed è stato impostato un think-time di ogni cliente pari a 0.
La configurazione iniziale della piattaforma per le valutazioni effettuate con il benchmark sintetico prevede 5 nodi in modalità full-replication (ogni nodo dispone di una propria copia dei dati in locale) e protocollo di replicazione Two-Phase Commit. L'evoluzione delle run di benchmark prevede che, dopo un dato istante di tempo ($\sim$3min), sia effettuata una richiesta allo Autonomic Manager per far variare la dimensione della piattaforma da 5 a 10 nodi. Dopo un ulteriore lasso di tempo, sufficientemente lungo per permettere al sistema sia di stabilizzarsi dopo la riconfigurazione, sia di raccogliere campioni relativi alle nuove performance, è effettuata una ulteriore richiesta di riconfigurazione della dimensione della piattaforma per scalare da 10 a 5 nodi. Adottando tale comportamento è possibile valutare, per ogni run, il comportamento della piattaforma al verificarsi di azioni sia di scale up, sia scale down.
Anche per la run effettuata con il benchmark TPC-C è stata adottato tale comportamento, a differenza però del numero di nodi iniziale/finale, pari a 2, e del numero di nodi su cui viene effettuata la riconfigurazione, pari a 5.
La decisione di adottare uno schema di replicazione full-replicated è voluta per poter mettere sotto stress i meccanismi propri del NBST, rendendo così ogni nodo presente nella piattaforma partecipe nell'esecuzione del processo o come donatore, o come accettore di dati.

Durante la valutazione portata a termine con il benchmark sintetico sono stati considerati due tipi di mix transazionali: il primo, *read-intensive*, presenta un numero di transazioni di tipo read-only pari al 90%; il secondo, *write-intensive*, è composto da un numero inferiore di transazioni di tipo read-only, pari al 50%.
Per quanto riguarda la valutazione portata a termine con il benchmark TPC-C è stato considerato solamente un mix transazionale, read-intensive, composto da un numero di transazioni di tipo StatusOrder (read-only) pari al 92% e in ugual numero, 4%, di transazioni Payment e NewOrder (update).
I risultati degli esperimenti eseguiti attraverso il benchmark sintetico sono riportati nei grafici in Figura 5.1 e Figura 5.2, rispettivamente per la run read-intensive e quella write-intesive, mentre nel grafico in Figura5.3 sono riportati i risultati per la run effettuata con TPC-C; ogni grafico presenta due insiemi di dati, rispettivamente il throughput della piattaforma e la scala della piattaforma.
Possiamo facilmente notare che in ogni run vi sono due intervalli in cui il NBST è attivo e il loro inizio coincide con una riconfigurazione della piattaforma da parte dello Autonomic Manager (tramite scale-up o scale-down). Nei grafici, si può notare che la durata dello state transfer che si verifica a seguito di una operazione di scale-up è maggiore rispetto alla durata dello stesso a fronte di una operazione di scale-down; tale aspetto è causato dalla scelta di configurare il sistema in modalità full-replicated e non comporta, durante lo scale-down, la ridistribuzione delle chiavi tra i vari nodi, in quanto ognuno già dispone di una copia locale. Inoltre, possiamo notare che la durata dello state transfer è indipendente dal tipo di profilo e mix transazionale.
Per quanto riguarda le performance della piattaforma transazionale, possiamo notare che, nel caso read-intensive, passando da un numero di nodi nella piattaforma pari a 5, ad un numero di nodi pari a 10, il numero delle transazioni che effettuano commit con successo aumenta di un fattore $\sim 1.5$; un tale incremento del throughput era atteso: secondo il modello di isolamento adottato all'interno della memoria transazionale di Cloud-TM, le transazioni read-only non creano conflitti e quindi non possono abortire. Inoltre, poichè il sistema è full-replicated, ogni server può gestire una richiesta senza dover contattare un nodo remoto. Lo stesso discorso è valido anche per quanto riguarda il test portato a termine attraverso il benchmark TPC-C.

