View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Electronic Thesis and Dissertation Archive - Università di Pisa

Università di Pisa
FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
Corso di Laurea Specialistica in Tecnologie Informatiche

Tesi di Laurea

REMEDY: UN SUPPORTO BASATO SU DHT PER LA RICERCA DI RISORSE DINAMICHE SU GRIGLIE COMPUTAZIONALI

Relatore:
Prof. Laura Ricci
Prof. Massimo Coppola

Candidato:
Emanuele Carlini

Anno Accademico 2007/08

Indice

1 Introduzione **4**

2 Stato dell'arte **9**

2.1 Introduzione . . . 9
2.2 Resource Discovery: problemi . . . 10
2.3 Resource Discovery Centralizzato: LDAP . . . 11
2.4 DHT . . . 14
2.4.1 Proprietà . . . 14
2.4.2 Struttura di una DHT . . . 14
2.5 MAAN . . . 16
2.6 Sword . . . 18
2.6.1 Architettura . . . 18
2.6.2 Struttura della Query . . . 20
2.6.3 Pubblicazione delle risorse e risoluzione di query . . . 21
2.7 InfoEye . . . 24
2.7.1 Architettura . . . 24
2.8 Peer-to-peer Discovery of Computational Resources for Grid Applications . . . 26
2.8.1 Architettura . . . 27

3 Remedy: Architettura generale **30**

3.1 Introduzione . . . 30
3.2 Chord . . . 30
3.3 Funzioni Hash . . . 32
3.4 Remedy: l'architettura generale . . . 35

3.4.1 Registrazione 36
3.4.2 Risoluzione di range query con singolo attributo 37
3.4.3 Risoluzione di range query multiattributo: metodo iterativo 37
3.4.4 Risoluzione con attributo dominante 38
3.5 Remedy: pubblicazione di attributi dinamici 40
3.5.1 Definizione del problema 40
3.5.2 Ottimizzazione del routing 42
3.5.3 Diminuzione del numero di pubblicazioni 45
3.5.4 Utilizzo della popolarità 48

4 Overlay Weaver 50
4.1 Introduzione 50
4.2 Architettura 51
4.2.1 Routing Driver 53
4.3 Utilizzo di Overlay Weaver 53
4.3.1 Utilizzo dei servizi ad alto livello 54
4.3.2 Modificare le funzionalità delle DHT 57
4.3.3 Implementare un nuovo algoritmo di routing 61
4.4 Strumenti di sviluppo 64
4.4.1 Emulatore 64
4.4.2 Visualizer 67

5 Remedy: Implementazione 68
5.1 Package 68
5.2 Il package Data 69
5.3 Pubblicazione 72
5.3.1 Pubblicazione: nodo provider 72
5.3.2 Pubblicazione: nodo destinatario 74
5.4 Risoluzione delle query 76
5.4.1 Query: nodo provider 76
5.4.2 Query: nodo destinatario 78

6 Risultati sperimentali **81**
6.1 Confronto fra Overlay Weaver e Bamboo 81
6.1.1 Organizzazione dei test 81
6.1.2 Primo test di scalabilità 83
6.1.3 Secondo test di scalabilità 86
6.1.4 Test di Affidabilità 89
6.1.5 Conclusioni 92
6.2 Remedy: risultati sperimentali 93
6.2.1 Organizzazione dei test 93
6.2.2 Numero di nodi visitati nella risoluzione di query . . . 94
6.2.3 Test sulla selettività 96
6.2.4 Test sul numero di attributi 97
6.2.5 Conclusioni 98
6.3 Analisi di dati reali 99
6.3.1 PlanetLab 100
6.3.2 Distribuzione dei dati 104
6.3.3 Churn 107
6.3.4 Funzioni Hash e bilanciamento del carico 109

Lista delle immagini **119**

List of Tables **122**

A Emulatore **123**
A.1 Comandi 123

B Grid5000 **126**
B.0.1 Accesso a GRID5000 126
B.0.2 Utilizzo di grid5000 126
B.0.3 Panoramica dei comandi 128

Capitolo 1

Introduzione

Negli ultimi anni si è assistito ad una crescente diffusione di sistemi computazionali distribuiti chiamati Grid. Le ricerche in ambito accademico ed industriale hanno come obiettivo quello utilizzare tali reti come piattaforme per l'utilizzo di sistemi distribuiti. Un efficiente meccanismo di *Resource Discovery* (RD) è uno dei requisiti fondamentali per un Grid, utilizzato per la gestione delle risorse e lo scheduling delle applicazioni. Il processo del RD consiste nella ricerca di risorse che corrispondono alla richieste effettuate dagli utenti.

Per la risoluzione del problema del resource discovery sono state proposte molte varianti, alcune di queste applicabili ad infrastrutture come le Grid. I primi tipi di soluzioni erano di tipo centralizzato; l'esponente principale di questa categoria è sicuramente LDAP [17], un sistema basato sul modello client/server. Tutte le richieste sono inviate ad un unico server che si occupa di rispondere alla richiesta. Il vantaggio di questo tipo di soluzione è la facilità di gestione dell'insieme dei dati, il suo limite è invece rappresentato dal singolo punto di fallimento che il server rappresenta.

Con la crescita delle reti, i sistemi centralizzati hanno mostrato tutti i loro limiti, e seppure usati ancora oggi in moltissimi ambiti applicativi, si assiste ad un progressivo sviluppo di protocolli *peer to peer* (P2P). A differenza delle soluzione centralizzate in cui ogni nodo ha il proprio ruolo statico, una rete P2P è composta da nodi equivalenti (i *peer* appunto) che hanno funzioni sia

di server che di client.

Negli ultimi anni si è sviluppata la tendenza di utilizzare protocolli P2P per l'organizzazione dei servizi di RD su Grid. Tali servizi si basano principalmente su una categoria di sistemi P2P strutturati chiamati *Distributed Hash Table* (DHT) che forniscono le funzionalità delle hash table ma distribuiscono i dati sulla rete.

Uno degli aspetti chiave delle DHT è l'utilizzo di funzioni hash per gestire il bilanciamento del carico. I nodi ottengono degli identificatori appartenenti ad uno spazio uniformemente popolato. La funzione hash applicata alla descrizione di una risorsa restituisce una chiave, appartenente allo stesso spazio degli identificatori di cui fanno parte i nodi, permettendo di allocare la risorsa sulla DHT. Un altro aspetto importante delle DHT sono le prestazioni, generalmente logaritmiche sul numero di nodi, sia per quanto riguarda la dimensione della tabella di routing sia per il numero di passi necessari a localizzare una risorsa.

Le attuali implementazioni delle DHT sono efficienti per la risoluzione di query ad una dimensione [26] in cui si cerca un valore esatto. Questa caratteristica dipende dall'utilizzo di funzioni hash che mappano le risorse in modo casuale sulla rete e che non consentono di conservare alcuna nozione di località.

Con *range query multi attributo* si intende una richiesta in cui sono presenti più attributi ed per almeno uno di essi è indicato come vincolo un *intervallo* di valori. In sistemi Grid questo tipo di interrogazioni sono molto comuni nelle fasi di ricerca di risorse computazionali.

L'estensione delle funzionalità di una DHT in modo da supportare anche range query in più dimensioni è un problema complesso.

In letteratura sono state proposte molte soluzioni che si preoccupano di risolvere in modo efficiente le problematiche legate alla scoperta di risorse su Grid [27]. In particolare uno dei problemi più complessi è quello della gestione delle *risorse dinamiche*, i cui valori cambiano spesso nel tempo. Il problema è da ricercarsi nel fatto che l'aggiornamento di una risorsa genera una quantità di traffico sulla rete, in termini di messaggi, non trascurabile. Iterare questo processo al ritmo di cambiamento delle risorse dinamiche potrebbe rallentare

la rete.

Questa tesi propone Remedy (**Re**duced **me**ssage on **d**ht modif**y**), un approccio basato su DHT per la risoluzione di range query multi attributo. Remedy offre funzionalità per la risoluzione di range query mappando i valori degli attributi su Chord [11], una particolare versione di DHT.

L'approccio è caratterizzato dall'utilizzo di una *funzione di hash che preserva la località*. In questo modo risorse vicine rimangono tali anche nello spazio degli identificatori.

Uno degli aspetti principali dell'approccio è quello di utilizzare la *selettività* di una query. Si definisce selettività, rispetto ad un attributo, il rapporto tra la dimensione dell'intervallo specificato nella query e la dimensione dello spazio degli identificatori. L'attributo detto *dominante* è quello a cui, all'interno di una query, corrisponde la più alta selettività possibile (quello in cui il rapporto di cui sopra si avvicina di più a zero). Il numero di passi da effettuare per risolvere una range query dipende infatti dalla selettività dell'attributo dominante della query. Quando la selettività è molto alta il numero di passi è logaritmico rispetto al numero di nodi, cosa che permette al sistema di essere scalabile indipendentemente dal numero di attributi utilizzati per descrivere una risorsa.

Uno degli aspetti innovativi della tesi è la proposta di una *ottimizzazione per la pubblicazione di risorse con attributi dinamici*, in modo da cercare di limitare il numero di messaggi inviati nella fase di pubblicazione delle risorse.

L'ottimizzazione è composta da due strategie differenti ed indipendenti, ma che possono coesistere. La prima strategia si basa sull'utilizzo di *cache* per memorizzare i routing più frequenti. La seconda sull'introduzione del concetto di *popolarità* di un attributo. La popolarità di un attributo è la frequenza con la quale un attributo viene scelto come dominate durante la fase di risoluzione delle query. L'idea è di utilizzare questa informazione per aggiornare con minor frequenza gli attributi con la popolarità più bassa in modo da ridurre i messaggi generati in fase di pubblicazione mantenendo il più possibile corretta la risoluzione di una query.

Una serie di misurazioni sono state utilizzate come campione per verificare qual è il grado di miglioramento che dobbiamo aspettarci dalla ottimizzazioni

utilizzando *dati reali*. Queste misure sono rilevazioni dei principali parametri, statici e dinamici, di workstation facenti parte di PlanetLab [19], una piattaforma composta da più cluster in cui sono attivi vari servizi.

Inoltre, utilizzando questi dati, si sono studiate funzioni di hashing specifiche per ogni attributo. Questo tipo di funzioni, basate sull'*effettiva distribuzione dei valori* dell'attributo, consentono una migliore gestione del bilanciamento del carico. Sono stati forniti inoltre spunti di ricerca su come dare una valutazione della distribuzione delle query, per ottenere un modello di comportamento della popolarità basato sulle distribuzioni *power-law* [28, 29].

La proposta è accompagnata da una serie di test sperimentali svolti principalmente su reti reali. I siti che hanno ospitato queste prove sono stati due: Pianosa, un cluster omogeneo composto da 32 nodi e Grid5000 un sistema Grid composto da più cluster eterogenei. In primo luogo sono stati effettuati test di comparazione fra due diverse implementazioni di DHT, Overlay Weaver [5] e Bamboo [13] per decidere quale utilizzare per l'implementazione di Remedy. Le prestazioni delle due piattaforme si sono rivelate confrontabili ma si è scelto di utilizzare Overlay Weaver in quanto più modulabile e facilmente configurabile per le diverse esigenze.

Sono stati effettuati dei test con Remedy per verificare la correttezza dell'implementazione e i possibili benefici, in termini di latenza per la risoluzione di query, derivanti dalla diminuzione del numero di messaggi durante la fase di pubblicazione. Questi test hanno mostrato che la riduzione del traffico in fase di pubblicazione delle risorse porta ad un effettivo beneficio per il tempo di risposta di risoluzione di una query.

La tesi è strutturata come segue. Nel capitolo 2 è presentata una selezione di sistemi che affrontano il problema del resource discovery. É descritto LDAP [17], un sistema centralizzato che ha lo scopo di fornire un modello di riferimento, per poi descrivere alcuni sistemi che si basano su DHT [3] ed in particolare gli approcci che danno una qualche rilevanza ad aspetti come le risorse dinamiche [1, 10, 14]. Nel capitolo 3 viene illustrato Remedy

con le relative ottimizzazioni. Nel capitolo 4 è presente una panoramica di Overlay Weaver, orientata alla modifica ed alla estensione dello strumento. Nel capitolo 5 sono presentati gli aspetti implementativi principali che caratterizzano Remedy ed è presente lo pseudo codice delle operazioni principali. Nel capitolo 6 sono presenti vari risultati sperimentali e l'approfondimento di alcuni aspetti come, ad esempio, funzioni hash specifiche per ogni attributo. Infine il capitolo delle conclusioni oltre a riassumere l'esperienza di questa tesi, descrive alcuni dei possibili sviluppi futuri.

Capitolo 2

Stato dell'arte

2.1 Introduzione

Negli ultimi anni è emersa una nuova generazione di sistemi distribuiti, che operano dinamicamente e in modo decentralizzato. Le due principali categorie sono rappresentate dai sistemi di *Grid Computing* (Grids) e da quelli *peer to peer* (P2P).

I sistemi Grids forniscono le infrastrutture di base per condividere insiemi di risorse computazionali, siano essi clusters, supercomputer o semplici desktop.

Un meccanismo efficiente di ricerca delle risorse è uno dei requisiti fondamentali per i sistemi Grids. L'attività di *Resource discovery* implica la ricerca delle risorse che corrispondano alle richieste degli utenti. Vari tipi di soluzioni sono state suggerite per questo problema, primi fra tutti in ordine cronologico gli approcci centralizzati e gerarchici. Tuttavia entrambe queste soluzioni hanno serie limitazioni riguardo alla scalabilità, alla fault tolerance e alla congestione della rete. Per risolvere questi problemi, sono stati proposti meccanismi P2P per essere affiancati ai sistemi Grids.

In questo capitolo sono inizialmente presentate le principali problematiche relative alla ricerca di risorse su reti di larga scala, per poi descrivere un tipico sistema che risolve il problema in modo centralizzato. Successivamente si descrivono vari approcci P2P, in particolare riguardo all'utilizzo di Distri-

bited Hash Tables (DHTs) le cui caratteristiche generali sono descritte nel primo paragrafo. Nei successivi, vedremo inizialmente una proposta per la risoluzione di range query chiamata MAAN e successivamente un sistema di *resource discovery* completo, chiamato SWORD, per poi soffermarci in modo approfondito sulle soluzioni per il *resource discovery* in ambienti le cui risorse hanno attributi dinamici. Sono presentati due approcci, il primo basato non su DHT, l'altro studiato apposta per integrarsi con le distribuited hash tables.

2.2 Resource Discovery: problemi

La principale difficoltà nell'utilizzo di sistemi distribuiti su larga scala come le Grids, è quella di localizzare nel miglior modo possibile i sottoinsiemi di risorse su cui ospitare servizi e applicazioni.

Il servizio di resource discovery è utilizzato in molti ambiti applicativi, data la eterogeneità di risorse con cui sono tipicamente composte le infrastrutture di rete su larga scala.

Applicazioni cpu-intensive, come ad esempio applicazioni di calcolo parallelo, sono interessate sopratutto a fattori quali il carico del processore e la quantità di memoria. Applicazioni di rete come le content distribution networks (CDN) si focalizzano su banda e latenze fra i nodi per ottimizzare la propria topologia. Altre applicazioni hanno necessità sia per quanto riguarda la potenza di calcolo, sia del punto di vista della rete, pensiamo ad esempio ai giochi multiplayer. D'altra parte caratteristiche aggiuntive possono essere utili a tutti i tipi di applicazioni. Ad esempio il fatto di utilizzare nodi il cui livello di disponibilità è storicamente alto, può aumentare la qualità del servizio.

La sfida lanciata del resource discovery è di fatto molto complessa, in quanto le problematiche da affrontare sono molte.

Per prima cosa il sistema deve essere *scalabile*, mantenere cioè un certo livello di prestazioni anche con un alto numero di nodi ed in aggiunta garantire un certo livello di *disponibilità*. In secondo luogo, dato che alcune caratteristiche dei nodi cambiano il loro valore molto velocemente, *attributi*

dinamici, deve essere possibile fornire una 'fotografia' della rete la meno obsoleta possibile. Infine, oltre alla dinamicità appena citata, nella descrizione dei nodi è presente sia una componente *statica* di cui è necessario tenere traccia, sia una componete che mette il nodo in relazione con altri definita dagli *attributi inter-nodo*.

2.3 Resource Discovery Centralizzato: LDAP

Il *Lightweight Directory Access Protocol* (LDAP) è uno standard che definisce metodi per l'accesso e l'aggiornamento di risorse in una directory. In particolare è utilizzato nei Grid Information System per la pubblicazione di risorse ed è dunque un possibile candidato a svolgere il compito della resource discovery nelle Grids. In questo paragrafo darò una breve descrizione di quelli che sono i principali concetti e funzionalità del protocollo.

Come già detto LDAP fornisce metodi per l'accesso di dati presenti su *directory*. La directory si discosta dal concetto di database relazionale (DB).

Per prima cosa, gli accessi ad una directory riguardano per la maggior parte operazioni di ricerca e questo rende necessaria una certa ottimizzazione nella fase di lettura. I database tradizionali invece, sono orientati verso un supporto non discriminante in termini di prestazioni fra operazioni di lettura e scrittura. Un'altra differenza risiede nel metodo di accesso alle informazioni. I DB si interrogano tramite un linguaggio strutturato come SQL e di conseguenza la sua gestione in un'applicazione può risultare costosa. Il punto di forza di LDAP, invece, consiste nella sua semplicità permettendogli di non gravare sulla complessità di una applicazione. D'altra parte, proprio per questo motivo LDAP non offre tutte le funzioni di un DB general purpouse. Il concetto di transazione, ad esempio, non è previsto in quanto permette di non appesantire il protocollo e per un servizio basato principalmente su letture è possibile accettare in rari casi dati inconsistenti.

Una directory in LDAP è strutturata come un *Directory Information Tree* (DIT), un modello gerarchico ad albero di rappresentazione dei dati. In generale nell'implementazione LDAP questo albero è strutturato in modo da riflettere una divisione in categorie dipendente dal modello scelto. I raggrup-

pamenti, ad esempio, possono essere fatti seguendo canoni politici, piuttosto che organizzativi o geografici. Inoltre LDAP non definisce nessuna specifica per come devono essere memorizzati i dati, difatti molte sue implementazioni utilizzano come back-end database standard.

Il funzionamento di LDAP è basato su iterazioni di tipo client/server, cosa che lo rende specifico per soluzioni di tipo centralizzato. Una generica interazione fra un client ed un server LDAP si compone di diversi passi. Inizialmente il client stabilice una sessione con il server, operazione questa che prende il nome di *binding*. Normalmente il client fornisce, per questa operazione, gli estremi per l'autenticazione ma sono previsti anche bind anonimi. A questo punto il client è pronto per poter eseguire operazioni di scrittura o di lettura sulla directory, attraverso delle query. Infine, una volta completate tutte le operazioni, il client chiude la sessione con il server tramite l'operazioni di *unbinding*.