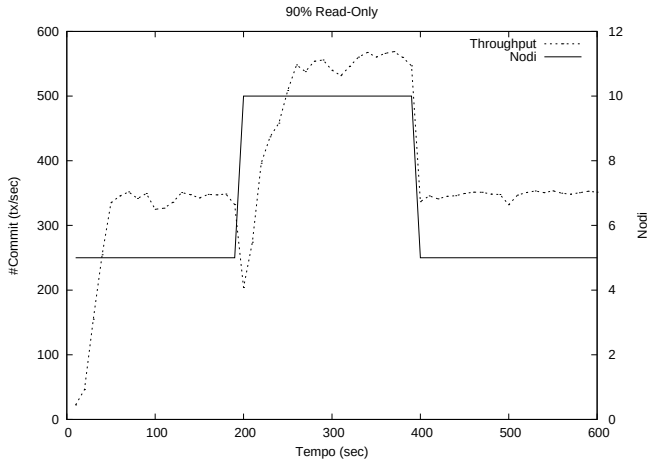


**Fig. 5.1: Effetti del NBST - Syntethic benchmark con workload read-intensive**

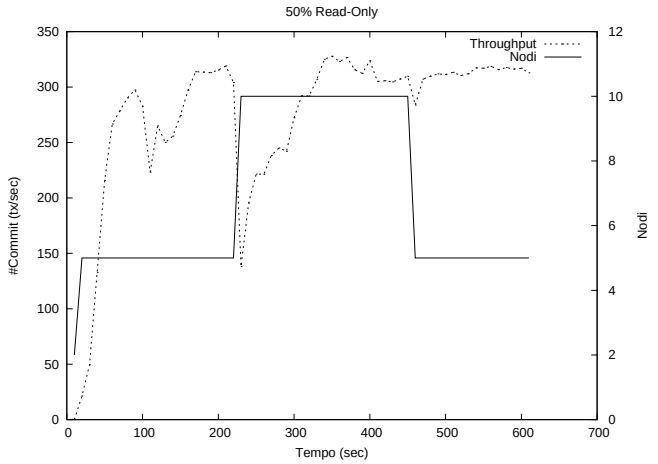


**Fig. 5.2: Effetti del NBST - Syntethic benchmark con workload write-intensive**

Per quanto riguarda invece l'esperimento write-intensive, il valore del throughput all'interno della piattaforma resta più o meno costante dopo la riconfigurazione della piattaforma; tale comportamento è dovuto poichè, all'aumentare del numero di nodi, vi è un aumento anche del grado di concorrenza sui dati da parte di transazioni di update. Poichè le transazioni di tipo update concorrenti creano conflitti, sono destinate ad effettuare un abort.

Tuttavia, tutti i risultati degli esperimenti effettuati presentano un notevole calo delle performance negli intervalli di tempo durante i quali ha luogo un processo di state transfer.

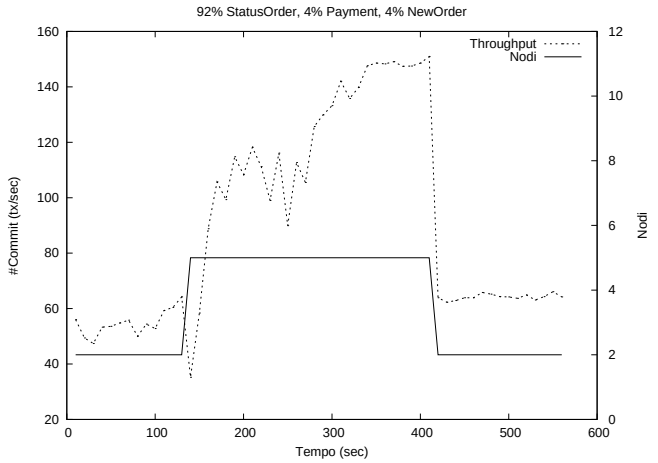


**Fig. 5.3: Effetti del NBST - TPC-C benchmark con workload read-intensive**

### 5.1.2 Ottimizzatore CPU-based

Molti fornitori di servizi cloud offrono la possibilità di effettuare scaling dinamico andando a definire delle soglie sul valore di CPU. Allo stesso modo, lo Autonomic Manager dispone dell'implementazione di un ottimizzatore della piattaforma capace di basare le sue decisioni sul carico attuale della CPU. L'ottimizzatore in questione è stato successivamente utilizzato per pilotare lo elastic scaling della piattaforma transazionale, i cui risultati sono presentati nel resto della sezione.
Durante tutti gli esperimenti portati a termine, le soglie max e min per la CPU sono state impostate rispettivamente a 80% e 20%, relative ad un azione di scale-up e scale-down.

**Sistema aperto e workload read-intensive** Una prima valutazione è stata portata a termine configurando ELASTIC DTM BENCH per simulare un sistema di code aperto la cui curva dello arrival rate fosse una ramp-up. Durante la valutazione lo arrival rate è stato fatto variare da 0 a 1000 tx/s in cinque step, incrementandolo di volta in volta di 225 tx/s. Affinchè il sistema avesse il tempo per stabilizzarsi, evitando così il thrash, il framework di benchmarking è stato configurato con un intervallo di campionamento elevato, cioè il periodo di tempo da trascorrere prima del verificarsi di un nuovo cambio di arrival rate, pari a 20 minuti; inoltre, ogni riconfigurazione è seguita da un periodo di tempo, detto *cool down*, pari a 3 minuti, durante il

quale lo Autonomic Manager non può intraprendere nuove riconfigurazioni; ogni azione di riconfigurazione intrapresa dallo Autonomic Manager è capace di far variare la dimensione della piattaforma di $\pm$ 2 nodi.
Il profilo del workload generato dal framework di benchmarking durante la valutazione è composto da due classi transazionali: la prima effettua 1000 accessi read-only sulle chiavi, la seconda, invece, effettua 10 accessi read-only e 10 accessi di update sulle chiavi. Il mix transazionale considerato si compone del 99% delle transazioni di tipo read-only, mentre, solo 1% delle transazioni sono di tipo update. La memoria transazionale è stata configurata per funzionare in modalità full-replicated e l'esperimento è stato portato a termine adottando il protocollo di replicazione *two-phase commit*.
Il risultato dell'esperimento è presentato in Figura 5.4; al suo interno sono presentati tre grafici relativi rispettivamente a throughput, numero di nodi e carico della CPU.
Possiamo notare che ogni 1200 secondi ha luogo un incremento del throughput e del carico della CPU, in risposta alla variazione dello arrival rate. L'incremento del carico della CPU è dovuto alla sinergia tra la natura del profilo di carico generato dal framework di benchmarking, prevalentemente read-intensive, e la modalità di funzionamento full replicated della piattaforma.
Il tempo d'esecuzione di una transazione si compone del tempo di CPU necessario per processare la richiesta e del tempo di rete necessario per contattare nodi remoti, o per accedere a dati non presenti localmente, attraverso operazioni get, o per chiedere il commit di una transazione di update. Durante l'esperimento, le transazioni read-only, prevalenti sull quelle di update, presentano tempi di rete nulli poichè tutti i dati sono disponibili localmente e quindi non effettuano richieste remote; cioè implica che la maggior parte del tempo d'esecuzione di una transazione è speso all'interno della CPU, la quale, come ben visibile in figura, raggiunge livelli elevati.
L'intervento dello Autonomic Manager è visibile osservando l'evoluzione della dimensione della piattaforma, raffigurata nel grafico centrale della Figura 5.4. Possiamo vedere come, attraverso l'aggiunta a runtime di nuovi nodi, il controllore cerca di ristabilire il carico medio della CPU della piattaforma al di sotto della soglia predefinita. Nella maggior parte dei casi, una sola riconfigurazione non è sufficiente per raggiungere tale scopo; in tali casi, il periodo di cool down è seguito da una nuova riconfigurazione.

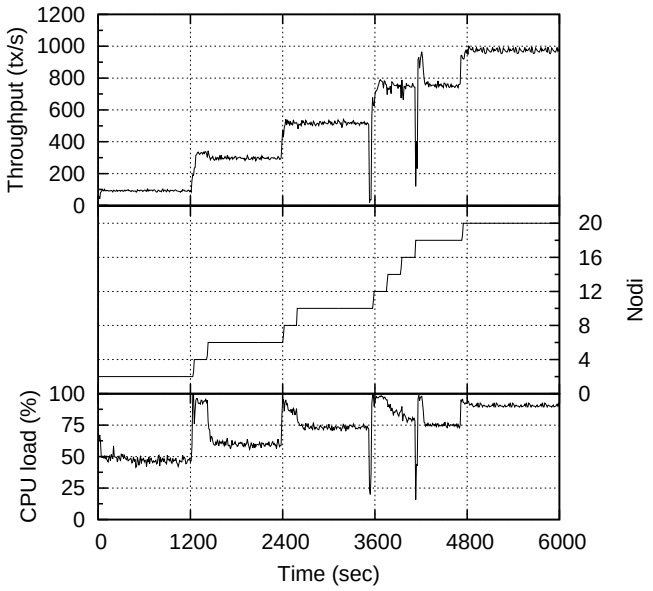


**Fig. 5.4: Evoluzione del sistema con arrival rate di tipo *ramp-up* (granularità: 1200 sec.; cooldown: 180 sec.) con profilo di carico read-intensive (99% read-only tx, 1000 op. read; 1% update tx, 10 op. read, 10 op. write)**

Nell'istante di tempo pari a 4200 sec possiamo notare che un primo calo del throughput è seguito da un immediato incremento ben al di sopra del valore dello arrival rate; tale incremento è dovuto ad un maggior accodamento delle transazioni nelle code dei singoli nodi avvenuto negli istanti precedente.
Possiamo concludere la valutazione affermando che, un siffatto ottimizzatore, basato sul livello della CPU, applicato su una piattaforma transazionale, si comporta egregiamente quando in ingresso al sistema vi è un profilo di workload read-intensive.

**Sistema mulo e workload write-intensive** Una seconda valutazione, simile a quella descritta nel paragrafo precendete, è stata portata a termine configurando il framework di benchmarking per simulare un sistema di code chiuso, al cui interno, il numero di clienti è pari al numero dei server del sistema (sistema mulo). All'interno dell'esperimento, ogni riconfigurazione della piattaforma è seguita da un periodo di tempo, detto *cool down*, pari a 2 minuti, durante il quale lo Autonomic Manager non può intraprendere

nuove riconfigurazioni; ogni azione di riconfigurazione intrapresa dallo Autonomic Manager è capace di far variare la dimensione della piattaforma di $\pm$ 2 nodo.
Il profilo del workload generato dal framework di benchmarking durante la valutazione è composto da due classi transazionali: la prima effettua 1000 accessi read-only sulle chiavi, la seconda, invece, effettua 5 accessi read-only e 2 accessi di update sulle chiavi, generando media contesa tra i dati. Il mix transazionale considerato è composto per il 50% da transazioni di tipo read-only, e il restante 50% di tipo update. La memoria transazionale è stata configurata per funzionare in modalità full-replicated e l'esperimento è stato portato a termine adottando il protocollo di replicazione *two-phase commit*.
Il risultato dell'esperimento è presentato in Figura 5.5; al suo interno sono presentati tre grafici relativi rispettivamente a throughput, numero di nodi e carico della CPU.
Possiamo notare che ogni 120 secondi, al termine dell'intervallo di tempo dedicato al *cooldown*, ha luogo una riconfigurazione della piattaforma, intrapresa dallo Autonomic Manager attraverso l'interrogazione dell'ottimizzatore CPU-based. Almeno per quanto riguarda le prime tre riconfigurazioni, l'aggiunta di nuovi nodi all'interno della piattaforma permette di far crescere il numero di transazioni correttamente gestite, il cui valore raggiunge in media le 300/350 tx/s. Tuttavia, quando la dimensione della piattaforma raggiunge 10 nodi, il trend di crescita del throughput si ferma e notiamo che il suo valore decresce; osservando il grafico posto in basso possiamo invece notare che la percentuale dell'uso della CPU subisce cresce ed è vicino a saturare la risorsa.
Possiamo concludere che il workload utilizzato per portare a termine questo esperimento non è in grado di scalare, all'aumentare del numero di nodi. L'aumento della CPU è dovuto sia all'esecuzione di transazioni di tipo read-only, le quali accedono a dati presenti nella memoria locale, sia al numero, sempre maggiore, di richieste di commit provenienti dai nodi responsabili dell'esecuzione di transazioni di tipo update. Inoltre, sebbene la valutazione portata a termine nel paragrafo precedente abbia presentato dei buoni risultati, possiamo affermare che l'utilizzo di un predittore di performance il quale basa le sue predizioni esclusivamente sull'osservazione del carico della CPU, non è adatto per controllare la dimensione, e quindi gestire lo elastic-scaling, di piattaforme transazionali.