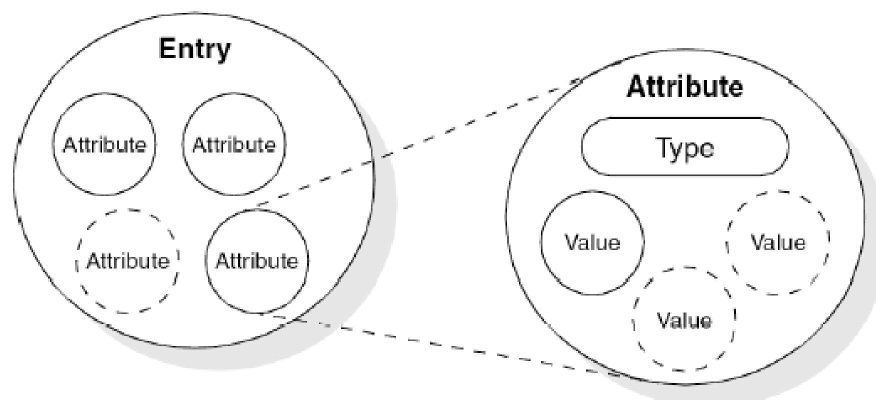

Figura 2.1: Relazioni fra entries, attributi e valori in LDAP

L'unità base di informazione utilizzata nel DIT è chiamata *entry* che rappresenta sostanzialmente una collezione di attributi. Le relazioni fra una entry di una directory e i suoi attributi sono mostrate in figura 2.1 Essendo organizzate una struttura ad albero ogni entry ha in generale ha un genitore e dei figli. Ogni entry è identificata univocamente dal suo *Distinguished Name* (DN). Il DN è formato da una insieme di RDN (Relative Distinguished Name) trovati navigando l'albero verso la radice a partire dall'entry a cui ci riferiamo.

Nel caso più semplice, un RDN è formato da una coppia (attributo, valore). Facendo un paragone con la struttura ad albero di un filesystem tradizionale, dove consideriamo come entry i file, possiamo pensare al DN come ad un path assoluto, mentre all'RDN come un path relativo, considerando sempre la distinzione che in un filesystem un path è descritto partendo dalla radice dell'albero, mentre il DN è scritto a partire dall'entry.

Un singolo server LDAP può non contenere l'intero DIT ma solo un sottoinsieme di esso e per questo motivo i server devono essere collegati tramite un meccanismo che permetta all'utente l'accesso a tutti i dati. Il problema è risolto utilizzando un sistema di riferimenti, chiamati *referrals.* quando un client invia una richiesta ad un server LDAP, la risposta può essere un *referral*, cioè un riferimento ad un altro server LDAP che contiene la risorsa cercata.

Per permettere al generico utente di accedere ai dati LDAP mette a disposizione varie operazioni, le quali possono essere divise in tre macrocategorie:

- Autenticazione: include le operazioni di bind e unbind accennate prima, oltre che l'operazione di *abbandon* utilizzata per interrompere un'operazione richiesta precedentemente.
- Query: operazioni di ricerca e comparazioni, in generale permettono di ottenere le informazioni presenti sulle directory.
- Aggiornamento: fanno parte di questa categoria le operazioni di inserimento, cancellazione e modifica dei dati presenti sulla directory.

Le operazioni più comune in ambiente LDAP sono quelle di query attraverso le quali l'utente richiede informazioni che devono soddisfare alcuni criteri. I parametri per definire questi criteri sono molti. Per prima cosa si deve stabilire un oggetto di partenza che rappresenta un oggetto del DIT, detto *base.* Lo *scope* indica il livello di profondità con cui cercare a partire dall'oggetto base ed è possibile indicare un numero massimo di oggetti che compongo la risposta. Un'altra parte importante delle query sono i filtri di ricerca in cui sono descritte una serie di vincoli per gli attributi specificati con condizioni come maggiore, minore od uguale.

2.4 DHT

Una DHT è un sistema distribuito che fornisce un servizio di ricerca simile a quello di una hash table: un dato è associato ad una chiave. Le coppie (chiave, valore) sono infatti l'entità minima memorizzata su una DHT. Ogni nodo appartenente all'overlay può, in generale, recuperare un valore se conosce la chiave corrispondente. Le operazioni per il mantenimento di una overlay strutturata come la DHT sono distribuite tra i nodi, in modo da limitare i danni relativi al fallimento di un insieme di nodi e ottenere prestazioni accettabili anche con reti su larga scala.

2.4.1 Proprietà

La principale caratteristica di una DHT è quella di essere un sistema strutturato completamente distribuito. Questo consente alle DHT di avere due proprietà interessanti:

- *Scalabilità*: la rete funziona ed ha prestazioni accettabili anche con una grande quantità di nodi.
- *Tolleranza ai guasti*: il funzionamento della rete prescinde dall'uscita di nodi dalla rete.

D'altra parte le DHT devono fare i conti con i soliti problemi relativi ai sistemi distribuiti: integrità dei dati, bilanciamento del carico, sicurezza.

2.4.2 Struttura di una DHT

La struttura di una DHT è composta da alcuni componenti principali. Le sue entità, chiavi e nodi, sono rappresentate da identificatori che fanno parte di uno *spazio astratto*, che può essere rappresentato, ad esempio, da una stringa di 160 bit.

La gestione di questo spazio è partizionata fra i nodi partecipanti. Una *overlay network* connette i nodi, fornendo la possibilità di identificare il proprietario di una qualsiasi chiave dello spazio.

Vediamo un esempio del funzionamento base di una DHT, in cui supponiamo che lo spazio delle chiavi sia rappresentato da una stringa di 160 bit. Per memorizzare una risorsa si calcola la chiave k associata, generalmente tramite tecniche di hashing, ed un messaggio di tipo *put(k, risorsa)* è inviato a un nodo qualsiasi della rete. Il messaggio attraversa l'overlay network fino a raggiungere il nodo che è responsabile della chiave k. Questo nodo è definito dalla politica di partizionamento delle chiavi e memorizza la coppia (k, risorsa). Un nodo che vuole ottenere la risorsa, calcola k e chiede alla DHT di trovarne il dato associato mediante l'invio di un messaggio del tipo *get(k)*. Analogamente al messaggio precedente esso attraversa l'overlay fino al nodo responsabile di k il quale risponde al soggetto che ha inviato la richiesta.

Spazio delle chiavi

La metodologia di gestione dello spazio delle chiavi è un componente fondamentale di ogni DHT. Quelli descritti sono concetti generali, che differiscono fra varie DHT per i dettagli.

La maggior parte delle DHT utilizzano varianti del *consistent hashing* per associare *identificatori* (ID) ai nodi della rete. Un nodo con ID i gestisce tutte le chiavi per le quali i è l'ID più vicino. Questa relazione di vicinanza è descritta da una funzione che definisce una distanza astratta fra gli identificatori. In generale quindi, la distanza non dipende dalla distanza geografica, dalla latenza fra due nodi o da altri parametri reali.

Una proprietà essenziale delle DHT è che la rimozione o l'arrivo di un nodo provocano cambiamenti solo negli insiemi di chiavi gestite dai nodi con un ID vicino, rendendo performante la DHT anche quando il livello di *churn* (rimozione ed inserzione di nodi) è alto. Quando la frequenza di churn è troppo elevata può rappresentare comunque un problema. Sono state proposte varie soluzioni al problema coma ad esempio in [13].

Overlay Network della DHT

Nelle overlay network ogni nodo mantiene un insieme di collegamenti ad altri nodi in una struttura dati chiamata *Routing Table* (tabella di routing) o *Nei-*

ghbors Set (insieme dei vicini). Un nodo sceglie chi considerare propri vicini (con chi mantenere un collegamento) sulla base di una struttura chiamata *topologia* della rete.

In generale le topologie delle DHT condividono tutte la seguente proprietà, data una qualsiasi chiave k, un generico nodo N gestisce k oppure esiste nella tabella di routing di N un collegamento ad un nodo che è più vicino a k. L'algoritmo che definisce il routing è di conseguenza di tipo greedy: ad ogni passo si inoltra il messaggio al nodo della routing table il cui ID è il più vicino possibile a k. Se non esiste nessun nodo di questo tipo, il messaggio è arrivato a destinazione. Questo tipo di routing è chiamato *key-based routing* [4].

La topologia di una DHT deve soddisfare due proprietà fondamentali: garantire che il massimo numero di *hops* sia basso per un routing qualsiasi e limitare il numero di collegamenti presenti nella routing table. Queste proprietà permettono rispettivamente di diminuire il tempo di risposta di una richiesta e ridurre l'overhead di mantenimento della routing table.

La scelta più comune è quella di avere sia la dimensione della tabella di routing che la lunghezza del percorso logaritmici rispetto alla dimensione della rete, come ad esempio succede in Chord [11]. Esistono molti altri algoritmi di DHT, tra i più importanti Kademlia [15], Pastry [12], e CAN (Content Adressable Networ) [16].

2.5 MAAN

Multi-Attribute Addressable Network (MAAN) è un estensione di una DHT (Chord). Queasta estensione arricchisce la DHT con un sistema che supporta la risoluzione di range query con più attributi [3].

Una risorsa è identificata da più coppie attributo-valore. Per ogni attributo esiste (staticamente) una funzione hash che ha il compito di mappare il valore dell'attributo stesso nell'intervallo $[0, 2^m - 1]$, dove m è la dimensione in bit dello spazio degli indirizzi.

Per inserire una risorsa nel sistema si memorizza, per ogni attributo, il suo valore al nodo opportuno, rappresentato dal successore sull'anello di Chord del valore stesso.

In realtà le informazioni che si memorizzano, oltre al valore dell'attributo, sono diverse. Fra queste troviamo gli estremi che identificano il nodo proprietario della risorsa (tipicamente: indirizzo ip e porta) e, centrale per il funzionamento di MAAN, sono memorizzate tutte le altre coppie attributo-valore relative alla risorsa che si vuole registrare. Di fatto una risorsa è registrata un numero di volte pari alla quantità di attributi.

Nella soluzione proposta da MAAN risolvere una range query significa risolvere tutte le sotto queries di cui è composta, una per ogni attributo. L'obiettivo è quello di riuscire a risolvere la query completa, sfruttando la DHT per risolvere solamente una delle sottoqueries, quella che corrisponde all'attributo dominate.

Il primo passo per la risoluzione di una richiesta è quindi la scelta dell'*attributo dominante* (l'attributo che identifica la query principale da risolvere). In generale l'attributo dominate scelto cambia da query a query in quanto la scelta cade su quello che garantisce la massima selettività possibile, per permettere di ridurre il numero di nodi da interrogare. Questa scelta è centrale in un modello di prestazioni come quello delle DHT, in cui si cerca di minimizzare il numero di messaggi che transitano sulla rete.

Successivamente si interrogano in successione i nodi il cui ID è nell'intervallo dei valori dell'attributo dominante. Considerando che:

1. tutta la query è trasportata nel messaggio di richiesta;

2. tutte le coppie attributo-valore della risorsa sono registrate;

ogni nodo interrogato trova le risorse che soddisfano la query eseguendo un intersezione in *locale*, senza l'invio di ulteriori messaggi.

In conclusione in [3] è proposta un architettura le cui performance di routing sono *indipendenti dal numero degli attributi*, che garantisce scalabi-

lità per le query multi attributo. I possibili overheads si identificano nella quantità di memoria per memorizzare tutta la risorsa e, aspetto più rilevante, nell'alto costo di aggiornamento nel caso in cui il valore di un attributo cambi.

2.6 Sword

SWORD [10] è un'architettura complessa che ha come obiettivo la risoluzione di range query sulle DHT. Oltre alle range query 'classiche' basate sugli attributi di un nodo, SWORD si distingue per aver integrato numerose funzionalità aggiuntive al processo di *resource discovery*. La possibilità dell'utente di determinare gruppi di nodi, con caratteristiche anche diverse, all'interno delle query, l'utilizzo di attributi inter-nodo (ad esempio la latenza fra due nodi) e l'introduzione di un sistema di costi per una valutazione qualitativa delle soluzioni trovate, permettono a SWORD di essere considerato uno degli approcci più completi nel campo del resource discovery su DHT.

2.6.1 Architettura

L'obiettivo di SWORD (e di un sistema di resource discovery in generale) è quello di permettere agli utenti di localizzare su una rete un sottoinsieme di risorse sulle quali eseguire le proprie applicazioni. Gli utenti possono esprimere i requisiti per un certo insieme di nodi, specificando due tipi di intervalli. Un intervallo più ampio che è il *range richiesto*, ed un altro, sottoinsieme del primo, che rappresenta il *range desiderato*. Inoltre è possibile per l'utente definire, a livello di query, *gruppi* creando di fatto dei veri e propri clusters virtuali di nodi. La selezione di nodi al di fuori dell'intervallo desiderato, ma comunque dentro quello richiesto, aggiungono un costo, detto *penalità*, alla risposta. La penalità rappresenta la distanza fra la richiesta di un utente e l'effettiva soluzione trovata.

L'intento finale dell'infrastruttura è quindi quello di trovare la soluzione che soddisfi la richiesta degli utenti che abbia la minore penalità possibile.

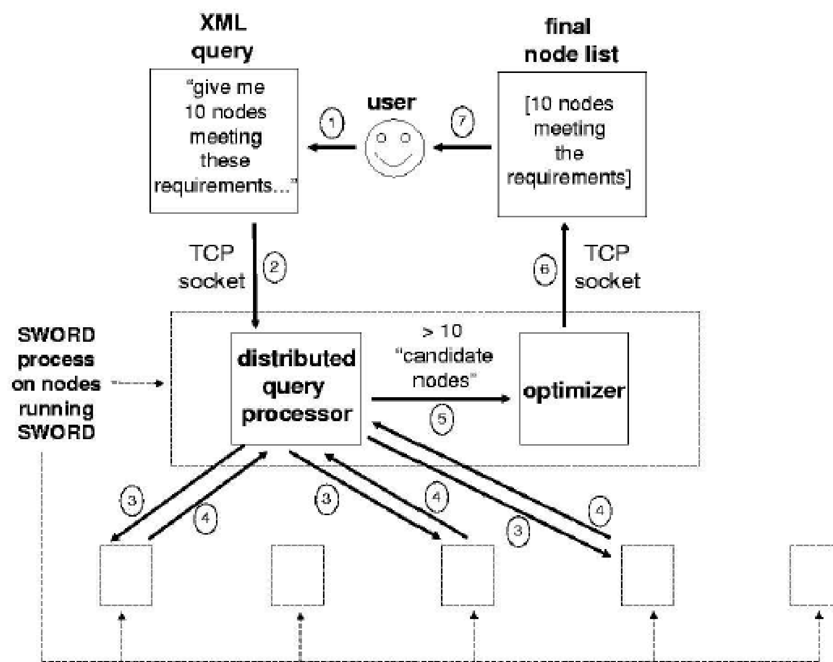

Figura 2.2: Architettura ad alto livello di SWORD

In figura 2.2 è rappresentata in modo astratto l'architettura di SWORD. Il processo di resource discovery è iniziato dall'utente (punto 1) il quale sottomette una query in XML, inviata ad un nodo qualsiasi facente parte di SWORD. La query è poi inoltrata al *distribuited query processor* (punto 2), componente che si occupa della risoluzione distribuita (punti 3 e 4). Come avviene l'effettiva risoluzione della query e come vengono scelti i nodi verrà spiegato successivamente. I nodi candidati trovati sono inviati all'*optimizer* (punto 5), il quale, dopo una fase di ottimizzazione, invia infine un sottoinsieme di essi all'utente (punto 6).

2.6.2 Struttura della Query

Una query in SWORD è rappresentata in formato XML ed è divisa logicamente in tre parti. La prima parte definisce i vincoli sulle risorse da utilizzare per la risoluzione della query, parametrizzando, ad esempio, il numero massimo di nodi da contattare nella fase di risoluzione.

Nella seconda parte sono definiti gli intervalli per gli attributi intra-nodo e inter-nodo. Inoltre vengono definiti i gruppi, ognuno dei quali può avere vincoli diversi a seconda delle esigenze.

Infine nell'ultima parte sono definiti i vincoli inter-nodo fra nodi che appartengono a gruppi *diversi*. Ad esempio si può rendere obbligatoria l'esistenza di un nodo in ogni gruppo tale che la sua latenza fra lui ed un nodo di un altro gruppo non superi una soglia specificata.

Un esempio di query, con relativa risposta, è presente in figura 2.3. In questo esempio la query in formato XML è stata convertita in un formato più leggibile, per necessità di presentazione. La parte a sinistra rappresenta la query vera e propria, in cui è visibile la descrizione di due gruppi di nodi (NA e Europe). Per ogni attributo sono definiti due intervalli di valori distinti quelli richiesti e quelli preferiti con relativa penalità. Nella sezione *InterGroup* sono presenti vincoli relativi al rapporto fra i gruppi. Nell'esempio in particolare è richiesto che almeno una coppia di nodi, ognuno di uno dei due gruppi, abbia una banda di almeno 3 Gb/s.

```
RequeryInterval 60
Group NA
 NumMachines 4
 Required Load [0.0, 2.0]
 Preferred Load [0.0, 1.0], penalty 100.0
 Required FreeDisk [500.0, MAX]  (MB)
 Preferred FreeDisk [1000.0, MAX], penalty 0.2
 Required OS [``Linux'']
 Required AllPairs Latency [0.0, 20.0] (ms)
 Preferred AllPairs Latency [0.0, 10.0], penalty 2.0
 Required AllPairs BW [0.5, MAX] (Mb/s)
 Preferred AllPairs BW [1.0, MAX], penalty 2.0
 Required Location [``NorthAmerica'', 0.0, 50.0] (ms)
Group Europe
 NumMachines 4
 Required Load [0.0, 2.0]
 Preferred Load [0.0, 1.0], penalty 100.0
 Required FreeDisk [300.0, MAX]  (MB)
 Preferred FreeDisk [1000.0, MAX], penalty 100.0
 Required OS [``Linux'']
 Required AllPairs Latency [0.0, 20.0] (ms)
 Preferred AllPairs Latency [0.0, 10.0], penalty 2.0
 Required AllPairs BW [0.5, MAX] (Mb/s)
 Preferred AllPairs BW [1.0, MAX], penalty 2.0
 Required Location [``Europe'', 0.0, 50.0] (ms)
InterGroup
 Required OnePair BW NA Europe [3.0, MAX] (Mb/s)
 Preferred OnePair BW NA Europe [5.0, MAX], penalty 0.5
```

```
SWORD Groups
-------------------
  Name: Group NA
  Max Latency: 20.0 ms
  Min Latency: 0.0 ms
  Overall Group Cost: 4.1
  Group Size:  4
    planetlab6.nbgisp.com
    planet1.halifax.org
    planet02.csc.ncsu.edu
    planetlab1.kscy.org


  Name: Group Europe
  Max Latency: 20.0 ms
  Min Latency: 0.0 ms
  Overall Group Cost: 2.2
  Group Size:  4
    planetlab-02.lip6.fr]
    planetlab2.dcs.ac.uk]
    planetlab-01.lip6.fr]
    planetlab2.uni-kl.de

  Total cost: 6.3
  Total query time:  0.095 s
```

Figura 2.3: Esempio di query in sword

2.6.3 Pubblicazione delle risorse e risoluzione di query

Per risolvere il problema delle range query multi attributo nella forma descritta in figura 2.3 SWORD ha implementato e studiato quattro approcci differenti.