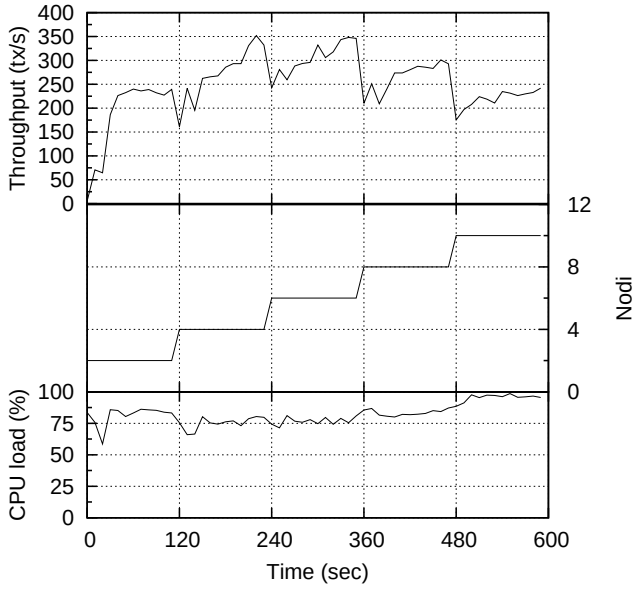


**Fig. 5.5: Evoluzione del sistema (mulo (granularità: 1200 sec.; cooldown: 180 sec.) con profilo di carico write-intensive (50% read-only tx, 1000 op. read; 1% update tx, 5 op. read, 2 op. write)**

# CAPITOLO 6

# CONCLUSIONI

Il lavoro è stato sviluppato all'interno del progetto europeo Cloud-TM e ha cercato di risolvere problematiche legate al ridimensionamento di piattaforme transazionali distribuite e alla configurazione autonomica dei loro schemi di replicazione, rispettivamente noti in letteratura come elastic scaling e self-tuning.
Dopo aver analizzato e classificato le varie memorie transazionali rappresentanti lo stato dell'arte, sono stati sviluppati strumenti strettamente correlati: il componente Autonomic Manager, presente all'interno dell'architettura della piattaforma Cloud-TM, e ELASTIC DTM BENCH, un framework di benchmarking per piattaforme transazionali distribuite elastiche; quest'ultimo è stato largamente utilizzato durante il lavoro per condurre esperimenti sullo Autonomic Manager.
Come risulta dalle varie attività di ricerca svolte negli ultimi anni, le prestazioni delle memorie transazionali distribuite, sono fortemente dipendenti dal volume e dal profilo del carico e indipendenti dai dettagli implementativi della specifica piattaforma; non esiste, quindi, una configurazione capace di assicurare prestazioni ottime in un vasto range di workload. Per tale motivo, il componente Autonomic Manager è stato disegnato ed implementato per garantire specifici Service Level Agreement, stabiliti con l'utente, intervenendo sia nella gestione delle risorse computazionali attraverso l'interazione con fornitori di servizi cloud IaaS, variando la dimensione della piattaforma,

sia nella gestione degli schemi di replicazione implementati nella memoria transazionale, variando grado e protocollo di replicazione. Tali aspetti, strettamenti relazionati con le prestazioni, permettono alla piattaforma di auto-adattarsi al carico di lavoro corrente e raggiungere prestazioni ottimali. L'architettura altamente modulare, presenta chiare interfacce, le quali rendono estremamente facile l'estensione e il riuso dei moduli presenti all'interno del componente; è possibile sviluppare ed integrare dei predittori di performance innovativi.
Lo Autonomic Manager dispone di modalità di funzionamento automatica, rendendo la piattaforma totalmente autonoma, semi-automatica, lasciando all'utente la possibilità di fissare alcuni aspetti della piattaforma e totalmente manuale, in cui l'utente si fa carico in toto della configurazione della piattaforma; il servizio what-if-analysis, infatti, è di supporto per l'utente e permette di effettuare una serie di predizioni, off-line, sulle prestazioni della piattaforma al variare di parametri significativi del workload.
Per facilitare sia il controllo della piattaforma Cloud-TM, sia l'interazione con i servizi di monitoring e analisi delle prestazioni, è stata sviluppata una applicazione web, la quale interagisce con lo Autonomic Manager attraverso chiamate remote ad una interfaccia esposta attraverso servizi REST.
ELASTIC DTM BENCH, sviluppato estendendo il noto progetto open-source Radargun, si pone come utile strumento per il benchmarking di piattaforme transazionali distribuite elastiche. La possibilità di poter portare a termine valutazioni elastiche rende ELASTIC DTM BENCH unico nel suo genere. Inoltre, sono stati aggiunti strumenti di supporto per la simulazione di sistemi a popolazione finita e infinita, oltre alla possibilità di generare forme d'onda e profili di carico personalizzabili dall'utente per analizzare il comportamento del sistema sotto particolari condizioni di lavoro.
ELASTIC DTM BENCH dispone di un insieme di applicazioni di benchmarking, capaci sia di simulare profili transazionali appartenenti a scenari reali, sia sintetici. Inoltre, sono state predisposte interfacce e classi astratte per favorire il riuso dei componenti già implementati e l'estensione del framework da parte della comunità; per tale motivo, è stato deciso di rilasciare il progetto sotto licenza open-source, disponibile quindi a tutta la comunità, sperando che rappresenti un tool di riferimento per i ricercatori attivi nell'area dello *elasting scaling*, area ancora emergente e stimolante.
All'interno del lavoro, ELASTIC DTM BENCH è stato utilizzato per validare

sia il corretto funzionamento dello Autonomic Manager, sia schemi innovativi, sviluppati durante il progetto Cloud-TM ed integrati all'interno di Infinispan: sono state valutate le prestazioni del Non-Blocking State Transfer, meccanismo capace di garantire l'esecuzione delle transazioni anche durante le fasi di riconfigurazione della dimensione della piattaforma. Tale caratteristica è di fondamentale importanza negli ambienti cloud, in quanto permette di minimizzare l'impatto delle riconfigurazioni sia sulle performance, sia sulla disponibilità del servizio.

# BIBLIOGRAFIA


[1] Mohammad Ansari, Christos Kotselidis, Kim Jarvis, Mikel Luján, Chris Kirkham, and Ian Watson. Lee-tm: A non-trivial benchmark for transactional memory. In *ICA3PP '08: Proceedings of the 7th International Conference on Algorithms and Architectures for Parallel Processing*. LNCS, Springer, June 2008.

[2] Hagit Attiya, Vincent Gramoli, and Alessia Milani. Brief announcement: combine – an improved directory-based consistency protocol. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, SPAA '10, pages 72–73, New York, NY, USA, 2010. ACM.

[3] Robert L. Bocchino, Vikram S. Adve, and Bradford L. Chamberlain. Software transactional memory for large scale clusters. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 247–258, New York, NY, USA, 2008. ACM.

[4] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[5] Maria Couceiro, Paolo Romano, and Luis Rodrigues. Polycert: polymorphic self-optimizing replication for in-memory transactional grids. In *Proceedings of the 12th ACM/IFIP/USENIX international conference on*

*Middleware*, Middleware'11, pages 309–328, Berlin, Heidelberg, 2011. Springer-Verlag.

[6] Maria Couceiro, Pedro Ruivo, Paolo Romano, and Luis Rodrigues. Chasing the optimum in replicated in-memory transactional platforms via protocol adaptation. *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 0:1–12, 2013.

[7] Diego Didona, Paolo Romano, Sebastiano Peluso, and Francesco Quaglia. Transactional auto scaler: elastic scaling of in-memory transactional data grids. In *Proceedings of the 9th international conference on Autonomic computing*, ICAC '12, pages 125–134, New York, NY, USA, 2012. ACM.

[8] Pascal Felber, Christof Fetzer, Patrick Marlier, and Torvald Riegel. Time-based software transactional memory. *IEEE Trans. Parallel Distrib. Syst.*, 21(12):1793–1807, December 2010.

[9] Lattice Framework. `http://clayfour.ee.ucl.ac.uk/lattice/`.

[10] Rachid Guerraoui, Michal Kapalka, and Jan Vitek. Stmbench7: a benchmark for software transactional memory. *SIGOPS Oper. Syst. Rev.*, 41(3):315–324, March 2007.

[11] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory, 2nd Edition*. Morgan and Claypool Publishers, 2nd edition, 2010.

[12] Maurice Herlihy and Ye Sun. Distributed transactional memory for metric-space networks. In *Proceedings of the 19th international conference on Distributed Computing*, DISC'05, pages 324–338, Berlin, Heidelberg, 2005. Springer-Verlag.

[13] Bettina Kemme, Fernando Pedone, Gustavo Alonso, Andre Schiper, and Matthias Wiesmann. Using optimistic atomic broadcast in transaction processing systems. *IEEE Trans. on Knowl. and Data Eng.*, 15(4):1018–1032, July 2003.

[14] Roberto Palmieri, Francesco Quaglia, and Paolo Romano. Aggro: Boosting stm replication via aggressively optimistic transaction processing.

In *Proceedings of the 2010 Ninth IEEE International Symposium on Network Computing and Applications*, NCA '10, pages 20–27, Washington, DC, USA, 2010. IEEE Computer Society.

[15] Diego Rughetti, Pierangelo Di Sanzo, Bruno Ciciani, and Francesco Quaglia. Machine learning-based self-adjusting concurrency in software transactional memory systems. In *Proceedings of the 2012 IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, MASCOTS '12, pages 278–285, Washington, DC, USA, 2012. IEEE Computer Society.

[16] Pedro Ruivo, Maria Couceiro, Paolo Romano, and Luis Rodrigues. Exploiting total order multicast in weakly consistent transactional caches. In *Proceedings of the 2011 IEEE 17th Pacific Rim International Symposium on Dependable Computing*, PRDC '11, pages 99–108, Washington, DC, USA, 2011. IEEE Computer Society.

[17] Mohamed M. Saad and Binoy Ravindran. Hyflow: a high performance distributed software transactional memory framework. In *Proceedings of the 20th international symposium on High performance distributed computing*, HPDC '11, pages 265–266, New York, NY, USA, 2011. ACM.

[18] Mohamed M. Saad and Binoy Ravindran. Snake: control flow distributed software transactional memory. In *Proceedings of the 13th international conference on Stabilization, safety, and security of distributed systems*, SSS'11, pages 238–252, Berlin, Heidelberg, 2011. Springer-Verlag.

[19] Kim Shanley and Amie Belongia. Tpc releases new benchmark: Tpc-c. *SIGMETRICS Perform. Eval. Rev.*, 20(2):8–22, November 1992.