- *SingleQuery.* Nella fase di pubblicazione si inviano tutte le informazioni ad n insiemi di servers, dove n è il numero di attributi con cui si descrivono le risorse. Ognuno di questi insiemi di server si occupa della gestione di un solo attributo, pur memorizzando l'intera descrizione della risorsa. Nella fase di risoluzione di query, la richiesta è inviata ad un nodo che gestisce un qualsiasi chiave di un qualsiasi attributo all'interno del range della query. Questo nodo poi inoltra la richiesta ai nodi successivi nello spazio degli indirizzi della DHT. Questo approccio è simile a quello specificato in MAAN [3].

- *MultiQuery.* Esiste un insieme di nodi che si occupa esclusivamente della gestione di un solo attributo. La fase di pubblicazione consiste

in n messaggi, ognuno dei quali contiene una sola coppia (attributo, valore). Rispetto al metodo SingleQuery la dimensione del messaggio di aggiornamento è più piccola, ma nella risoluzione della query sono inviati n messaggi, uno per ogni insieme di server. I risultati ottenuti vengono poi intersecati localmente al nodo che ha inviato la query.

- *Fixed.* Questo è sostanzialmente un approccio centralizzato in cui si utilizza un numero variabile di server. La risorsa è inviata ad uno solamente degli n server presenti, scelto a caso all'avvio del nodo, per distribuire il carico uniformemente fra i server. Nella fase di risoluzione della query sono interrogati i rimanenti $n - 1$ servers che inviano i risultati come risposta.

- *Index.* Questo approccio è un ibrido fra la soluzione distribuita e quella centralizzata. Esistono dei nodi speciali, *index node*, che hanno la funzione di mappare un range nello spazio delle chiavi con il nodo della DHT che lo gestisce. Periodicamente ogni nodo della DHT invia informa uno degli index node del range che gestisce. Per la fase di aggiornamento possono essere utilizzati gli stessi metodi presenti negli approcci SingleQuery e MultiQuery. La query è inviata agli index node, i quali si occupano di inoltrala ai nodi della DHT che gestiscono i range richiesti.

La chiave per associare le informazioni ai nodi presenti nella DHT è costruita come segue. Ad ogni attributo A è associata una funzione monotona crescente $f_A(x)$ che associa al valore x un identificatore sulla DHT. Le query sono inviati ai nodi che gestiscono questi identificatori. Ad esempio, con l'approccio SingleQuery, viene scelto un attributo B come attributo di ricerca e si calcolano le chiavi corrispondenti a $f_B(n)$ e $f_B(m)$ dove n e m rappresentano il valore minimo e massimo specificato nel vincolo per l'attributo B. Utilizzando il routing proprio della DHT si trova il nodo che gestisce $f_B(n)$. A questo punto la query è inoltrata ai nodi successivi finché non si incontra il nodo gestore di $f_B(m)$. Il fatto che la funzione f sia monotona e crescente, ci assicura che tutti i nodi che gestiscono il range specificato nella query siano contattati.

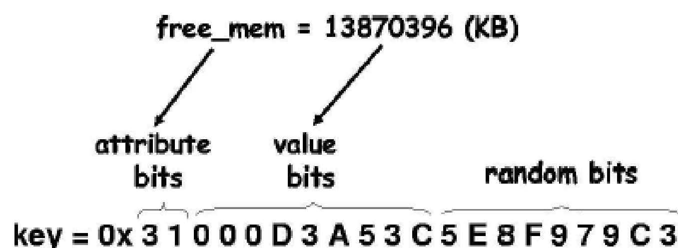

Figura 2.4: Mapping tra il valore di un attributo e la chiave nello spazio della DHT.

Un esempio di funzione utilizzata è in figura 2.4. I primi bit sono utilizzati per codificare la descrizione dell'attributo. Questi organizzazione, di fatto divide lo spazio degli identificatori n più 'sottospazi', ognuno dei quali gestisce un attributo. Gli altri bits sono utilizzati per memorizzare il valore, mentre alcuni sono scelti a caso. Il significato di questa scelta è quello di cercare di bilanciare il più possibile il carico nel caso un attributo abbia lo stesso valore in un numero considerevole di nodi diversi.

Quella appena descritta si riferisce alla prima fase di risoluzione di una query, quella in cui i sottoinsiemi di nodi vengono scelti in base alle loro misurazioni intra-nodo. La seconda fase nel processo di risoluzione della query consiste nell'ottenere le informazioni riguardo gli attributi *inter-nodo*. Questo tipo di attributi sono memorizzati sul nodo che effettua la misurazione e non sono esplicitamente pubblicati. Una possibile ottimizzazione suggerita, è quella di raccogliere le misurazioni inter-nodo solo su alcuni nodi detti *nodi rappresentativi*, i quali memorizzano i valori degli attributi vicini. In questo modo, fra tutti in nodi candidati, possono essere contattati solo i nodi rappresentativi. Questo, unito alla prima fase di scelta effettuata sugli attributi intra-nodo, dovrebbe garantire un numero di comunicazioni non troppo elevate.

2.7 InfoEye

InfoEye è un protocollo studiato per gestire gli attributi dinamici nelle Service Overlay Networks (SONs) [14]. L'approccio non è quindi basato specificatamente su un architettura p2p come la DHT, ma fornisce una comunque valida visione sul difficile compito delle gestione della dinamicità sulla rete.

InfoEye si basa sull'osservazione che per ogni sistema di gestione delle informazioni vi sono essenzialmente due approcci per la gestione degli attributi dinamici:

- *information push* dove la rete periodicamente informa i nodi riguardo allo stato proprie risorse
- *information pull* in cui i nodi richiedono informazioni direttamente per la risoluzione di query.

Entrambi questi approcci hanno i loro lati positivi. Ad esempio, fare *push* è conveniente quando la frequenza di arrivo delle query è alta, in modo da ammortizzare il costo dei messaggi push fra le molte query. D'altra parte fare *pull* è più efficiente quando la frequenza di arrivo delle query è bassa, poiché si raccolgono solo i dati che servono. Di conseguenza, un sistema di gestione delle informazioni efficiente, deve essere in grado di adattarsi dinamicamente alla frequenza delle query.

2.7.1 Architettura

L'architettura di InfoEye è composta da una serie di sensori, eseguiti su ogni nodo, che effettuano la misurazione degli attributi ed, in base ad uno studio statistico sulle query sottomesse al sistema, decidono dinamicamente come comportarsi.

Logicamente esistono quindi due categorie di nodi, quelli che pubblicano le risorse ed i nodi gestori, che le memorizzano ed a cui vengono inviate le query. In figura 2.5 è mostrato uno schema dell'overlay di InfoEye.

InfoEye effettua un adattamento dinamico basandosi sui dati statistici ottenuti dall'elaborazione delle query sottomesse al sistema. Più precisamente i dati mantenuti sono i seguenti:

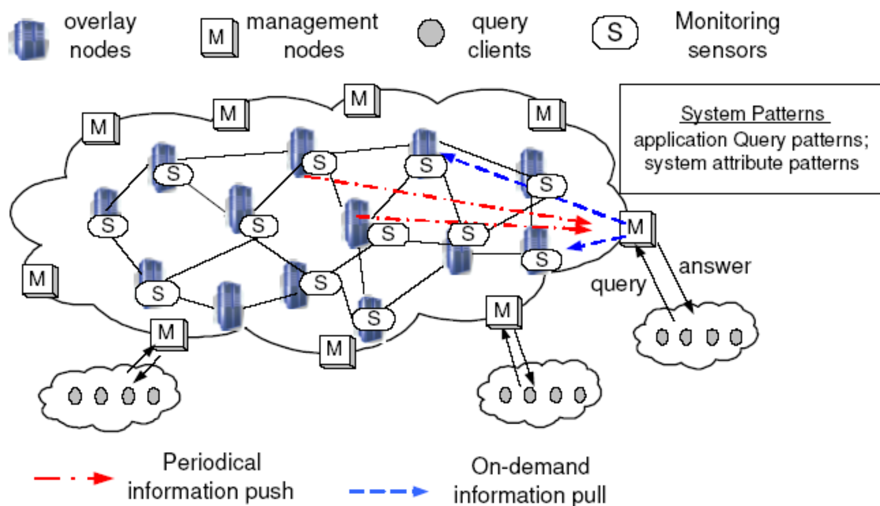

Figura 2.5: Nodi gestori(M) e sensori (S) su InfoEye.

- *Attributi maggiormente richiesti.* Sebbene una risorsa è composta da una collezione relativamente ampia di attributi, è lecito aspettarsi che solo un sottoinsieme di essi siano frequentemente richiesti dalle applicazioni. Un possibile provvedimento quindi, è quello di inviare tramite push il solo sottoinsieme di attributi più richiesti.

- *Intervalli di valori più richiesti.* Sono valide le stesse considerazioni fatte per gli attributi più richiesti. Per esempio, se la maggior parte delle query richiede una CPU con meno del 20% di carico i nodi che hanno una CPU al 50% non hanno bisogno di inviare il loro valore della CPU con una push.

- *Vincoli temporali più frequenti.* Quando un applicazione sottomette una query, può specificare un *vincolo temporale* con cui indica la massima obsolescenza che può avere il valore di un attributo. Lo studio della frequenza di questi vincoli può permettere di calibrare l'intervallo fra le push dinamicamente, in modo che sia appena sufficiente per soddisfare le query.

Dato che InfoEye utilizza sia push che pull per la raccolta delle infor-

mazioni, il modello dei costi è formato da due parti in cui si considerano rispettivamente i costi derivanti dalle operazioni di push e da quelle di pull. L'obiettivo del sistema è quello di configurare dinamicamente i sensori in modo da minimizzare il costo complessivo della pubblicazione di risorse. Per fare questo si interviene su tre parametri:

1. *Selezione degli attributi di push.* L'idea è quella di inviare tramite push i valori di un solo sottoinsieme A^* di attributi in modo da minimizzare il costo totale. Aumentando la dimensione di A^* si ha un incremento del costo di push, diminuendo A^* questo costo si riduce, ma con esso diminuisce anche la frequenza di risoluzione diretta delle query e di conseguenza si alzano i costi di pull.

2. *Valore limite per il push.* Per ogni attributo di A^* si definsce una soglia, per la quale si fa push solo se l'attributo la supera. Analogamente al primo punto, alzare il valore della soglia genera un costo minore per delle push ma aumenta quello per le pull.

3. *Intervallo di push.* Per ogni attributo è definito un intervallo di tempo che deve passare tra due push consecutive. Un intervallo maggiore riduce il costo delle push ma aumenta quello delle pull, per la stessa ragione dei punti precedenti.

2.8 Peer-to-peer Discovery of Computational Resources for Grid Applications

Questa soluzione, presentata in [1], offre un modello di resource discovery basato su DHT ed approfondisce in particolare la gestione degli attributi dinamici in questo contesto.

L'approccio è diviso in due fasi. Inizialmente è stato studiato il comportamento delle principali risorse dinamiche per poi proporre un modello basato su queste osservazioni.

Osservazioni

Sono stati effettuati rilevamenti corrispondenti a circa 3000 ore macchina su workstations di una griglia. I dati raccolti riguardano l'utilizzo di CPU, la disponibilità di memoria e lo spazio disco disponibile. L'utilizzo della CPU tende ad essere molto dinamico, con alti picchi di utilizzo ma con un carico medio del 4%, il che suggerisce che per molto tempo il processore è in uno stato di idle. Lo spazio su disco è relativamente stabile, mentre la memoria varia spesso anche se non presenta picchi accentuati. Queste rilevazioni sono utilizzate per ottimizzare la fase di aggiornamento, spiegata più in dettaglio successivamente.

2.8.1 Architettura

Come overlay di appoggio è utilizzata un'implementazione di Pastry. Il problema consiste nel mappare le risorse sulla DHT, tenendo conto che gli attributi statici di una risorsa sono discreti, mentre quelli dinamici hanno nella continuità una loro caratteristica. Il primo passo è quindi quello di rendere discreti gli attributi dinamici, utilizzando valori percentuali e rappresentadoli con un certo numero di bit. L'ID che rappresenta la risorsa è quindi in parte formato da una descrizione statica, in parte da quella dinamica.

Il dettaglio su come l'ID di una risorsa viene creato è schematizzato in figura 2.6. In questo modo il valore degli attributi statici identifica un arco sullo spazio degli indirizzi della DHT sul quale risiedono tutte le possibili configurazioni degli attributi dinamici. In altre parole una risorsa associata ad un nodo risiede sempre nello stesso arco rappresentato dagli attributi statici, ma varia la posizione al suo interno in accordo con il valore degli attributi dinamici.

Aggiornamenti

Aggiornare le risorse è un aspetto essenziale del protocollo, vista la natura dinamica delle risorse. Gli aggiornamenti sono effettuati dal nodo che pubblica la risorsa quando si verifica uno dei seguenti eventi: dopo un certo intervallo

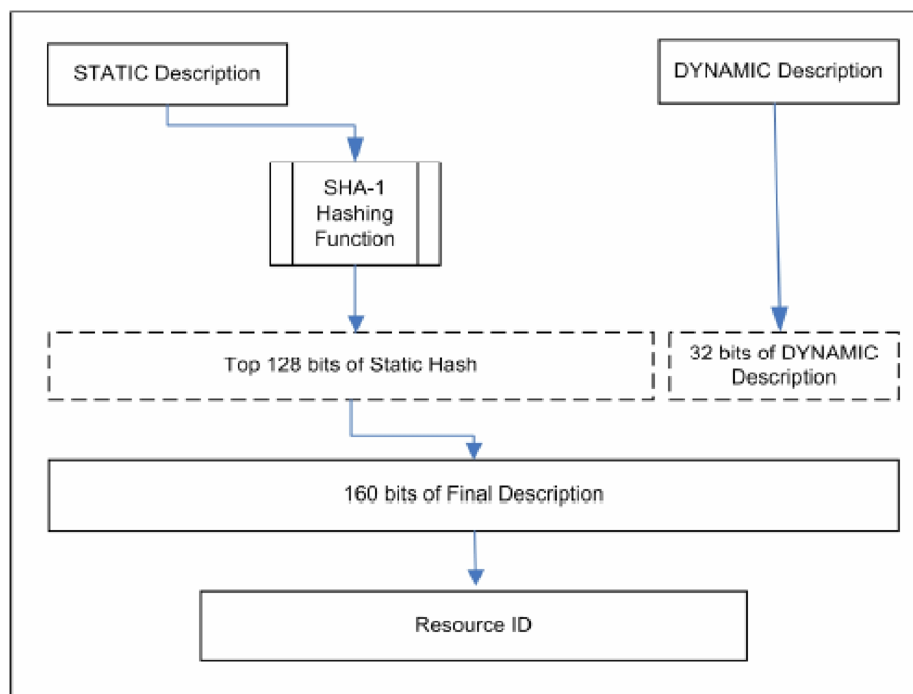

Figura 2.6: Traduzione attributi dinamici e statici in ID

prefissato, definito URATE, oppure quando si ha un cambiamento significativo dello stato della risorsa, parametrizzato nell'array UCHANGE[], che specifica la variazione che un attributo deve registrare per eseguire un aggiornamento. Ad esempio, se UCHANGE[CPU] è 3.2 allora si avrà un aggiornamento quando lo stato di idle della CPU varia di un valore corrispondente al 3,2

Risoluzione delle query

Per la risoluzione delle query sono state proposte tre strategie di ricerca:

1. *Ricerca singola.* Si identifica una risorsa desiderata, si calcola il suo ID e tramite il routing della DHT si trova il nodo che la gestisce. Questo nodo invia un messaggio di risposta all'utente che contiene tutte le risorse che memorizza le quali si avvicinano a quella cercata.

2. *Ricerca ricorsiva.* Un estensione del primo metodo, che ad ogni hop cambia i bit meno significativi dell'ID cercato (quelli relativi agli attributi dinamici), contattando di fatto più nodi per ampliare lo spettro di ricerca. La ricerca è limitata nel numero di passi da un contatore che ha funzioni di TTL.

3. *Ricerca parallela.* Si avviano più ricerche contemporaneamente per diminuire il tempo di risposta.

Capitolo 3

Remedy: Architettura generale

3.1 Introduzione

Remedy (**Re**duced **me**ssage on **d**ht modif**y**) è basato sull'approccio descritto in [3] che di fatto rappresenta un estensione di una DHT per il supporto di range query con più attributi.

In [3] il metodo con cui sono calcolati gli identificatori per le risorse utilizza un *hashing che preserva la località*. Grazie a questa proprietà è possibile risolvere range query multi attributo con un numero di passi il cui ordine è logaritmico rispetto alla dimensione della rete, indipendentemente da quanti attributi sono utilizzati per descrivere le risorse. In questo paragrafo mostriamo prima le caratteristiche principali di Chord, la DHT utilizzata in [3], quindi illustriamo il funzionamento di Remedy.

3.2 Chord

In questo paragrafo descrivo brevemente la DHT Chord [11]. Come in tutte le DHT, i dati in Chord sono composti da coppie (chiave, valore) e sono disponibili operazioni di ricerca ed inserimento.

Chord utilizza uno spazio degli identificatori unidimensionale ad *anello* con modulo 2^m, dove m è il numero di bit utilizzati per rappresentare gli identificatori. Ogni nodo in Chord è associato ad un identificatore unico

ID, ricavato tramite una funzione di hashing (tipicamente la SHA1) a cui sono passati parametri quali, ad esempio, l'indirizzo IP del nodo e porta del servizio DHT. Come i nodi, anche gli oggetti hanno associato il proprio ID, la *chiave* dell'oggetto. Una generica chiave k è assegnata al primo nodo il cui identificatore è uguale o segue k nello spazio degli indirizzi. Questo nodo è chiamato il *successore* di k ed è rappresentato come $successor(k)$.

La figura 3.1 mostra una rete Chord con 8 nodi e uno spazio degli indirizzi di 2^6 elementi. Il nodo $N20$ ha come ID 20 e gestisce gli oggetti con chiave 10 e chiave 15.

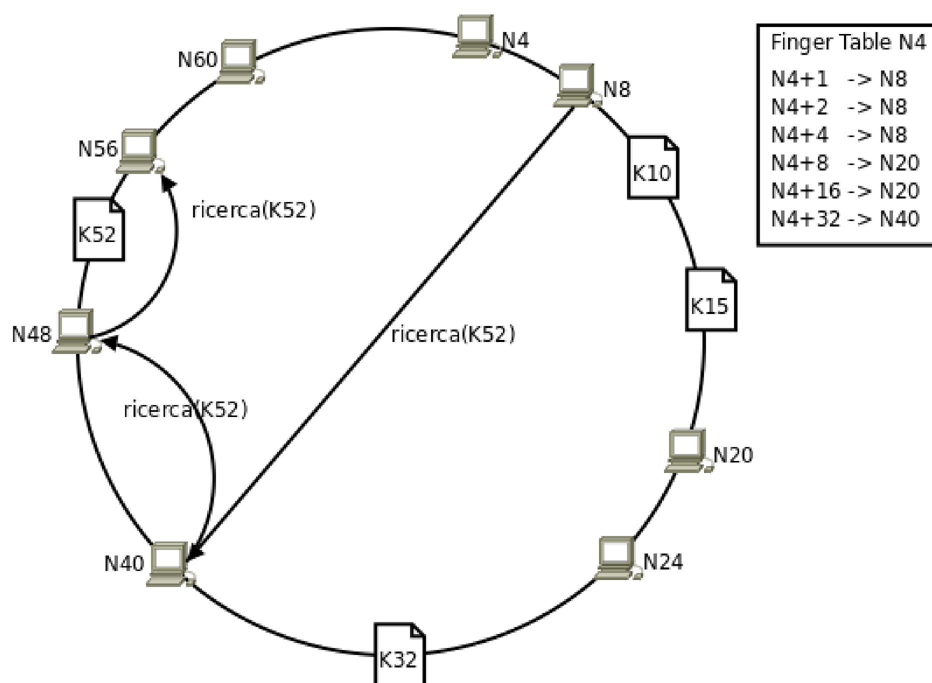

Figura 3.1: Una rete Chord con identificatori di 6 bit.

Ogni nodo in Chord mantiene due insiemi di vicini, la *successor list* e la *finger table*. I nodi nella successor list sono quelli che immediatamente seguono il nodo sull'anello, mentre i nodi nella finger table sono distribuiti esponenzialmente nello spazio degli identificatori.

La finger table può contenere al massimo m elementi. Per un nodo con ID n l'i-esimo elemento della finger table contiene il riferimento al primo nodo, il quale supera n di almeno 2^{i-1} sull'anello. In Chord, questo nodo è chiamato

l'i-esimo *finger* del nodo n ed è rappresentato da $n.finger[i]$. Il primo finger è l'immediato successore di n (i = 1). Ogni nodo quindi mantiene solo $O(logN)$ vicini in una rete Chord con N nodi. Ad esempio, in figura 3.1 i fingers di $N4$ sono $N8$, $N20$ e $N40$.

Quando un nodo n vuole cercare un oggetto con chiave k deve inviare una richiesta al nodo x successore di k, $x = successor(k)$. L'algoritmo di routing funziona in questo modo: data una richiesta per la chiave k, il nodo cerca nella sua successor list il successore di k e vi inoltra la richiesta. Se non è presente, inoltra la richiesta al nodo j il cui ID è il maggiore tra quelli che precedono k nella finger table. Ripetendo questo processo, la richiesta si avvicina ad ogni passo al successore di k. Infine x finalmente riceve la richiesta, recupera i dati localmente ed invia la risposta al nodo n.

Per esempio se $N8$ in figura 3.1 inizia una ricerca per la chiave $K52$, invia la richiesta al suo finger $N40$, quello più vicino a $K52$ nello spazio degli identificatori. $N40$ inoltra la richiesta a $N48$ che, a sua volta, la invia a $N56$. Dato che $N56$ è il nodo successore di $K52$, raccoglie il dato localmente ed invia il risultato a $N4$.

Dato che i fingers sono distanziati esponenzialmente nella finger table, ogni *hop* (salto) dal nodo n al successivo copre almeno metà della distanza tra n e k. Il numero medio di hop per una ricerca è quindi $O(logN)$, dove N è il numero di nodi connessi alla rete.

3.3 Funzioni Hash

In una DHT, lo scopo di una funzione hash è quello di disperdere il più possibile i dati in modo da favorire taluni aspetti, tra cui il bilanciamento del carico. Questa proprietà però, perde di efficacia quando è necessaria la risoluzione di range query, in quanto distrugge ogni informazione sulla località dei dati. La località, per la risoluzione di range query è importante in quanto risorse vicine in termini di valori lo sono anche nello spazio degli identificatori creato dalla funzione hash.

Una funzione H preserva la località se verifica le seguenti proprietà:

- $H(v_i) < H(v_j)$ sse $v_i < v_j$
- se un intervallo $[v_i, v_j]$ è diviso in $[v_i, v_k]$ e $[v_k, v_j]$, allora il corrispondente intervallo $[H(v_i), H(v_j)]$ deve essere diviso in $[H(v_i), H(v_k)]$ e $[H(v_k), H(v_j)]$

Supponiamo di avere un attributo a che può assumere valori numerici nell'intervallo $[v_{min}, v_{max}]$. Una semplice funzione hashing che preservi la località è la seguente:

Esempio 1.

$$H(v) = (v - v_{min}) \times (2^m - 1)/(v_{max} - v_{min})$$

Questa funzione è molto semplice ma non tiene conto dell'effettiva distribuzione dei dati e può causare sbilanciamenti del carico molto marcati.

Si assuma che ogni risorsa sia caratterizzata da M attributi $a_1, .., a_M$ di valore $v_1, .., v_M$. Il valore v_i di ogni attributo a_i, è contenuto nell'intervallo $[v_{imin}, v_{imax}]$ secondo una certa distribuzione che indichiamo con $D_i(v)$, la quale deve essere continua monotona crescente e nota a priori. Per ogni attributo, quindi è possibile, utilizzando D, generare una funzione di hashing che preservi la località del tipo

Esempio 2.

$$H_i(v) = D_i(v) \times (2^m - 1)$$

che fornisce uno strumento mappare un valore qualsiasi di un attributo nello spazio di Chord. Da notare che questa funzione è più specifica rispetto a quella, comunque sempre valida, enunciata nel paragrafo precedente in quanto tiene conto dell'effettiva distribuzione degli attributi.

Si consideri un attributo i cui valori sono distribuiti in modo non uniforme nell'intervallo di valori $[0, 10]$ come da figura 3.2 (diagramma a).

Una possibile funzione D può essere costruita a partire dalla probabilità che un valore v sia maggiore rispetto ad uno casuale x. In particolare

$$D(v) = P(x <= v)$$

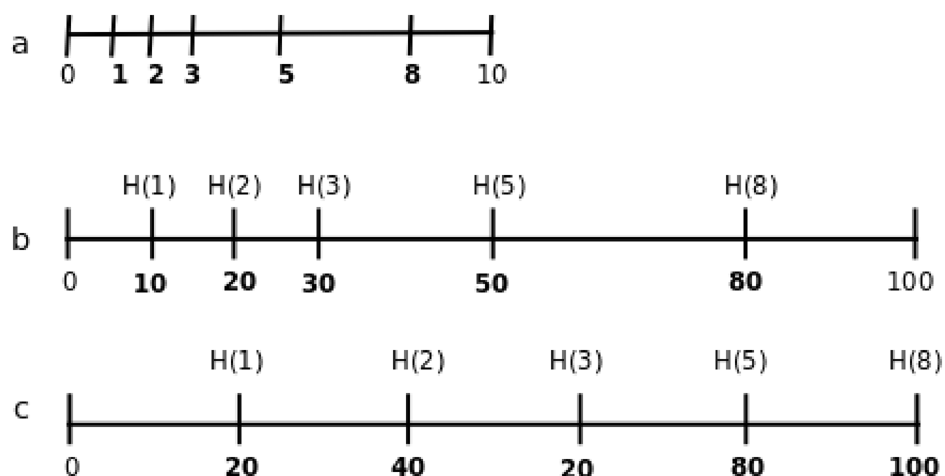

Figura 3.2: Hashing in Remedy

da cui, ad esempio, abbiamo:

$$D(3) = \frac{3}{5} D(8) = \frac{5}{5}$$

Supponiamo inoltre di disporre di uno spazio degli identificatori rappresentato dall'intervallo I $[0, 100]$ (per semplificare la presentazione l'estremo superiore dell'intervallo non è una potenza di due). Considerando $|I|$ come l'ampiezza dell'intervallo ed utilizzando la funzione di distribuzione D appena calcolata, la funzione hash diventa:

Esempio 3.

$$H(v) = D(v) \times |I|$$

Questa funzione distribuisce i valori *in modo uniforme* sull'intervallo I nel modo rappresentato in figura 3.2 (diagramma c). Se avessimo utilizzato la funzione dell'esempio 1 per mappare gli stessi dati otteniamo la distribuzione del diagramma b, che come si vede non distribuisce i dati in modo efficace.

Un problema che riguarda le funzioni appena descritte è il rapporto che esiste tra l'ampiezza degli intervallo I dei possibili valori per un attributo ed il numero di nodi presenti nella rete, N. Utilizzando H per mappare i valori degli attributi nello spazio degli identificatori, quando I è molto inferiore rispetto ad N, i valori saranno mappati sempre sugli stessi nodi e non sempre valori contigui saranno mappati su nodo contigui.

Ad esempio, supponiamo di utilizzare H dell'esempio 1 per l'hashing di un attributo a con valori percentuali. In questo caso il dominio è rappresentato dall'intervallo $[0, 100]$ e di conseguenza $I = 100$. Se $N = 200$, allora un nodo ogni due non memorizza mai la risorsa indicizzata da a. Come detto prima questa situazione ha due conseguenze: un generale sbilanciamento del carico e risorse con valori 'vicini' non sono memorizzate su nodi vicini nello spazio degli identificatori. La seconda conseguenza potrebbe avere impatto anche sulla risoluzione delle query poiché si interrogano nodi che non hanno informazioni utili. In realtà per come funziona Remedy il numero di nodi da visitare nella fase di risoluzione di una query non dipende da questo fattore.

Tutti questi problemi sono affrontati in [10]. La sua funzione di hash infatti divide lo spazio degli identificatori assegnando una parte ad un solo attributo. Questo riduce la probabilità che si creino dei 'vuoti' nella memorizzazione dei valori. Inoltre la parte meno significativa di una generica chiave è composta da valori casuali. Ciò permette di avere sempre una diversificazione anche quando l'insieme I è molto piccolo.

3.4 Remedy: l'architettura generale

Come molte altre DHT, Chord mette a disposizione un metodo efficente e scalabile per la memorizzazione e la ricerca di dati singoli. Tuttavia, non supporta le *range queries* e le ricerche multi attributo. L'approccio [3] mira ad estendere Chord per risolvere questi aspetti.

Per assegnare gli identificatori Chord utilizza un algoritmo di hashing (SHA1) che genera chiavi distribuite in tutto lo spazio degli indirizzi. Tuttavia questo algoritmo distrugge la *località* delle chiavi rendendo impossibile un approccio più complesso per affrontare le range query.

Remedy, come MAAN, utilizza una funzione di hashing che preserva la *località* per assegnare identificatori ai valori degli attributi. La funzione SHA1 è utilizzata comunque per generare gli ID dei nodi e per calcolare gli identificatori degli attributi di tipo stringa.

Usando una funzione H come quella dell'esempio 1 per mappare i valori negli attributi nello spazio degli indirizzi, data una query $[l, u]$ con l e u rispettivamente il limite inferiore e quello superiore, gli attributi i cui valori cadono nell'intervallo specificato hanno un identificatore maggiore o uguale di $H(l)$ e minore o uguale a $H(u)$. Come si può notare il fattore $(2^m - 1)/(v_{max} - v_{min})$ indica il rapporto tra la dimensione dell'intervallo considerato e la dimensione dello spazio degli identificatori ed è utilizzato per mappare uniformemente i valori nell'intervallo $[v_{min}, v_{max}]$ all'interno della DHT. È importante notare che se i valori reali degli attributi non sono uniformemente distribuiti nell'intervallo $[v_{min}, v_{max}]$ non lo sono neppure nella DHT.

Questa proprietà di località è sostanziale nella fase di risoluzione delle query.

3.4.1 Registrazione

La risoluzione di query con più attributi, complica la fase di registrazione di una risorsa rispetto ad una DHT standard.

Ogni nodo registra la proprie informazioni al nodo $n_i = successor(H(v_i))$ per ogni attributo a_i, in cui H rappresenta la funzione hash utilizzata. Se si conoscono le distribuzioni dei vari attributi è possibile usare la funzione descritta nel paragrafo precedente all'esempio 2.

Inoltre, Ogni nodo mantiene le coppie (valore-attributo, info-risorsa) *indicizzate* per attributo. Quando un nodo riceve una richiesta di registrazione da parte di x con il valore dell'attributo $a_i = v_{ix}$ e le informazioni sulla risorsa r_x, aggiunge la coppia (v_{ix}, r_x) alla lista indicizzata dall'attributo a_i. Come si vede in figura 3.3 ogni nodo mantiene una struttura indicizzata per attributo, nell'esempio dagli attributi identificati con cpu e memoria.

Durante la fase di ricerca la query è composta da più sotto queries, che specificano gli intervalli desiderati per ogni attributo. Esistono due approcci

per la risoluzione delle query multi attributo, quello *iterativo* e quello tramite *attributo dominante*. Entrambi questi approcci si basano sulla risoluzione di range query con singolo attributo.

3.4.2 Risoluzione di range query con singolo attributo

Supponiamo un nodo n voglia cercare risorse il cui attributo a ha un valore v che è compreso fra l e u. Il nodo utilizza il routing di Chord per inoltrare la richiesta a n_l il successore di $H(l)$. La richiesta contiene vari parametri: la chiave k usata per il routing (inizialmente $k = H(l)$), il range R dell'attributo desiderato e la lista X dei risultati (inizialmente vuota). Quando il nodo n_l riceve la richiesta, controlla se fra le sue risorse locali qualcuna soddisfa la query ed eventualmente la aggiunge ad X. Questo passo è necessario, in quanto il nodo n_l, essendo il primo ad essere contattato, può contenere risorse il cui valore sia minore di l. Lo stesso tipo di ragionamento è applicato all'ultimo nodo contattato.

Successivamente, verifica se egli stesso è il successore di $H(u)$. In caso affermativo invia i risultati al nodo n altrimenti inoltra la richiesta al suo immediato successore n_{l+1}. Questo processo si ripete finché la richiesta non raggiunge n_u.

La fase di routing verso n_l costa $O(logN)$ hops, la fase di inoltro della richiesta da n_l a n_u costa $O(K)$, dove K è il numero di nodi presenti fra n_l e n_u. In totale quindi servono $O(logN + K)$ hops per risolvere una query con singolo attributo.

3.4.3 Risoluzione di range query multiattributo: metodo iterativo

L'approccio iterativo è immediato. Se un nodo n cerca risorse tramite una query da M attributi, n risolve iterativamente ognuna delle sotto query ed *interseca* i risultati *localmente*. L'algoritmo utilizzato per ognuna delle sotto query è essenzialmente lo stesso utilizzato per la risoluzione di una query con singolo attributo. Sebbene questo approccio sia semplice da implemen-

tare non è molto efficiente, poiché necessita di un numero di hops pari a $\sum_{i=1}^{M}(logN + K_i)$ per risolvere una query.

Ad esempio, supponiamo che n inizi una query composta da due attributi a_1 e a_2. In questo caso la query è composta da due sotto-queries che verranno risolte separatamente. Inizialmente si risolve la prima delle due sotto-query, quella dell'attributo a_1, come se fosse una query a se stante. Il risultato è un insieme di risorse che indichiamo con R_1. Successivamente itero lo stesso procedimento per la seconda sotto-query, ottenendo l'insieme R_2. A questo punto, l'insieme dei dati che soddisfano l'intera query originale e dato dall'intersezione di R_1 con R_2.

3.4.4 Risoluzione con attributo dominante

Con l'approccio iterativo si risolve ad ogni passo una sottoquery, il cui insieme dei risultati x_i è un sottoinsieme dei risultati della query X. Dato che, per ogni attributo, si registrano i valori di tutti gli attributi della risorsa, i nodi che contengo le risorse per soddisfare x_i contengono anche le informazioni sugli altri attributi.

L'approccio con attributo dominante può utilizzare le informazioni supplementari per risolvere la query controllando solo l'insieme di candidati x_k che soddisfano la sottoquery dell'attributo a_k. Chiameremo quindi a_k l'attributo *dominante* di una query.

Tutta la query (composta da più sottoqueries) è trasportata nel messaggio di richiesta, ogni nodo controlla localmente se ha risorse che la soddisfano. Quando un nodo n inizia una ricerca, sceglie un attributo ed inoltra la richiesta a $n_l = successor(H(l))$. L'algoritmo è simile a quello visto per la risoluzione con attributo singolo e si ferma una volta raggiunto $H(u)$, ma localmente ogni nodo aggiunge alla lista dei risultati solo quelle risorse che soddisfano *tutta* la query, cioè quelle risorse per cui tutti gli attributi soddisfano i vincoli imposti dalla query.

Dato che questo approccio effettua una sola iterazione attraverso l'anello, utilizza $(logN + N \times s_k)$ hops per risolvere la query, dove con s_k è indicata la *selettività* dell'attributo scelto. Si definisce *selettività*, rispetto ad un at-

tributo, il rapporto tra la dimensione dell'intervallo specificato nella query e la dimensione dello spazio degli identificatori. La selettività del generico attributo i è calcolata come:

$$s_i = \frac{H(v_{iu}) - H(v_{il})}{2^m}$$

Per minimizzare il numero di nodi contattati è quindi necessario scegliere come attributo dominante quello con selettività minore.