[20] Bo Zhang and Binoy Ravindran. Brief announcement: Relay: A cache-coherence protocol for distributed transactional memory. In *Proceedings of the 13th International Conference on Principles of Distributed Systems*, OPODIS '09, pages 48–53, Berlin, Heidelberg, 2009. Springer-Verlag.

# APPENDICE A

# MANUALE D'USO DI ELASTIC DTM BENCH

Lo scopo di questa Appendice è quello di fornire supporto all'installazione, configurazione ed uso del software di benchmarking sviluppato all'interno del lavoro di tesi; per ogni attività enunciata è, di fatto, dedicata una sotto sezione.

## A.1 Installazione

Lo sviluppo del software è stato tenuto sotto controllo di versione attraverso il sistema *Git*. I sorgenti del software sono pubblicamente disponibili, nella piattaforma online *GitHub* la quale permette di effettuare il clone del progetto all'interno del proprio repository GitHub; in alternativa è possibile clonare il progetto direttamente sulla propria macchina locale, digitando sul terminale:

```
git clone https://github.com/xsurfer/ScalableRadargun.git
```

Una directory *ScalableRadargun*, dovrebbe essere stata creata nella directory corrente e riempita con i sorgenti, script e file di configurazione. È possibile apportare modifiche al codice sorgente, oppure procedere direttamente alla compilazione. Grazie all'uso del tool *Maven* per la gestione delle dipendenze,

il processo di compilazione richieste banalmente l'esecuzione del seguente comando:

```
mvn clean install
```

In questo modo, *Maven* provvederà a soddisfare le dipendenze, scaricandole se necessario da repository remoto, e compilando i sorgenti.
A fine operazione verrà generato, all'interno della cartella *target/distribution*, sia un archivio *.zip*, sia una cartella *RadarGun-1.1.0-SNAPSHOT*, all'interno della quale è estratto il contenuto dell'archivio.

## A.2 Configurazione

È possibile personalizzare l'esecuzione del test di benchmark modificando il file *benchmark.xml*, presente nella directory *target/distribution/RadarGun-1.1.0-SNAPSHOT/conf*.
Tale file è suddiviso in sezioni. All'interno del tag *master* è possibile specificare, l'host/ip del nodo master e la porta sulla quale è in ascolto (default: 2103). Attraverso gli attributi del tag *benchmark*, è possibile specificare il numero minimo di nodi da attendere prima di dar inizio al test. È inoltre previsto uno scenario in cui il test è eseguito più volte, facendo variare, automaticamente, il numero degli slave, il quale sarà incrementato di un valore specificato. All'interno del tag *bechmark* è possibile specificare l'insieme di stage che compongono il benchmark.
All'interno del tag *products* è possibile definire il prodotto sul quale si vuole svolgere il test di benchmark. Infine, all'interno della sezione *reports*, è possibile specificare stage capace di generare dei report, quali, ad esempio, grafici.

### A.2.1 Benchmark Applicativi

Come descritto nel Capitolo 3, il framework di benchmarking dispone di tre applicazioni di benchmark, TPC-C, Vacation e SyntheticBenchmark, la cui logica è implementata rispettivamente all'interno dei moduli *TpccBenchmarkStage*, *VacationBenchmarkStage* e *SyntheticBenchmarkStage*. Tutti e tre i benchmark, dispongono di uno specifico stage di *warm-up* o di popolamento, che permettono di inizializzare la piattaforma transazionale.

**TPC-C** A seguire è presentato un esempio di configurazione per lo stage di popolamento e per lo stage di benchmark dell'applicazione TPC-C. Sono forniti commenti sul significato di ogni attributo.

```
<TpccPopulation
  <!-- id univoco per stage -->
  id="d"
  <!-- numero di warehouse da popolare -->
  numWarehouses="1"
  <!-- bitmask per la generazione dei cognomi dei clienti -->
  cLastMask="0"
  <!-- bitmask per la generazione dei numeri dei clienti -->
  olIdMask="0"
  <!-- bitmask per la generazione dei numeri degli ordini -->
  cIdMask="0"
  <!-- abilita il popolamento parallelo su più thread -->
  threadParallelLoad="true"
  <!-- numero di operazioni all'interno di ogni transazione -->
  batchLevel="200"
  <!-- numero di thread da utilizzare -->
  numLoaderThreads="4"
  <!-- effettua il popolamento una sola volta -->
  oneWarmup="true"
  <!-- permette di non eseguire lo stage sui nuovi nodi -->
  skippable="true" />

<TpccBenchmark
  <!-- id univoco per stage -->
  id="h"
  <!-- numero di warehouse da popolare -->
  statsSamplingInterval="30000"
  <!-- numero di Consumer che elaboreranno transazioni -->
  numOfThreads="2"
  <!-- tempo di esecuzione dello stage -->
  simulationTimeSec="6000"
  <!-- numero di warehouse da popolare -->
  accessSameWarehouse="false"
  <!-- percentuale di transazioni di tipo Payment -->
  paymentWeight="43"
  <!-- percentuale di transazioni di tipo orderStatus -->
  orderStatusWeight="53"
  <!-- politica da adottare in caso di abort -->
  retryOnAbort="RETRY_SAME_CLASS"
  <!-- tempo di attesa prima di retry -->
```

```
  backOffTime="0"
  <!-- permette di non eseguire lo stage sui nuovi nodi -->
  skippable="true"
  <!-- permette a nuovi nodi di effettuare il join -->
  scalable="true" >
```