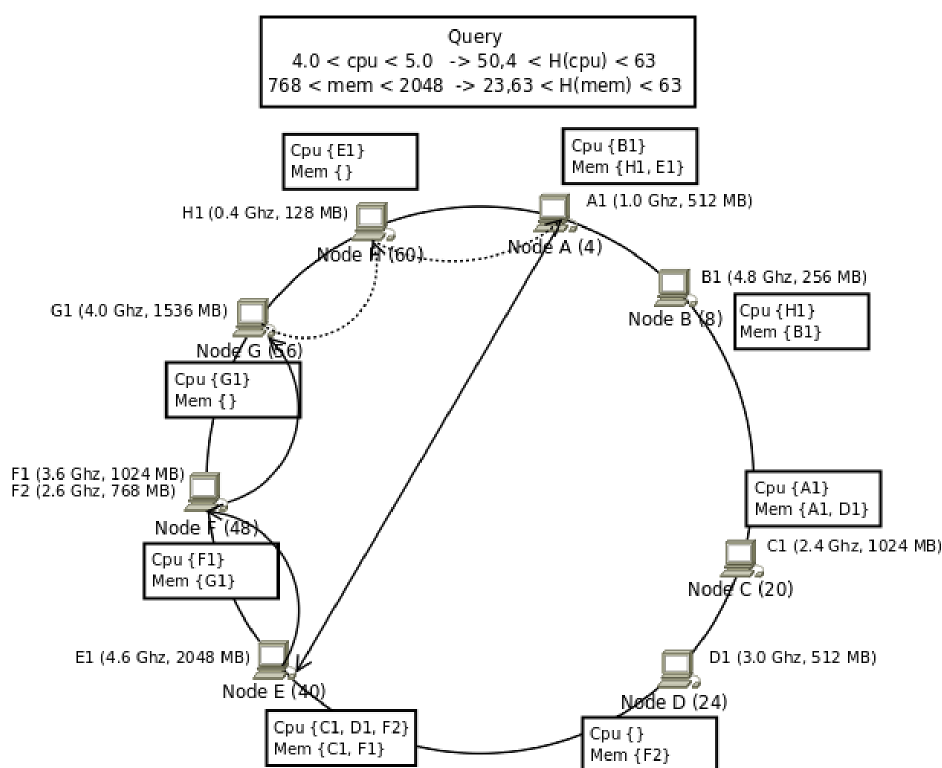

Figura 3.3: Un esempio di risoluzione di query utilizzando l'attributo dominante.

La figura 3.3 mostra un esempio di query risolta con attributo dominate in una rete con 8 nodi e in cui sono memorizzate 9 risorse. Ogni risorsa ha due attributi, velocità della cpu (Cpu) e quantità di memoria (Mem), ed un nodo può pubblicare una o più risorse. Il nodo A cerca risorse con Cpu che varia da 4.0 a 5.0 Ghz, e Mem da 768 MB a 1024MB. Per prima cosa, usando H, associa questi intervalli ad altri corrispondenti nello spazio degli

identificatori e successivamente sceglie l'attributo con selettività minore che, nel nostro caso, è Cpu. Il nodo G viene raggiunto tramite il routing di Chord, la richiesta viene poi successivamente inoltrata ad H ed infine ad A (linee tratteggiate) dove si conclude l'operazione in quanto A è anche il nodo che ha avviato la ricerca.

In conclusione l'architettura proposta presenta performance di routing *indipendenti dal numero degli attributi* e garantisce scalabilità per le query multi attributo. I possibili overheads si identificano nella quantità di memoria per memorizzare tutta la risorsa e, aspetto più rilevante, nell'alto costo di aggiornamento nel caso in cui il valore di un attributo cambi.

Nel paragrafo successivo mostreremo un insieme di ottimizzazioni il cui scopo è diminuire l'overhead relativo alla gestione degli attributi dinamici.

3.5 Remedy: pubblicazione di attributi dinamici

L'utilizzo dell'approccio mostrato nel paragrafo 3.4 per la gestione della fase di pubblicazione (aggiornamento) di una risorsa è molto dispendioso in termini di messaggi scambiati fra i nodi. Il traffico di rete generato dagli aggiornamenti delle risorse può interferire su quello generato dalle query, allungando di fatto il tempo medio di risposta delle query stesse.

3.5.1 Definizione del problema

In diverse DHT la pubblicazione di un dato può essere divisa in due fasi distinte. Alla fase di routing (che ha costo logaritmico rispetto alla dimensione della rete) segue l'invio del messaggio di PUT vero e proprio e l'invio della relativa risposta. Come vedremo nel capitolo 4 Overlay Weaver implementa la fase di pubblicazione secondo questa strategia. In generale può essere neces-

sario un piccolo sforzo aggiuntivo per fare in modo che sia possibile mandare un messaggio direttamente ad un nodo conoscendone l'indirizzo, senza utilizzare il routing della DHT. In altre parole, riferendosi alla PUT, dovrebbe essere presente una funzionalità di *put(IP)*, leggermente differente da quella generale di una DHT, *put(ID)* ma comunque di facile implementazione.

Successivamente indicheremo con MPA (Messaggi Per Aggiornamento) la quantità di messaggi scambiati da un nodo per la fase di aggiornamento, considerando l'unità *messaggio* a livello del protocollo di trasporto (TCP o UDP). Il nodo che pubblica una risorsa sarà indicato come *provider*.

Se indichiamo con n il numero di nodi presenti nella rete e con K il numero di attributi che rappresentano una risorsa, una prima stima dei messaggi scambiati per la pubblicazione di una singola risorsa è la seguente:

$$MPA_{base} = (P + lg(n)) \times K$$

dove P indica il numero di messaggi scambiati per la PUT nel nostro caso P=2).

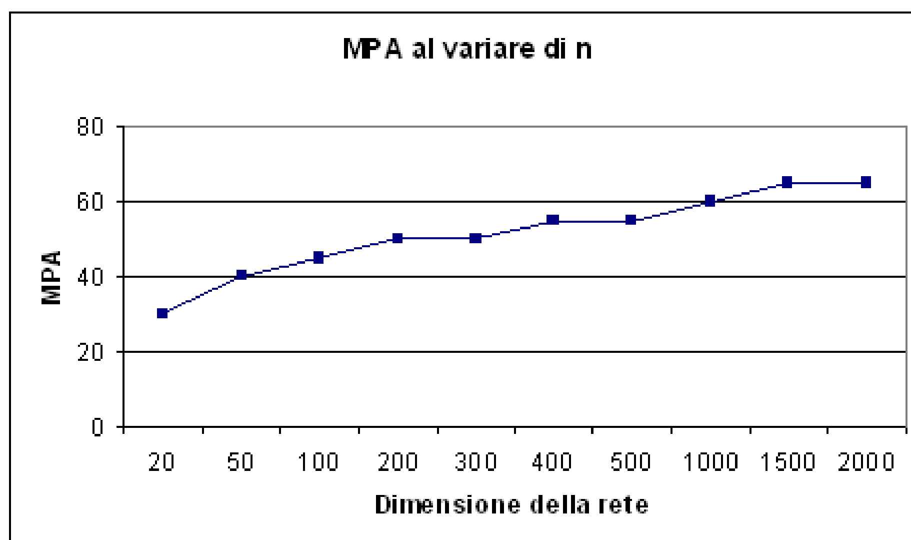

Figura 3.4: MPA al variare della dimensione della rete, con K = 5.

In figura 3.4 è mostrato l'andamento degli MPA al crescere della rete, in figura 3.5 al crescere del numero di attributi.

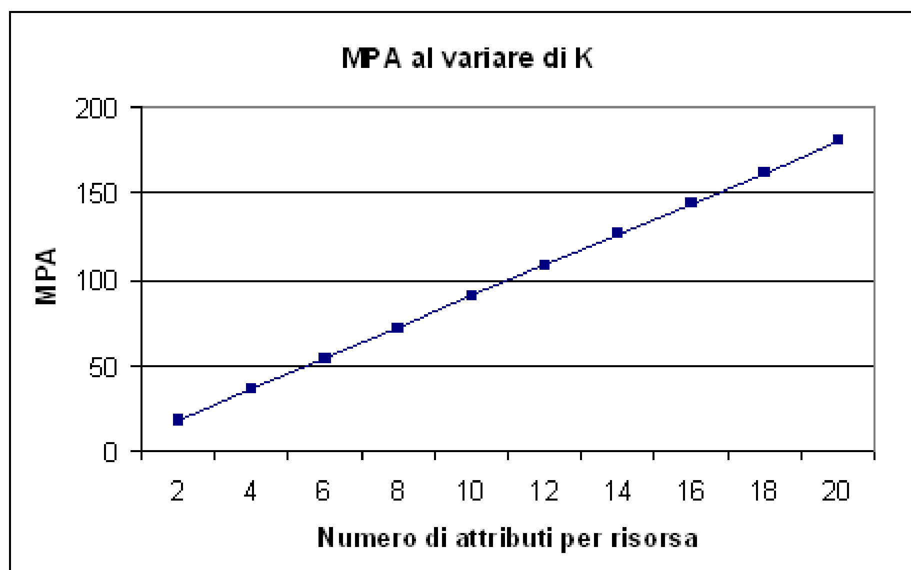

Figura 3.5: MPA al variare del numero di attributi, con una rete di 100 nodi.

L'andamento del numero di MPA è logaritmico per quanto riguarda la dimensione della rete, mentre diventa lineare con il numero degli attributi. L'approccio basato sul paragrafo precedente si rivela quindi scalabile sul numero degli attributi per quanto riguarda la risoluzione delle query mentre non risulta scalabile per quanto riguarda la pubblicazione delle risorse.

Diventa quindi importante limitare quanto più possibile il traffico derivante dalla pubblicazione delle risorse riducendo il numero di MPA, per rendere un approccio di questo tipo applicabile ad una rete reale.

La soluzione più immediata consiste nel rappresentare le risorse con il minor numero di attributi. Se da un lato questo può essere possibile in alcuni ambiti con una progettazione del sistema più accurata, non è certo una soluzione generale. L'intervento deve essere quindi relativo a minimizzare il numero di messaggi che vengono inviati per attributo.

3.5.2 Ottimizzazione del routing

La soluzione proposta si articola in due passi che, almeno inizialmente, possono essere analizzati in modo indipendente.

Il primo passo è mirato all'eliminazione, dove possibile, dei messaggi necessari per il routing eseguito in fase di pubblicazione di una risorsa. Il presupposto da cui partire è che gli attributi, pur avendo una componente di *dinamicità elevata*, abbiano una possibilità relativamente alta di avere lo stesso valore in due misurazioni successive.

È presumibile, ad esempio, che il carico di un *processore* resti *nullo* per *periodi di tempo relativamente lunghi* quando si trova in uno stato di idle oppure che l'insieme di applicazioni in esecuzione non richieda l'allocazione di memoria aggiuntiva per un periodo di tempo relativamente lungo.

Queste considerazioni sono alla base di un meccanismo per l'ottimizzazione del traffico relativo alla pubblicazione delle risorse. Consideriamo infatti la $i$esima pubblicazione effettuata da un nodo per la risorsa R e relativa all'attributo A. Se il valore di A non è cambiato o la differenza con il valore precedente di A risulta inferiore ad un valore soglia, è possibile che il nodo D, destinatario della pubblicazione, sia il solito della $i-1$esima pubblicazione. In queste condizioni è possibile eliminare i messaggi relativi alla fase di routing, procedendo direttamente all'invio del messaggio di PUT. Si noti che, anche se il valore dell'attributo A è rimasto identico, è possibile che il valore di qualche altro attributo sia cariato, rendendo comunque necessaria la PUT:

Si noti inoltre che, per come è strutturata una DHT, in particolare Chord, non è necessario che il valore sia rimasto invariato ma che la differenza fra le due misurazioni sia tale che il nodo destinatario della pubblicazione non cambi.

Nei paragrafi successivi, quando si afferma che una risorsa ha valori uguali in due misurazioni successive, intenderemo in realtà che i valori sono abbastanza vicini da non modificare il nodo destinatario delle pubblicazione. Inoltre, in seguito indicheremo una pubblicazione che non richiede la fase di routing come *diretta*.

Quella appena descritta è comunque una condizione solamente necessaria per l'utilizzo della pubblicazione diretta. Può succedere infatti che, nell'intervallo fra le due misurazioni successive, la topologia dell'overlay cambi e di conseguenza la stessa risorsa venga gestita da un nodo differente, in se-

guito ad una redistribuzione dei dati sulla rete dovuta all'inserimento o alla disconnessione dei nodi. Questo dipende dal fenomeno del *churn*, tipico delle reti peer to peer. In questo caso, la pubblicazione diretta non è attuabile ed è necessario eseguire la fase di routing.

Diventa chiaro quindi che l'eliminazione della fase di routing dipende dal verificarsi di due condizioni indipendenti. Indicando con x la probabilità che due misurazioni successive di un attributo siano uguali, con y la probabilità che la PUT non sia influenzata dal churn e con P il numero di messaggi scambiati nella fase di PUT, abbiamo che:

- con probabilità $x \times y$ si inviano P messaggi, la pubblicazione diretta.
- con probabilità $x \times (1 - y)$ si inviano $2P + lg(n)$ messaggi, quelli richiesti nella fase di routing e le due PUT.
- con probabilità $(1 - x)$ si inviano $lg(n) + P$ messaggi, quelli richiesti nella fase di routing più la PUT.

Da queste considerazioni la formula ottenuta per calcolare il numero di MPA è:

$$MPA_1 = (P + lg(n) - (P + lg(n))xy + Px) \times K$$

Se poniamo $M_1 = (P + lg(n))xy - Px$, il numero di MPA è ridotto di un fattore M_1 per ogni attributo rispetto al caso base. Nell'espressione di M_1 si nota come per poter eliminare il costo del routing è necessario che si verifichino entrambe le condizioni. L'altro fattore (Px) rappresenta i due messaggi necessari per fare la PUT nel caso che il nodo gestore della risorsa rimanga (a meno di effetti del churn) invariato.

In figura 3.6 è presente una valutazione dell'andamento del numero di MPA al variare di x. Si nota che una relativa stabilità nei valori degli attributi ha significato solo se è accompagnata da un fattore churn relativamente basso.

Avendo risorse rappresentate da soli attributi statici ($x = 1$) in una rete con un livello di churn molto marcato ($y = 0$) porta ad un peggioramento rispetto al caso base. Il costo maggiore è dato dal tentativo costante di utilizza-

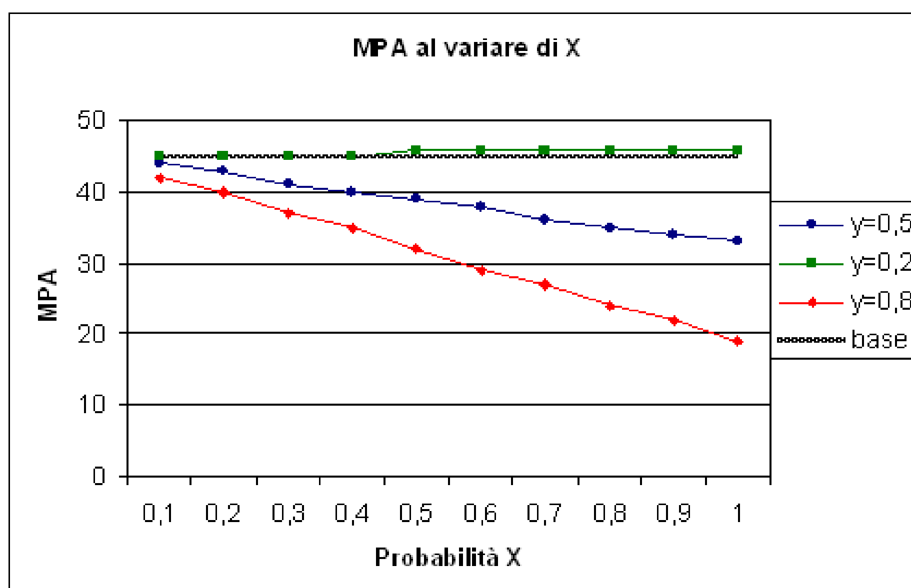

Figura 3.6: MPA al variare di x, in una rete di 100 nodi e con K=5.

re la pubblicazione diretta che fallisce, a causa della costante redistribuzione dei nodi sulla rete.

Osservando entrambe le figure 3.6 e 3.7 è evidente come il caso migliore sia quello in cui x e y abbiano valori alti. In una rete di 100 nodi come quella di figura 3.7 si ha una diminuzione di più di 20 MPA, circa la metà del totale.

3.5.3 Diminuzione del numero di pubblicazioni

Il secondo passo sfrutta l'implementazione delle range query in Remedy, in particolare il fatto che le risorse che soddisfano una query risiedono su un solo arco dell'anello che compone lo spazio degli identificatori della DHT. Questo arco è quello descritto dall'attributo che ha la selettività maggiore (che diminuisce la dimensione dell'appena citato arco) ed è detto *attributo dominante*.

La considerazione di base è che possono esistere attributi che sono scelti con più frequenza dominanti rispetto ad altri. Se per ipotesi consideriamo il caso limite in cui le query siano effettuate utilizzando sempre lo stesso attributo dominate, le risorse indicizzate dagli altri attributi non verrebbero

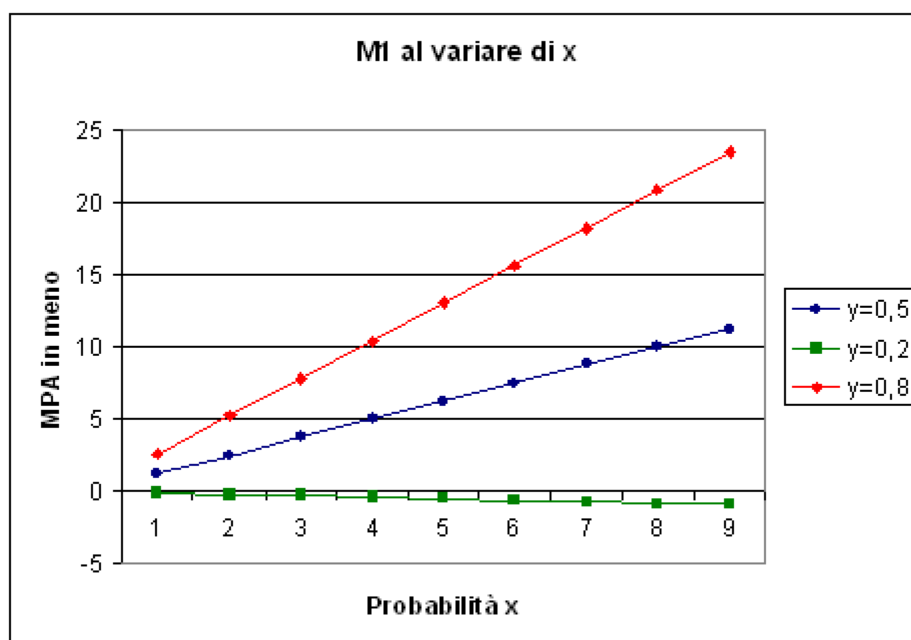

Figura 3.7: M_1 al variare di x, in una rete di 100 nodi e con K=5.

mai utilizzate nella fase di risoluzione delle query rendendo di conseguenza il loro aggiornamento evitabile.

Ovviamente questa ipotesi non può essere aderente ad una situazione reale, ma esistono sicuramente attributi che in certe porzioni di arco sono utilizzati più spesso per risolvere le query, in altre parole alcuni attributi possono definirsi *più popolari* rispetto ad altri. La popolarità è quindi la frequenza con cui un attributo viene scelto come dominate. D'altra parte questo fattore può variare nel tempo in base alle query sottomesse, da cui deriva una necessità di rendere la soluzione più dinamica possibile.

I nodi provider possono sfruttare la conoscenza della popolarità limitandosi ad effettuare gli aggiornamenti più significativi. Intuitivamente infatti, se un attributo ha popolarità bassa la risorsa che indicizza può essere aggiornata con frequenza minore.

Resta da definire come viene calcolata la popolarità, se in base a tutte le query sottomesse al sistema oppure considerando le query localmente risolte da ogni nodo. La scelta di un approccio rispetto all'altro modifica il metodo con cui i nodi provider ottengono le informazioni sulla popolarità.

Soluzione centralizzata

Con questo modello, la popolarità di un attributo viene calcolata prendendo in esame tutte le query che sono sottomesse alla DHT. È necessario quindi un punto di centralizzazione (supponiamo si tratti di un nodo della DHT) che si occupi di effettuare il calcolo. Esiste un solo valore di popolarità per ogni attributo. Periodicamente i nodi provider interrogano questo nodo, tramite una GET, per conoscere la popolarità degli attributi.

Questo approccio soffre di tutti i problemi tipici che presentano le soluzioni centralizzate che riguardano la scarsa scalabilità al crescere della rete e il problema del *single point of failure*. Anche se non si considerano questi problemi ve ne sono altri legati all'algoritmo. In primo luogo per mantenere le informazioni distribuite su tutti i provider sono necessari messaggi supplementari, che possono mitigare l'effetto benefico dell'ottimizzazione.

Inoltre le informazioni globali sulla popolarità hanno poca utilità in quanto il concetto di popolarità è prettamente locale. Ad esempio consideriamo un generico attributo A. Una situazione plausibile è che la maggior parte delle query richiedano più spesso valori di A presenti in un range, gestito da un insieme di nodi N. Come conseguenza, A è molto popolare per i nodi presenti in N mentre per gli altri nodi la popolarità è più bassa. Se si considera la popolarità come globale, ad esempio una media, i nodi in N avranno una popolarità più bassa di quella reale, gli altri una più alta annullando parzialmente l'effetto delle ottimizzazioni.

Soluzione decentralizzata

In questo approccio ogni nodo mantiene una propria stima sulla popolarità degli attributi. A differenza della visione globale, la popolarità di un attributo può variare da nodo a nodo. Dato che le informazioni sono locali, il nodo provider le ottiene direttamente dal nodo su cui la risorsa deve essere pubblicata. Si può evitare di introdurre un nuovo messaggio, utilizzando come veicolo il messaggio di risposta della PUT.

Se ad una risorsa corrisponde lo stesso nodo gestore alla PUT successiva, l'informazione ottenuta precedentemente è utilizzabile. Questo approccio è

completamente distribuito e non immette ulteriori messaggi sulla rete. Inoltre le informazioni utilizzate dal provider sono più accurate poiché locali, rispetto a quelle calcolate globalmente.

Ad esempio supponiamo che in una rete con 100 nodi e in cui le risorse sono rappresentate da due attributi, si abbiano due distinti insiemi di nodi; il primo insieme N_1 composto da 30 nodi in cui la popolarità dell'attributo A sia uguale all'80%, il secondo di 70 nodi N_2 in cui la popolarità di A sia uguale al 10%.

Con il metodo centralizzato le risorse dei nodi appartenenti a N_1 sarebbero meno aggiornate rispetto al necessario, in quanto globalmente la popolarità è più bassa. D'altra parte le risorse dell'insieme N_2 lo sarebbero troppo spesso, in quanto globalmente la popolarità di A è più alta della loro. Con il sistema distribuito questo problema non esiste, in quanto ogni nodo fa una valutazione locale, e quindi esatta, della propria popolarità.

3.5.4 Utilizzo della popolarità

L'approccio distribuito è decisamente il più vantaggioso. Inoltre il fatto di utilizzare un messaggio già esistente per la comunicazione delle popolarità relative ad un nodo mette in correlazione i due punti della soluzione. Il fatto che un attributo abbia lo stesso valore in due misurazioni successive, non solo permette di evitare il routing, ma permette anche di contare sulla conoscenza acquisita dal provider, che potrebbe permettere di eliminare un certo numero di messaggi e ridurre di conseguenza gli MPA.

Avendo indicato con x la probabilità di effettuare due PUT successive sullo stesso nodo, indichiamo con j quella di poter evitare la PUT per un qualsiasi attributo. Dato che la fase di pubblicazione può essere evitata solo quando sono vere entrambe le condizioni, si indica la proprietà composta come $z = x \times j$.

Dal punto vista analitico abbiamo che:

$$MPA_2 = MPA_1 \times (1 - z)$$

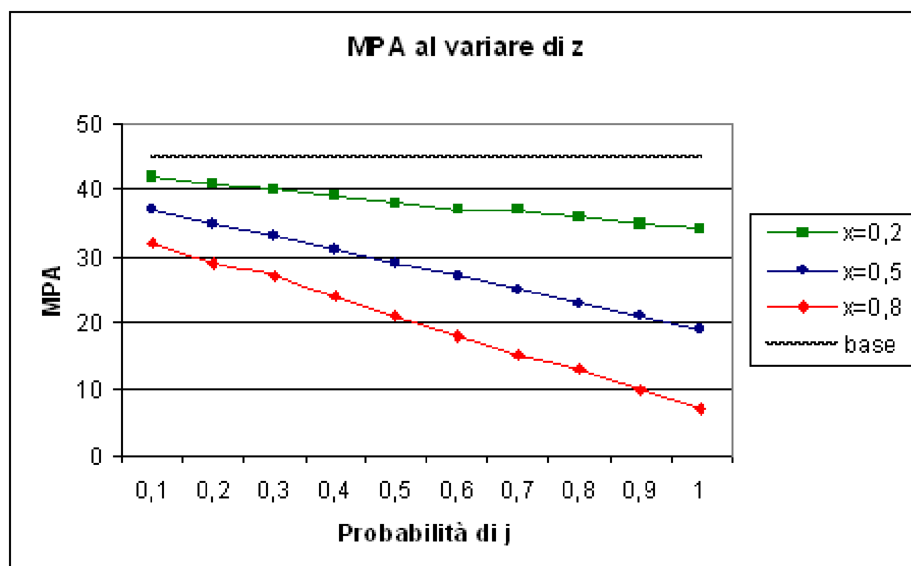

Figura 3.8: M_2 al variare di z, in una rete di 100 nodi e con K=5.

In figura 3.8 vediamo l'andamento degli in MPA con varie combinazioni di x e j in una rete a medio livello di churn $y = 0,5$. Dal grafico è chiaro, ancora una volta, come siano necessarie entrambe le componenti per avere un risparmio considerabile in termini di MPA. Inoltre emerge anche il fatto che valori di z uguali non necessariamente conducono alle stesse prestazioni. Ad esempio se $z = 0,4$ con $j = 0,5$ e $x = 0,8$ si hanno 21 MPA contro i 23 se $j = 0,8$ e $x = 0,5$, differenza destinata a crescere con l'aumentare della forbice fra i due valori. In generale quindi la componente x ha più peso in termini di una riduzione di MPA, rispetto a j.

Capitolo 4

Overlay Weaver

4.1 Introduzione

La trasformazione dell'idea di un algoritmo in un software funzionante richiede una discreta quantità di lavoro. Prendiamo come esempio una DHT. Gran parte dello sforzo viene impiegato in aspetti che non riguardano direttamente l'algoritmo stesso, ad esempio la gestione dei dati e le comunicazioni.

La proposta di [4] permette di separare servizi come DHT, multicast e anycast dal livello di routing sottostante. Il modello proposto suggerisce l'utilizzo di un *routing layer* basato sul concetto di key-based routing (KBR). Tuttavia il livello di routing rimane ancora *monolitico* e quindi di difficile implementazione.

Overlay Weaver ha migliorato l'approccio sfruttando l'idea che l'algoritmo di routing non è l'unica parte del *livello di routing* il quale deve eseguire altre funzioni tra cui le comunicazioni. Esiste una parte del routing, detta *routing process*, comune a tutti gli algoritmi. Da qui l'idea di dividere il livello di routing in più moduli, passando da una definizione monolitica ad una più flessibile.

OW fornisce una implementazione utilizzabile di questi concetti. Di fatto, si scinde il legame fra le funzionalità proprie di un algoritmo di routing e quelle del routing process. Una serie di interfacce sono create per generaliz-

zare le interazioni fra questi due componenti e supportano gli algoritmi più comuni quali Chord, Pastry, Tapestry, Kademlia e Koorde.

Questa decomposizione del livello di routing permette a implementazioni diverse del routing process di essere combinate con vari algoritmi di routing.

Il design modulare quindi, contribuisce alla rapidità dello sviluppo di algoritmi di routing, poiché consente di tralasciare gli aspetti comuni ad essi. Inoltre OW mette a disposizione un *emulatore*, strumento in grado di aiutare gli sviluppatori a testare e migliorare i propri algoritmi senza avere a disposizione una rete reale di nodi.

4.2 Architettura

OW è stato costruito prendendo come riferimento il key-based routing. Come già detto, in questa proposta il livello di routing è comunque monolitico, e di conseguenza, più faticoso da implementare.

L'idea principale di OW è stata quindi quella di dividere il livello di routing in più moduli, come si vede dalla figura 4.1.

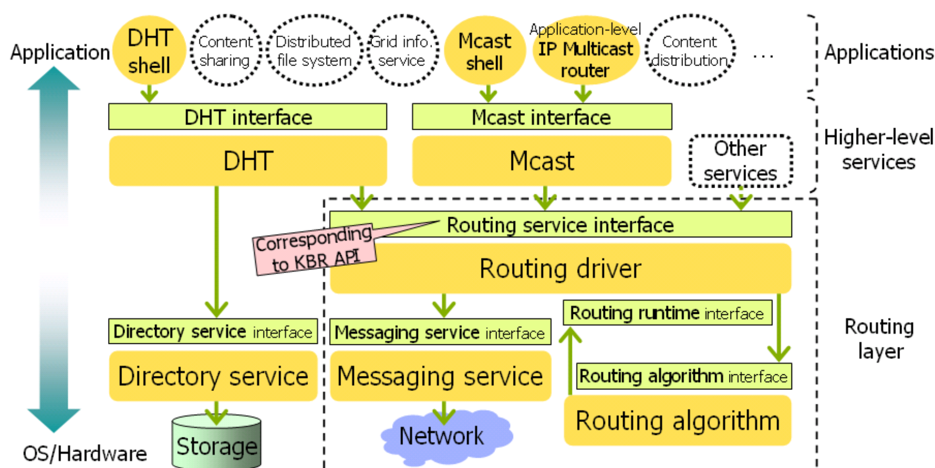

Figura 4.1: Architettura di Overlay Weaver

- *Routing driver*: è il componente che si occupa della gestione della fase di routing. Per eseguire il routing utilizza le informazioni ottenu-

te dall'algoritmo di routing. Due stili diversi di routing driver sono implementati, l'iterativo e il ricorsivo.

- *Routing algorithm*: il componente che viene utilizzato dal routing driver per svolgere il routing.

- *Messaging service*: questo componente si occupa della gestione dei messaggi TCP, UDP e anche delle comunicazioni intra-thread (utili per le fasi di emulazione).

Il routing layer in figura, composto dagli elementi appena descritti, corrisponde al key-based routing layer proposto da [4]. I servizi ad alto livello sono separati e costruiti sopra il *routing layer*.

Il componente *Directory Service* non fa parte del livello generale di routing in quanto è utilizzato in maniera esclusiva dalle DHT. Questo modulo fornisce le astrazioni per l'implementazione di un servizio di memorizzazione dei dati.

In generale il modello utilizzato da OW permette di poter combinare più componenti per la costruzione del livello di routing, senza che sia necessaria alcuna modifica.

Allo stato attuale sono fornite sette implementazioni di algoritmi di routing, due stili di routing process, due varianti di directory service più due implementazioni del messaging service. (Sono presenti altre due implementazioni del messaging service che eseguono comunicazioni inter-thread per l'ultilizzo dell'emulatore).

Un livello di personalizzazione così ampio permette ad un generico utente di poter scegliere la configurazione che più si adatta alle sue esigenze e permette allo sviluppatore interessato a modificare un solo aspetto un lavoro semplificato.

4.2.1 Routing Driver

Overlay Weaver fornisce due implementazioni del routing driver, iterativa e ricorsiva. Tra queste è possibile scegliere dinamicamente quale utilizzare. La figura 4.2 illustra come avvengono le comunicazione per entrambi gli stili di routing. I cerchi rappresentano i nodi, le linee i messaggi con a fianco indicato l'ordine di passaggio. Le linee tratteggiate rappresentano quei messaggi che non sono necessariamente scambiati durante il routing, ma che servono con protocolli non affidabili come UDP.

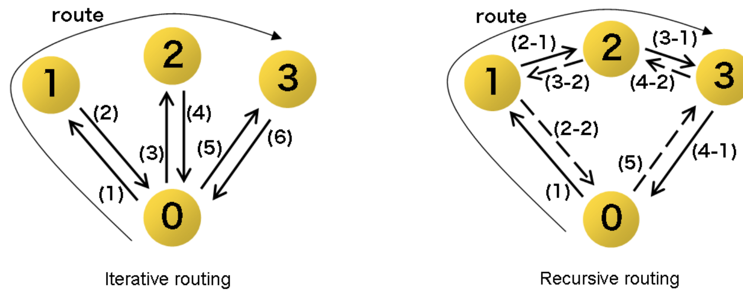

Figura 4.2: Routing iterativo e ricorsivo

In ogni caso i nodi danno priorità maggiore all'instradamento della query, piuttosto che alla conferma di un messaggio ricevuto. Perciò è lectio aspettarsi che il routing ricorsivo abbia una latenza minore rispetto all'iterativo, dove dominano i tempi di comunicazione.

Il numero di messaggi scambiati è $2n$ nel caso iterativo, $2n + 1$ in quello ricorsivo. Questo è valido per gli algoritmi che non effettuano un passo di routing supplementare (*adjustRoot*) come Chord. In questo caso il numero di messaggi è $2(n + 1)$.

4.3 Utilizzo di Overlay Weaver

La struttura a livelli di OW permette di utilizzare lo strumento a livelli diversi, in modo da ottenere il livello di personalizzazione desiderato, senza dover implementare servizi già implementati.

Per comprendere i paragrafi successivi è necessario avere un'idea di quali sono i packages che compongono il codice di Overlay Weaver. La tabella 4.1 fornisce un elenco di quelli principali.

Package	Descrizione
ow.dht	Funzioni standard di una DHT
ow.directory	Memorizzazione dei dati
ow.id	Gestione degli identificatori
ow.messaging	Astrazione del livello di rete
ow.routing	Routing sulle DHT
ow.tool	Oggetti per lo sviluppo

Tabella 4.1: Lista dei packages in Overlay Weaver e loro descrizione.

Adottando un approccio incrementale possiamo dividere le fasi di utilizzo e di successiva modifica, utili a raggiungere il livello di personalizzazione scelto.

I paragrafi successivi sono organizzati come segue. Nel paragrafo 4.3.1 analizziamo le conoscenze necessarie per utilizzare una DHT, posizionandoci (in riferimento alla figura 4.1 al livello applicazioni. Successivamente in 4.3.2 studiamo l'implementazione standard di una DHT e quali sono i suoi aspetti principali, posizionandoci tra i servizi ad alto livello e il routing layer. Infine in 4.3.3 approfondiamo i passi necessari per l'implementazione di un algoritmo di routing.

4.3.1 Utilizzo dei servizi ad alto livello

Overlay Weaver mette a disposizione dell'utente due servizi ad alto livello: multicast e distribuited hashed table.

Le funzionalità offerte dalla DHT di OW sono comuni indipendentemente dal tipo di routing scelto. I metodi sono specificati nell'interfaccia **ow.dht.DHT**, logicamente divisi in due gruppi: gestione del'overlay e metodi per la gestione dei dati.

Gestione overlay

I metodi *joinOverlay(String hostAndPort, int defaultPort)* e *joinOverlay(String hostAndPort)* effettuano la procedura di ingresso del nodo nell'overlay network. Il parametro *hostAndPort* deve indirizzare un nodo esistente dell'overlay di cui si vuole far parte, il nodo di bootstrap, nella forma *nomehost:porta.*

Il metodo *void clearRoutingTable()* è utilizzato per lasciare la rete cancellando ogni riferimento nella tabella di routing, mentre *void clearDHTState()* cancella le risorse della DHT, in particolare le coppie chiave-valore memorizzate localmente. Generalmente questi due metodi sono utilizzati congiuntamente quando un nodo vuole uscire dall'overlay.

Metodi classici DHT

L'inserimento di un dato sulla DHT è effettuato tramite l'operazione di PUT. Sono presenti due metodi che svolgono questa funzionalità: *put(ID key, V value, long ttl)* e *put(ID key, V value, long ttl, ByteArray hashedSecret)*

Il primo metodo inserisce la coppia chiave-valore nella DHT. Esiste la possibilità di associare più valori alla stessa chiave, ma valori uguali (*equals()*) sono unificati. Il parametro *ttl*,espresso in millisecondi, rappresenta il periodo temporale in cui il dato è valido ed è a disposizione sulla rete. Passato il valore indicato dal ttl il dato viene rimosso.

Il parametro *hashedSecret* è un oggetto che viene associato al valore per aspetti di sicurezza. In generale quindi, un dato è identificato dalla coppia <valore,secret>. In altre parole due dati sono uguali se coincidono sia per quanto riguarda il valore, sia per il *secret* associato. La rimozione di una chiave è permessa solo se all'operazione è associato l'oggetto secret corrispondente.

Per recuperare dati sulla DHT è utilizzato il metodo *get(ID key)* il quale restituisce l'insieme di valori associati alla chiave (*key*) specificata. Se nessun valore è stato trovato *null* è restituito.

Esistono due possibili metodi per la rimozione di un dato dalla DHT. Il metodo *remove(ID key, V value, ByteArray hashedSecret)* rimuove la coppia

chiave-valore specificata. Se alla chiave è associato più di un valore, è rimosso solo quello indicato dal metodo. Il metodo *remove(ID key, ByteArray hashedSecret)* rimuove tutti i valori associati alla chiave specificata (oltre che la chiave stessa).

Generalmente quando si definisce un servizio di overlay specifico si indicano alcuni parametri, per personalizzare al massimo la DHT. Il comportamento di default può essere modificato tramite la classe **ow.dht.DHTConfiguration** (tabella 4.2).