**Vacation** A seguire è presentato un esempio di configurazione per lo stage di popolamento e per lo stage di benchmark dell'applicazione Vacation. Sono forniti commenti sul significato di ogni attributo.

```
<VacationPopulation
  <!-- id univoco per stage -->
  id="d"
  <!-- dimensione delle tabelle -->
  relations="20"
  <!-- permette di non eseguire lo stage sui nuovi nodi -->
  skippable="true"/>

<VacationBenchmark
  <!-- id univoco per stage -->
  id="h"
  clients="1"
  numOfThreads="1"
  number="4"
  relations="1024"
  simulationTimeSec="60"
  readOnly="50"
  user="98"
  skippable="true"
  scalable="true">
```

**Synthetic Benchmark** A seguire è presentato un esempio di configurazione per lo stage di popolamento e per lo stage di benchmark dell'applicazione Vacation. Sono forniti commenti sul significato di ogni attributo.

```
<SyntheticWarmup
  <!-- id univoco per stage -->
  id="d"
  keyGeneratorClass="org.radargun.stages.stressors.ContentionStringKeyGenerator"
  numberOfAttributes="100000"
  sizeOfAnAttribute="1000"
  numOfThreads="2"
```

```
  transactionSize="100"
  skippable="true"/>

<SyntheticBenchmark
  <!-- id univoco per stage -->
  id="h"
  <!-- tempo di esecuzione dello stage -->
  simulationTimeSec="18000"
  <!-- dimensione di ogni oggetto presente nella cache -->
  sizeOfAnAttribute="1000"
  <!-- classe da utilizzare per generare le chiavi -->
  keyGeneratorClass="org.radargun.stages.stressors.ContentionStringKeyGenerator"
  <!-- numero di chiavi presenti nella cache -->
  numberOfAttributes="100000"
  <!-- numero di Consumer che elaboreranno transazioni -->
  numOfThreads="2"
  <!-- numero di op write in transazione di tipo update -->
  updateXactWrites="10"
  <!-- numero di op read in transazione di tipo update -->
  updateXactReads="10"
  <!-- numero di op read in transazione di tipo read -->
  readOnlyXactSize="10"
  <!-- percentuale di transazioni di tipo update -->
  writePercentage="10"
  <!-- politica da adottare in caso di abort -->
  retryOnAbort="RETRY_SAME_CLASS"
  statsSamplingInterval="15000"
  <!-- permette di non eseguire lo stage sui nuovi nodi -->
  skippable="true"
  <!-- permette a nuovi nodi di effettuare il join -->
  scalable="true">
```

### A.2.2 Generatore di carico

Quando si specifica uno stage di benchmark è possibile descrivere, al suo interno, il tipo di sistema che si vuole simulare, aperto o chiuso, e, nel caso di sistema aperto, è possibile specificare quale generatore di workload si vuole usare.

#### Sistema Aperto e generatori di carico

```
<OpenSystem>
  <Ramp
    rateDistribution="exponential"
```

```
    granularity="5000"
    initTime="0"
    slope="1"
    yintercept="0"
    maxArrivalRate="100" /-->

  <!--Trace
    rateDistribution="exponential"
    granularity="1"
    initTime="0"
    file="trace.log"
    maxArrivalRate="3000" /-->

  <!--Sin
    rateDistribution="exponential"
    granularity="500"
    initTime="0"
    amplitude="1000"
    maxArrivalRate="3000" /-->
</OpenSystem>
```

**Sistema Chiuso**

```
<ClosedSystem thinkTime="2000" population="10" rateDistribution="exponential" />
```

**Sistema Mulo**

```
<MuleSystem thinkTime="0" rateDistribution="exponential" />
```

## A.3 Uso

Una volta che l'utente ha completato la fase di configurazione, è possibile eseguire il test di benchmark, avviando prima di tutto il nodo master. Posizionandosi nella home directory del framework eseguire, tramite terminale, il seguente comando:

```
./bin/master.sh
```

Sarà avviato il processo master e, all'interno del file *stdout_master.out* è possibile tenere sotto controllo i messaggi di log stampati dal processo. Una volta che il master è pronto per ricevere connessioni da parte di nodi slave, possiamo avviare quest'ultimi eseguendo:

```
./bin/slave.sh -m <masterHost:ip>
```

Anche i processi slave generano, all'interno della stessa directory dove sono stati eseguiti, un file di log che raccoglie tutte le stampe.

# APPENDICE B

# MANUALE D'USO DELLO AUTONOMIC MANAGER

Lo scopo di questa Appendice è quello di fornire supporto all'installazione, configurazione ed uso del componente Autonomic Manager sviluppato all'interno del lavoro di tesi,; ad ogni attività enunciata è dedicata una sotto sezione.