Tipo	Nome	Valore di default
String	ROUTING_STYLE	Iterative
String	ROUTING_ALGORITHM	Chord
String	MESSAGING_TRANSPORT	UDP
boolean	DO_EXPIRE	true
int	NUM_REPLICA	1

Tabella 4.2: Principali parametri dell'interfaccia DHTConfiguration.

La variabile ROUTING_STYLE permette di scegliere fra due tipologie di routing diverse. DO_EXPIRE abilita la rimozione dei dati una volta scaduto il TTL, mentre NUM_REPLICA determina il numero di copie dei dati che vengono replicate automaticamente. Questi parametri sono necessari per l'esecuzione di una DHT. Se il valore di un parametro non è specificato esplicitamente è usato il valore di default.

Modalità di accesso

Per accedere ai metodi elencati precedentemente sono possibili due modalità. In generale quando l'accesso alla DHT avviene dalla stessa macchina dove è in esecuzione un'istanza di OW è possibile accedere alla DHT tramite chiamate di metodo. In questo caso si hanno a disposizione tutti i metodi dell'interfaccia della DHT, ed è possibile controllare e modificare la configurazione di partenza, così come eseguire operazioni sulla DHT.

Se non è possibile chiamare direttamente i metodi è necessario utilizzare il protocollo RPC (Remote Procedure Call). In questo caso le funzioni sono più limitate, si hanno a disposizione solo le funzioni base quali get, put e

remove. Questo perché per avviare un server RPC è necessario che la DHT sia già in esecuzione.

4.3.2 Modificare le funzionalità delle DHT

In alcuni ambiti è necessario sfruttare il routing pensato per una DHT per l'implementazione di servizi diversi (o semplicemente per cambiarne l'implementazione) rispetto alle funzioni classiche offerte dalla DHT.

In generale è possibile:

1. Lasciare intatta l'interfaccia originale della DHT, fornendo un'implementazione diversa delle stesse funzionalità.

2. Modificare e aggiungere nuove funzionalità come estensione di quelle base.

In questo paragrafo analizzeremo quali sono gli aspetti principali da considerare per realizzare le funzionalità presentate ai punti precedenti.

Overlay Weaver fornisce una implementazione di default di una DHT alla classe **ow.dht.StandardDHTImpl**.

Per studiare questa classe è necessario dividere i vari aspetti che la compongono, ed analizzare ognuno in modo isolato.

Configurazioni

Gli aspetti generali della configurazione di una DHT vengono gestiti, come abbiamo visto precedentemente, dalla classe **DHTConfiguration** (tabella 4.2).

Nella classe *StandardDHTImpl* è importante conoscere che altri tipi di parametri possono essere specificati sia per il routing process che per l'algoritmo di routing scelto.

La tabella 4.3 presenta i principali parametri presenti nella classe **RoutingServiceConfiguration** che controlla il routing process. Ad esempio sono presenti parametri per controllare la dimensione della struttura di routing e il numero di 'vicini' inviati nella risoluzione del routing.

Tipo	Nome	Valore di default
int	NUM_OF_CLOSEST_NODES_REQUESTED	8
int	NUM_OF_NODES_MAINTAINED	20
int	TTL	128

Tabella 4.3: Principali parametri in RoutingServiceConfiguration.

Per quanto riguarda la configurazione specifica dell'algoritmo di routing scelto, alla tabella 4.4 sono elencati, come esempio, i parametri presenti nella classi di configurazione per Chord.

Tipo	Nome	Valore di default
boolean	DO_FIX_FINGERS	true
double	PROB_PROPORTIONAL_TO_ID_SPACE	0.7
double	FIX_FINGERS_INTERVAL_PLAY_RATIO	0.3

Tabella 4.4: Principali parametri in ChordConfiguration.

Routing

L'interfaccia **ow.routing.RoutingService** definisce i metodi che permettono ad una applicazione di usufruire dei servizi di routing. L'implementazione di una DHT utilizza invocazioni ai metodi definiti in questa interfaccia per effettuare il routing verso un nodo.

Esistono diverse classi che implementano questa interfaccia secondo strategie diverse (routing iterativo o ricorsivo). A loro volta queste classi fanno riferimento ai metodi descritti nel paragrafo 4.3.3 la cui implementazione dipende dall'algoritmo utilizzato.

I metodi che compongo questa interfaccia sono:

- *RoutingResult invokeCallbacksOnRoute(ID target, int numRootCandidates, java.io.Serializable[] returnedValueContainer, CallbackResultFilter filter, int tag, java.io.Serializable[] args).*
  Esegue il routing fino al nodo target. In ogni nodo attraversato durante il processo vengono invocati metodi di *callback*, precedentemente registrati tramite il metodo *void addCallbackOnRoute(CallbackOnRoute*

callback). I risultati vengono memorizzati nell'array *returnedvalueConteiner*.

- *RoutingResult routeToRootNode(ID target, int numRootCandidates)*. Come il precedente esegue il routing fino al nodo specificato, ma non invoca nessun metodo.

In particolare questi metodi hanno la funzione, dato un ID, di trovare quale nodo memorizza il dato identificato con quell'ID, detto nodo *root*. Questa informazione è contenuta nell'oggetto restituito **ow.routing.RoutingResult**. Inoltre contiene anche la lista dei nodi attraversati per arrivare alla destinazione oltre che una lista di successori del nodo root. La dimensione di questa lista dipende dal valore presente nella tabella 4.3.

Un'altra funzione di RoutingService è quella di rendere, almeno a questo livello, trasparente l'arrivo e la conseguente gestione dei messaggi.

Messaggi

I messaggi vengono utilizzati per realizzare la comunicazione fra i nodi appartenenti all'overlay. Le classi per la gestione dei messaggi sono contenute nel package **ow.messaging**.

Generalmente, quando si vuole modificare il funzionamento di una DHT si devono creare nuovi tipi di messaggi in aggiunta a quelli esistenti.

Un messaggio di base è composto da due elementi fissi quali il *TAG* (un intero che identifica il tipo di messaggio) e le informazioni sul mittente. Inoltre è possibile aggiungere un numero variabile di informazioni aggiuntive.

Supponiamo ad esempio che si voglia modificare il funzionamento della GET per aggiungere una nuova funzionalità. Dopo aver individuato quale nodo contiene le informazioni richieste dalla GET è necessario comunicare a quest'ultimo, mediante un messaggio diverso rispetto a quello della GET originale, quali operazioni effettuare.

L'aggiunta di un nuovo tipo di messaggio può essere schematizzata come segue:

- Estendere la classe **ow.messaging.Tag** in modo da aggiungere un nuovo tag per il messaggio.
- Modificare la classe **ow.dht.impl.DHTMessageFactory** in modo da creare un messaggio del nuovo tipo automaticamente.
- Creare una classe di gestione per il messaggio. L'interfaccia da estendere è **ow.messaging.MessageHandler**. Nel metodo *process(Message msg)* di questa classe vanno inserite tutte le azioni da compiere alla ricezione del messaggio.
- Aggiungere l'handler del messaggio creato precedentemente all'oggetto RoutingService utilizzato, tramite il metodo *void addMessageHandler(int tag, MessageHandler handler)*

Per l'invio dei messaggi si utilizzano i metodi presenti nella classe *MessageSender*. Questa classe fornisce l'opportunità di mandare i messaggi in modo sincrono (si aspetta la risposta) oppure in modo asincrono.

Identificatori

Il modo in cui gli identificatori vengono assegnati ai nodi ed ai dati di una DHT assume un ruolo chiave per lo sviluppo di nuove funzionalità.

Il package **ow.id** fornisce tutti gli strumenti per la manipolazione di questo aspetto. La creazione di un oggetto ID è piuttosto semplice, tutto quello che si deve indicare è il suo valore e la sua lunghezza espressa in byte.

Un oggetto della classe ID è utilizzato attivamente in due contesti. Può essere assegnato come identificatore di un nodo insieme alla classe IDAddressPair, in modo da collocarlo nello spazio della DHT, oppure viene assegnato ad un dato.

La classe *IDAddressPair* è la coppia ID-indirizzo che identifica un nodo. In una implementazione TCP o UDP l'indirizzo è rappresentato da un indirizzo IP e da una porta.

Directory

Con directory si intendono i meccanismi utilizzati per la memorizzazione locale delle informazioni a carico di un nodo. Nel package **ow.directory** sono fornite, come già accennato prima, implementazioni di due tipi di directory. La distinzione è sul tipo di architettura utilizzata.

Sono utilizzate sia le Hashmap standard di Java che l'implementazione Java del *Berkley DB* [6].

Per ognuna di queste due classi esistono due implementazioni che differiscono dal fatto di poter associare più valori ad ogni chiave oppure uno solo.

Nel package **ow.directory.expiration** sono presenti le classi che forniscono le funzionalità di *expiring* (ogni valore associato ad una chiave ha una scadenza in termini temporali). É presente un'implementazione sia per valori singoli che per valori multipli e, per ognuna, è possibile scegliere se utilizzare le Hashmap o il Barkley DB.

4.3.3 Implementare un nuovo algoritmo di routing

Overlay Weaver fa una distinzione fra il *processo di routing* e *l'algoritmo utilizzato*, in modo che l'implementazione di un algoritmo non dipenda dal tipo di routing scelto.

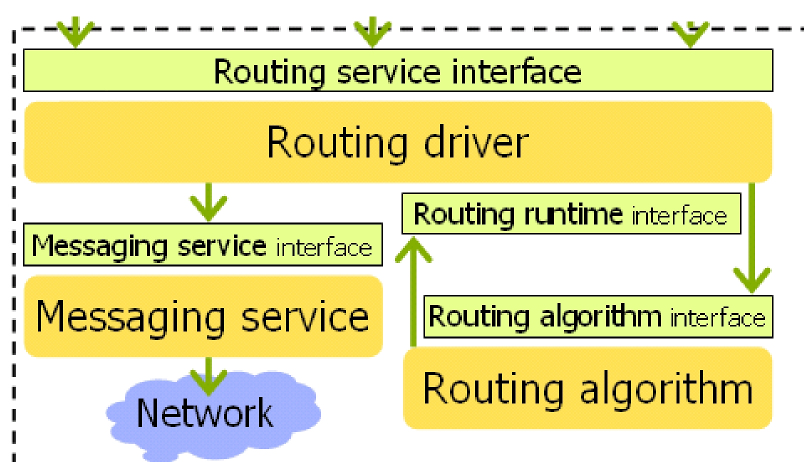

Figura 4.3: Routing layer di Overlay Weaver

La figura 4.3 è un particolare dell'architettura di OW: Il modulo **routing driver** è il componente che esegue il processo di routing e per farlo si appoggia all'algoritmo di routing. Per implementare un nuovo algoritmo di routing devono essere implementati i metodi da esporre al routing process. Questi metodi sono presenti nell'interfaccia **ow.routing.RoutingAlgorithm**

Successivamente riferiremo come nodo **root** per un ID, il nodo che memorizza la risorsa identificata dall'ID all'interno di una DHT.

Di seguito un elenco dei metodi dell'interfaccia RoutingAlgorithm.

- *void join(IDAddressPair[] route)*
  Quando un nodo vuole unirsi all'overlay, il routing driver esegue il routing usando l'ID del nodo entrante come obiettivo. Questo metodo è invocato sul nodo subito dopo l'ingresso del nodo nella rete. Il parametro *route* è la lista dei nodi attraversati fino alla root. *IDAddressPair* è una coppia che specifica l'identificatore di un nodo e l'indirizzo di un 'canale' con cui comunicare con quel nodo. Nel caso di TCP o UDP l'identificatore consiste nell'indirizzo IP del nodo, il canale corrisponde al numero di porta.

- *IDAddressPair[] closestNodes(ID target, int maxNumber, RoutingContext context)*
  Questo metodo restituisce un array ordinato di nodi che sono vicini al *target* specificato. La definizione di vicinanza varia a seconda dell'algoritmo. Il parametro *maxnumber* limita il numero di nodi che devono essere restituiti. Il routing driver tratta i nodi come i candidati per l'hop successivo. E' possibile che uno dei nodi della lista sia lo stesso che esegue il metodo. *RoutingContext* contiene informazioni supplementari riguardanti il routing appena effettuato.

- *IDAddressPair[] adjustRoot(ID target)*
  Questo metodo è invocato nell'ultima fase del routing, su un nodo il cui ID è particolarmente vicino all'ID obiettivo del routing (*target*). Il metodo restituisce una lista di candidati per il nodo root. Se il nodo stesso è la root, è restituito un array vuoto.

- *void join(IDAddressPair joiningNode,IDAddressPair lastHop, boolean isRootNode)*
  Questo metodo è invocato su ogni nodo del percorso durante l'esecuzione del routing per una richiesta di join. Il parametro *joiningNode* rappresenta il nodo che vuole entrare nell'overlay, mentre *lastHop* e il nodo precedente sul percorso. Il parametro *isRootNode* è vero quando la root esegue questo metodo.

- *void touch(IDAddressPair from)*
  Questo metodo è invocato quando un nodo riceve un messaggio da *from*

- *void forget(IDAddressPair node)*
  Comunica al nodo di escludere dalla tabella di routing *node*. Il metodo viene chiamato quando la comunicazione con un nodo fallisce un certo numero di volte successive.

- *BigInteger distance(ID to, ID from)*
  Resituisce la distanza fra i due identificatori specificati. Questo metodo non fa parte dell'interfaccia RoutingAlgorithm, poiché non è chiamato dal routing driver ma viene usato solo nell'implementazione interna dell'algoritmo. In pratica solo un certo numero di algoritmi utilizzano il metodo e questo giustifica il suo non inserimento nell'interfaccia.

Quelli appena visti sono i metodi che devono essere implementati per la definizione di un algoritmo di routing. Il prossimo paragrafo illustra più in dettaglio come e quali metodi dell'interfaccia appena descritta vengono implementati dagli algoritmi presenti in OverlayWeaver.

Algoritmi: dettagli sull'implementazione

Overlay Weaver, come già detto più volte, mette a disposizione implementazioni dei più conosciuti algoritmi di routing.

Non tutti questi algoritmi hanno però bisogno di implementare tutti i metodi presenti nell'interfaccia RoutingAlgorithm. La tabella 4.5 specifica quali metodi sono implementati dagli algoritmi.

	Chord	Pastry	Tapestry	Kademlia
closestNode	X	X	X	X
adjustRoot	X			
join(1)	X			
join(2)		X	X	
touch		X	X	X
forget	X	X	X	X
distance	X	X		X

Tabella 4.5: Corrispondenze fra i metodi dell'interfaccia RoutingAlgorithm e vari algoritmi di routing.

Il metodo principale dell'interfaccia è *closestNodes*, il routing driver infatti ne ha bisogno per tutte le implementazioni. Inoltre tende ad essere il metodo con il maggior numero di linee di codice.

Chord è l'unico algoritmo che implementa *adjustRoot*. Negli altri algoritmi il nodo più vicino (trovato con *closestNode*) è il nodo root. Con Chord questo non è sempre vero, in quanto il nodo root può essere il successore del nodo più vicino. Quindi si utilizza *adjustRoot* per risolvere la situazione.

Tapestry non utilizza la distanza fra identificatori per eseguire il routing ma determina l'hop successivo secondo una procedura propria. Pastry non usa la distanza nella prima fase di routing ma il metodo *distance* è usato per ordinare i nodi nelle strutture dati (leaf set) usate nell'ultima parte del routing.

Quando *touch* è invocato si aggiunge il nodo alle proprie strutture di routing. Sebbene questo comportamento è specifico di Kademlia, in Overlay Weaver è utilizzato anche per Pastry e Tapestry.

4.4 Strumenti di sviluppo

4.4.1 Emulatore

L'emulatore è lo strumento di riferimento per la fase di test in un ambiente virtuale. Può funzionare sia su singolo host oppure in modo distribuito su

più macchine, permettendo di valutare gli algoritmi avendo a disposizione un numero considerevole di nodi virtuali.

Il package di riferimento all'interno di Overlay Weaver è **ow.tool.emulator**.

Utilizzo

L'emulatore è uno strumento che può essere usato in due modalità.

- Modalità interattiva, digitando direttamente i comandi da shell.
- Modalità batch, tramite la definizione di un file, detto *scenario*, contenete una lista di comandi.

Sicuramente la seconda opzione è la più interessante in quanto permette la creazione di ambienti più complessi ed elaborati. Una possibilità interessante è quella di poter confrontare nuove soluzioni (a qualsiasi livello) con quelle precedenti usando lo stesso ambiente, avendo in questo modo un confronto diretto.

L'emulatore può funzionare secondo due ulteriori modalità. In modalità normale l'emulatore è eseguito su un solo computer, alternativamente più computer collaborano nell'esecuzione dell'emulatore. Il file scenario viene utilizzato in entrambi in casi, ma per l'emulatore distribuito è necessaria la definizione di un altro file con il nome dei computer coinvolti. In questo file devono essere specificati i nomi dei nodi coinvolti seguiti da un numero che indica il primo nodo emulato assegnato all'host. Vediamo un esempio.

Esempio 4. *File di descrizione per uno scenario distribuito.*
hostname0 0
hostname1 1
hostname2 101

In questo caso solo il primo nodo virtuale (identificato dall'emulatore con emu0) è assegnato al computer *hostname0*. Al nodo *hostname1* vengono assegnati 100 nodi virtuali, mentre i rimanenti sono assegnati al nodo *hostname2*.

Per eseguire una sessione di emulazione distribuita, l'utente manda in esecuzione l'emulatore principale il quale contatta tramite SSH gli altri computer sui quali, a loro volta, è invoato l'emulatore. Le istanze remote dell'emulatore sono dette *workers*. Una serie di comandi specifici sono utilizzati per gestire le sessioni distribuite.

Normalmente per eseguire l'emulatore si esegue il file **owemu** presente nella directory *bin* del package di Overlay Weaver, passando come opzione il file scenario ed eventualmente il file con la lista dei computer da utilizzare.

Scenario

Tipicamente uno scenario è composto da un elenco di comandi che verranno eseguiti in successione dall'emulatore. Un esempio del contenuto di un file scenario è mostrato nella figura 4.4.

```
timeoffset 2000

# invoca il primo nodo
class ow.tool.dhtshell.Main
arg -p 10000
schedule 0 invoke

# invoca 3 nodi
arg
schedule 1000,1000,3 invoke

timeoffset 7000

# i 3 nodi enteano nell'overlay
schedule 0 control 1 init emu0
schedule 1000 control 2 init emu0
schedule 2000 control 3 init emu0

# eseguo put e get
schedule 4000 control 1 put a_key a_value 600 secret
schedule 5000 control 1 get a_key
```

Figura 4.4: Esempio di file scenario

Il comando *class* indica la classe da eseguire mentre *arg* ne definisce gli argomenti. *Schedule* esegue l'ultima classe specificata chiamandone il metodo *main*. Esiste la possibilità di specificare il numero di esecuzioni e un intervallo temporale fra due invocazioni. Più dettagli sui comandi dell'emulatore nell'appendice A.