## B.1 Installazione

Il codice sorgente del componente Autonomic Manager è stato rilasciato sotto licensa *open-source* ed è pubblicamente disponibile sulla piattaforma *GitHub*. È possibile clonare il progetto all'interno del proprio repository GitHub, o, alternativamente, clonarlo direttamente sulla propria macchina, eseguendo, sul terminale, il seguente comando:

```
$ git clone https://github.com/xsurfer/AdaptationManager.git
```

La directory *AdaptationManager* dovrebbe essere stata creata e riempita con i sorgenti, script e file di configurazione presenti nel repository remoto. È possibile apportare modifiche al codice sorgente, oppure procedere direttamente alla compilazione. Grazie all'uso del tool *Maven* per la gestione delle dipendenze, è possibile compilare i sorgenti eseguendo:

```
$ mvn clean install
```

In questo modo, *Maven* provvederà a soddisfare le dipendenze, scaricandole se necessario dal repository remoto, e a compilare i sorgenti.
A fine operazione è generato, all'interno della cartella *target*, un archivio *.tar.gz* e una cartella *Adaptation Manager*, all'interno della quale è estratto il contenuto dell'archivio.
A supporto dello *Adaptation Manager*, sono stati creati un insieme di script raccolti in un repository pubblico su *GitHub*. L'uso degli script richiede che le risorse sia già disponibili. È possibile clonare il progetto contenente gli script eseguendo:

```
$ cd ~
$ git clone https://github.com/xsurfer/AutonomicManagerScripts.git
```

L'operazione di clone genererà la directory *AutonomicManagerScripts* all'interno della home dell'utente, la cui struttura è rappresentata nel Listato B.1
All'interno della direcory *repository* sono disponibili gli eseguibili dello Autonomic Manager (directory *Controller*), del framework di monitoring WPM (directory *monitor*) e quelli del framework di benchmarking (directory *RadarGun-1.1.0-SNAPSHOT*). Nel caso in cui l'utente fosse intenzionato ad apportare modifiche al codice sorgente di tali progetti, dovrà farsi carico di ricompilare questi ultimi e sostituire i nuovi file eseguibili nelle rispettive directory.
Partendo dalla directory *repository*, è possibile incontrare i file di configurazione del framework di monitoring e del framework di benchmarking all'interno della directory *configs*. Il file di configurazione dello Autonomic Manager, *config.properties*, è invece presente all'interno della direcory *Controller/config*; tale file contiene una sufficiente documentazione al suo interno.
Per procedere all'installazione, è prima di tutto necessario specificare, nel file *node_ips*, gli indirizzi ed i ruoli di ciascun nodo; un nodo deve essere etichettato come Master, mentre i restanti come Slave. Un esempio di tale file è mostrato nel Listato B.2.
Il passo successivo consiste nella generazione e scambio delle chiavi pubbliche tra la coppia di nodi <Master, Slave> poichè è necessario che entrambi i nodi possano stabilire una connessione ssh tra di loro. Tale ope-

**Listing B.1: Semplificazione dell'algoritmo svolto dal componente Master**

```
.
|-- env.sh
|-- executeCmdTo.sh
|-- history.txt
|-- install_master.sh
|-- install_nodes.sh
|-- keyExchange.sh
|-- node_ips
|-- README.md
|-- repository
|   |-- configs
|   |-- Controller
|   |-- monitor
|   |-- RadarGun-1.1.0-SNAPSHOT
|-- scripts
    |-- master
    |   |-- setup.sh
    |   |-- start.sh
    |   |-- stop.sh
    |-- slave
        |-- nodeStart.sh
        |-- nodeStop.sh
```

razione può essere portata a termine eseguendo, all'interno la directory *AutonomicManagerScripts*, il seguente script.

```
$ bash ./keyExchange.sh
```

Si consiglia di configurare il sistema, come descritto nella sezione **??**, prima di procedere con i seguenti passi

È possibile procedere nell'installazione del nodo master e, successivamente, dei nodi slave eseguendo da linea di comando gli scipt:

```
$ bash ./install_master.sh
$ bash ./install_nodes.sh
```

## B.2 Uso

Il processo di installazione crea, all'interno del nodo master, una serie di script utili per avviare e fermare i processi relativi ai componenti LogService

**Listing B.2: Semplificazione dell'algoritmo svolto dal componente Master**

```
MASTER:vm-148-121.uc.futuregrid.org
SLAVE:vm-148-124.uc.futuregrid.org
SLAVE:vm-148-122.uc.futuregrid.org
SLAVE:vm-148-123.uc.futuregrid.org
SLAVE:vm-148-125.uc.futuregrid.org
```

e Master; tali script sono rispettivamente *start.sh* e *stop.sh*. All'avvio dei processi, sono mostrati i log del processo Master del framework di benchmarking. Nel frattempo, in un altra shell, è possibile avviare il processo dello Autonomic Manager eseguendo:

```
$ cd ./Controller
$ bash java -Djava.library.path="lib" \
        -cp lib/*:autonomicManager-1.0-SNAPSHOT.jar \
        eu.cloudtm.autonomicManager.Main
```

Se tutti i comandi sono andati a buon fine, lo Autonomic Manager dovrebbe presentare, a questo punto, una console interattiva a linea di comando, mostrata di seguito, dalla quale è possibile controllare la piattaforma.

```
$ java -Djava.library.path="lib" \
        -cp lib/*:autonomicManager-1.0-SNAPSHOT.jar \
        eu.cloudtm.autonomicManager.Main
 INFO ControllerLogger: Web Server + REST Interface started
 INFO Console: Actions:
 INFO Console: 0 - Exit
 INFO Console: 1 - Prints current platform configuration
 INFO Console: 2 - Prints current tuning configurations
 INFO Console: 3 - Change configurations
 INFO Console: 4 - WhatIfService
 INFO Console: 5 - Optimize now
 INFO Console: 6 - Enable/Disable WorkloadAnalyzer [ current: disabled ]
 INFO Console: 7 - Forecast (w/o reconfigure)
 INFO Console: 8 - Change forecaster
```

In alternativa è possibile aprire un browser web e connettersi all'indirizzo della macchina sulla quale è installato lo Autonomic Manager (porta 1515), facendo bene attenzione a controllare che tale macchina sia raggiungibile dall'esterno, senza applicazioni firewall che ne impediscano l'accesso ai servizi.