4.4.2 Visualizer

Il *Visualizer* è uno strumento per visualizzare i nodi e le loro comunicazioni sulla rete in tempo reale. La posizione in cui un nodo è disegnato è influenzata dall'ID del nodo stesso.

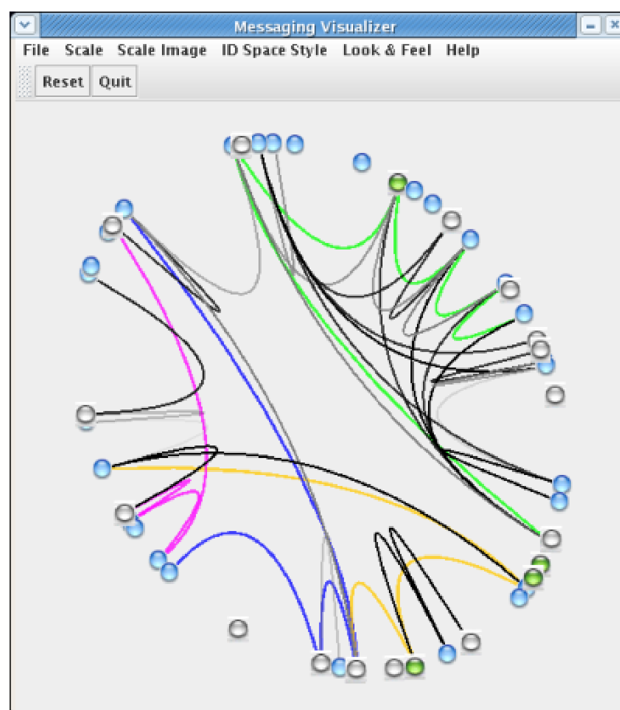

Figura 4.5: Interfaccia del Visualizer

La figura 4.5 mostra un'immagine dello strumento in esecuzione. Gli archi rappresentano una *comunicazione fra nodi*, a loro volta rappresentati da cerchi. Questo tool può essere utile per avere una valutazione intuitiva del funzionamento degli algoritmi e può facilitarne i test.

Il visualizzatore può essere utilizzato sia quando l'overlay viene valutata mediante l'emulatore, sia quando la valutazione avviene su rete reale. In quest'ultimo caso bisogna però considerare che il carico di messaggi sulla rete aumenta.

Capitolo 5

Remedy: Implementazione

In questo capitolo verrà esaminato Remedy da un punto di vista architetturale ed implementativo. In particolare verranno descritti i packages, verranno mostrate le principali scelte progettuali e le classi definite.

5.1 Package

Forniamo inizialmente un elenco dei package che compongono Remedy, accompagnati da una breve descrizione.

- **remdey**: il package principale. Contiene le classi che forniscono le funzionalità principali (put, get) e si appoggia direttamente ai servizi offerti da Overlay Weaver
- **data**: contiene le classi relative alle strutture per lo scambio di dati
- **monitor**: package che contiene le classi che funzionano da *sensori* per recuperare i dati dall'host monitorato
- **rpc**: contiene le funzionalità di Remote Procedure Call per utilizzare le funzionalità della DHT da remoto.
- **test**: package in cui sono contenute tutte le classi relative ai test da me effettuati

5.2 Il package Data

In Remedy una risorsa viene rappresentata un insieme di coppie (attributo, valore), da un proprietario e da un *time to live* (TTL) espresso in millisecondi.

```
public class Resource extends Hashtable<String, Integer>
{
    private Long TTL = 200001;
    private String owner;

    public Resource(String owner)
    {
        super();
        this.owner = owner;
    }

    public void putAttribute(String attribute, int value) {..}

    public void setTTL(Long millisec) {..}

    public String toString() {..}

    public String getOwner() {..}

    public boolean equals(Object o) {..}

    public Object[] toRPCFormat() {..}

    public static Resource fromRPCFormat(Object[] data) {..}

    public static Resource getRandomResource(int attributes) {..}

    public static Resource getResourceByState() {..}

}
```

Figura 5.1: Struttura della classe *Resource*

La classe Resource eredita da una Hastable in cui le chiavi sono i nomi degli attributi. Oltre al metodo per aggiungere una nuova coppia all'insieme, sono previsti anche alcuni metodi di utilità. In figura 5.1 sono presenti i metodi e le variabili di istanza che la compongono. In particolare i metodi *toRPCFormat()* e *fromRPCFormat(Object [])* sono utilizzati come metodi rispettivamente di codifica e di decodifica negli scambi via RPC. Il metodo *getResourceByState()* crea una risorsa a partire dalle rilevazioni fatte dalle classi presenti nel package **monitor**. Infine il metodo *getRandomResour-*

ce(int) crea una risorsa in cui i valori degli attributi sono casuali e con un numero di attributi pari a quanto specificato nel parametro.

```
-<root>
  -<attribute>
     <name> MemorySizeMB </name>
     <vmin> 64 </vmin>
     <vmax> 4096 </vmax>
   </attribute>
  -<attribute>
     <name> CpuSpeedMhz </name>
     <vmin> 100 </vmin>
     <vmax> 4000 </vmax>
   </attribute>
  -<attribute>
     <name> CpuFreePercent </name>
     <vmin> 0 </vmin>
     <vmax> 100 </vmax>
   </attribute>
  -<attribute>
     <name> AvailableRam </name>
     <vmin> 0 </vmin>
     <vmax> 4096 </vmax>
   </attribute>
  -<attribute>
     <name> AvailableSwapSpace </name>
     <vmin> 0 </vmin>
     <vmax> 4096 </vmax>
   </attribute>
 </root>
```

Figura 5.2: File XML contenente la descrizione degli attributi

Tutti i possibili parametri utilizzabili sono definiti, insieme ad un indicazione del valore minimo e massimo che possono assumere, in un file chiamato *attributespace.xml*, di cui un esempio in figura 5.2. I dati presenti in questo file sono caricati dalla classe statica *AttributeSpace*.

La classe IDCache (figura 5.3) memorizza le corrispondenze fra nodo e il range ID che gestisce e le informazioni riguardo la popolarità. il metodo *resolve(ID)* restituisce la coppia (ID, indirizzo IP) del nodo che gestisce l'identificatore passato come parametro. Nel caso il nodo non sia presente nella struttura, restituisce *null*. Il metodo *popularity(ID, String)* restituisce le informazioni sulla popolarità per uno specifico attributo riguardanti il nodo che ha come identificatore ID.

```
public class IDCache
{
    Vector container;

    public IDCache()
    {
        container = new Vector();
    }

    public void add(IDAddressPair node, ID min, ID max) {}

    public void remove(IDAddressPair node) {}

    public boolean isInCache(IDAddressPair node) {}

    public IDAddressPair resolve(ID key) {}

    public boolean popularity(ID node, String att) {}
}
```

Figura 5.3: Struttura della classe IDCache

5.3 Pubblicazione

La *pubblicazione* di una risorsa si divide in due parti: la parte gestita dal nodo provider e quella gestita dal nodo che riceve la richiesta di PUT. Nel paragrafi successivi indicheremo con il termine *nodo provider* il nodo che inizia un operazione, sia essa una pubblicazione o la risoluzione di una query. Il nodo che riceve queste operazioni sarà indicato come *nodo destinatario.*

5.3.1 Pubblicazione: nodo provider

In figura 5.4 è presentato lo pseudo codice dell'esecuzione di una pubblicazione in Remedy. Per ogni coppia (attributo, valore) presente nella risorsa da memorizzare si eseguono le azioni elencate di seguito.

Inizialmente si recuperano dalla classe *AttributeSpace* le informazioni relative all'attributo, in particolare il suo range di valori, in modo da poter calcolare la chiave.

Successivamente si prepara il messaggio da inviare in remoto il quale contiene, oltre all'ID del nodo provider, la chiave corrispondente al valore dell'attributo e la risorsa da memorizzare. A questo punto si controlla nell'oggetto *IDCache* se si hanno informazioni sul nodo che gestisce la chiave.

Se il risultato è positivo si controllano le informazioni sulla popolarità per verificare se è possibile evitare la PUT. Viene utilizzato il valore *quanto* (Q) che rappresenta l'intervallo di tempo (in secondi) fra due successive misurazioni dello stato della macchina dal processo *Monitor*. Ad esempio nella attuale implementazione questo valore è impostato a 30 secondi; successivamente useremo questo valore se non specificato diversamente.

Per ogni nodo destinatario nella cache del nodo sorgente sono quindi presenti le seguenti informazioni: il numero di query per quanto (QpQ) ricevute dal nodo e, per ogni attributo, la sua popolarità (POP) espressa in termini percentuali. Queste informazioni sono inviate dal nodo destinatario nel messaggio di risposta della PUT.

Utilizzando queste informazioni è calcolato il numero di Q necessari affinché l'attributo sia utilizzato come dominante nella risoluzione di una query. Indichiamo questo valore con NQ; esso è definito da questa formula:

```
for (Attribute a : Resource)
{
  key = calcola_chiave(a.value, a.vmin, a.vmax);
  msg = create_message(PUT, ID e IP mittente, key, a, Resource);

  target = cache.resolve(key);

  if (target != null)
  {
    if cache.poplarity(key, a)
      continue;

    reply = send(msg, target);
    cache.add(reply.source, info);
    if (reply == PUT_FAILED)
      cache.remove(target);
    else
      continue;
  }

  target = do_routing(key);
  reply = send(msg, target);

  cache.add(reply.source, info);

}
```

Figura 5.4: Pseudo codice della pubblicazione sul nodo provider

$$NQ = \frac{1}{\frac{QpQ \times POP}{100}}$$

Esempio 5. *Si consideri un nodo N con $QpQ = 10$ e in cui l'attributo A abbia una popolarità del 10%. Questo significa che ogni 30 secondi ($NQ = 1$), N elabora una query con A come attributo dominante.*

Verificando il numero di NQ su un attributo è possibile scegliere se effettuare l'aggiornamento o meno. Con $TQ = 3$, ad esempio, dopo la prima pubblicazione possono essere evitate le due successive.

Se la PUT non è andata a buon fine (messaggio di tipo PUT_FAILED) si elimina la vecchia entry nella cache riguardo al nodo destinatario e si passa al punto successivo. In entrambi i casi le informazioni del messaggio di risposta vengono inserite nella cache.

Nel caso non si abbiano informazioni presenti nella cache riguardo alla chiave o queste siano obsolete si procede con una PUT normale. Il primo passo è quello di effettuare il routing in modo da ottenere, attraverso OW, il nodo che gestisce la chiave. Si invia il messaggio al nodo e le informazioni sulla risposta sono aggiunte in cache.

5.3.2 Pubblicazione: nodo destinatario

Alla ricezione di una richiesta di PUT il nodo esegue le azioni descritte nello pseudo codice in figura 5.5. La prima operazione è quella di decodificare il contenuto del messaggio e controllare di essere il nodo che gestisce la chiave con cui viene indicizzata la risorsa. Ricordiamo che il nodo potrebbe non essere quello che gestisce la chiave nel caso in cui la struttura dell'overlay sia cambiata rispetto all'ultimo aggiornamento effettuato. In caso negativo si prosegue con l'invio al mittente di una messaggio di tipo PUT_FAILED.

Se la chiave è corretta si aggiungono i dati nella directory locale. La directory locale è implementata attraverso l'oggetto *RemDir*. Questo oggetto è una *wrapper* per l'oggetto directory di OW, che memorizza i valori utilizzando come chiave il nome dell'attributo e come valore l'intera risorsa. Una

```
data = msg.decode();
if (valid(data.key))
{
  dir.add(msg.attribute, data.resource);
  msg = create_message(DHT_REPLY, info);
}
else
{
  msg = create_message(PUT_FAILED, info);
}
send(msg, data.source);
```

Figura 5.5: Pseudocodice della pubblicazione sul nodo destinatario

generica entry per la tabella è mostrata in figura 5.6; il generico attributo A_i indicizza un insieme di risorse R.

Ai	R1,...,Ri,...,Rn

Figura 5.6: La generica i-esima entrata della directory in cui sono memorizzate le risorse.

Nel messaggio di ritorno, in entrambi i casi, si inseriscono le informazioni riguardo l'intervallo di identificatori che viene gestito dal nodo, oltre le informazioni sulla popolarità dei vari attributi.

Il TTL della risorsa è impostato seguendo lo stesso procedimento che il nodo provider utilizza per verificare la popolarità. Il TLL è calcolato a partire dalle informazioni sull'attributo che indicizza la risorsa. Esso assume un valore pari a NQ moltiplicato per Q, a cui si aggiunge un 10% per compensare eventuali ritardi nelle comunicazioni. Il numero di QpQ e la popolarità di un attributo sono aggiornati al momento di invio del messaggio di risposta della PUT, in modo da avere dati sincronizzati fra i due nodi.

5.4 Risoluzione delle query

La risoluzione di una query ha inizio con l'immissione della richiesta nella rete da parte del nodo che vuole cercare risorse. Il messaggio è inoltrato da ogni nodo al successivo attraversando la parte di DHT interessata. Infine i risultati sono inviati al nodo iniziale.

5.4.1 Query: nodo provider

Il metodo di risoluzione di una query utilizza come parametro l'oggetto *Query* del package **data**. L'elenco dei metodi principali di questo oggetto è fornita in figura 5.7.

```
public class Query
{
    private Double maxSelectivity;
    private String mostSelectAttr;
    private int maxResult;

    public Query()
    {
        mostSelectAttr = "";
        maxSelectivity = Double.MAX_VALUE;
        maxResult = Integer.MAX_VALUE;
    }

    public void addCostraint(String attribute, int vmin, int vmax) {}

    public String getMostSelective() {}

    public Double getCurrentSelectivity() {}

    public int getCostraintNumber() {}

    public boolean match(Resource res) {}

    public Object[] toRPCFormat() {}

    public static Query fromRPCFormat(Object[] data) {}

    public static Query getRandomQuery() {}

    public static Query getRandomQuery(int lowSelPercentage) {}
}
```

Figura 5.7: Struttura della classe Query

In questo oggetto sono contenuti tutti i vincoli espressi come un attributo accompagnato da due valori, uno minimo ed uno massimo che rappresentano

l'intervallo cercato. I campi *Double maxSelectivity* e *String mostSelectAttr* rappresentano in ogni momento rispettivamente la selettività massima attuale (più piccolo è questo valore e più alta è la selettività della query) e la descrizione dell'attributo che fornisce tale selettività; tali valori sono aggiornati ogni volta che un nuovo vincolo è aggiunto alla query. Il valore *maxResult* è il limite superiore dei risultati da cercare. Anche in questo oggetto sono presenti metodi di utilità come quelli di codifica e decodifica per le comunicazioni RPC vi sono anche due metodi che generano query casuali. Di particolare utilità durante i test si è rilevato il metodo *getRandomQuery(int)* che fornisce una query casuale la cui selettività è pari alla percentuale indicata dal parametro.

Lo pseudo codice dell'algoritmo di risoluzione della query è presente in figura 5.8.

```
keyMin = calcola_chiave(a.vmin, a.min, a.max);
keyMax = calcola_chiave(a.vmax, a.min, a.max);

target = do_routing(keyMin);

msg = create_message (RECURSIVE_GET, source, last, keyMax,
    query, results, visitedHost, query_id);

send(msg, target);
```

Figura 5.8: Pseudo codice della risoluzione di una query multi attributo sul nodo provider

Inizialmente, utilizzando i dati dell'oggetto *Query* sono calcolate la chiave minima *keyMin* e massima *keyMax* che rappresentano lo spazio da esplorare sull'anello della DHT. Si esegue il routing utilizzando come chiave *keyMin* e si invia il messaggio di query (*RECURSIVE_ GET)* al primo nodo sull'arco. Questo messaggio contiene i seguenti campi:

- una coppia di valori (ID e indirizzo IP) che identifica il nodo provider, quello che ha iniziato la query;

- una coppia di valori (ID, IP) che rappresenta l'ultimo nodo che ha inviato il messaggio.

- la chiave massima da raggiungere (*keyMax*);
- tutta la query;
- una struttura dati contenente i risultati trovati *results*;
- una struttura dati con l'elenco dei nodi visitati *visitedHost*;
- un identificatore unico della query, assegnato dal nodo provider.

Il codice in figura 5.8 è eseguito da un thread che si mette in attesa dopo aver inviato il messaggio. Se dopo un certo timeout nessun dato è arrivato la query è considerata fallita.

Quando tutti i nodi che contengono possibili dati sono stati attraversati, al nodo provider è inviato un messaggio del tipo *RECURSIVE_REPLY_MESSAGE* che sveglia il thread (riconosciuto usando l'identificatore della query) e passando come parametro l'elenco dei risultati ottenuti. Questa operazione è quindi *asincrona* e il nodo provider, durante la fase di risoluzione, può eseguire altre azioni.

5.4.2 Query: nodo destinatario

Alla ricezione del messaggio descritto nel paragrafo precedente, un generico nodo compie le azioni schematizzate dallo pseudo codice in figura 5.9

Dopo aver decodificato il messaggio, controlla di non essere già presente in *visitedHost* per evitare loop, possibili nel caso di reti con un numero di nodi pari a poche unità e con query la cui selettività non sia molto alta. Nel caso si verifichi un loop invia immediatamente i risultati al nodo provider interrompendo di fatto la diffusione del messaggio, altrimenti aggiunge il suo nome alla lista dei nodi visitati.

Successivamente controlla nella directory locale se alcuni dati soddisfano i vincoli della query utilizzando il metodo *match(Resource)* dell'oggetto *Query*, nel caso aggiungendoli a *results*. Se il numero dei risultati trovati fino a quel momento supera il massimo numero indicato dalla query invia i risultati al nodo provider. Viene quindi confrontato poi l'ID del nodo con quello

```
data = msg.decode();

self = get_self_ID();
if (visitedHost.contains(self))
  invia_risultati_sorgente();
else
  visitedHost.add(self);

localData = RemDir.get(query);

foreach (Resource res : localdata)
{
  if (query.match(res))
    data.results.add(res);

  if (data.results.size >= query.getMaxResult())
    invia_risultati_sorgente();
}

if (self > keyMax)
  invia_risultati_sorgente();

nextHop = get_immediate_successor();
nextMsg = create_message(RECURSIVE_GET, data);
send(nextHop, nextMsg);
```

Figura 5.9: Pseudo codice della risoluzione di una query multi attributo sul nodo destinatario

contenuto in *keyMax*. Se è maggiore od uguale significa che tutto l'arco delle possibili soluzioni è stato controllato, al che i risultati sono inviati al nodo provider.

Nel caso in cui nessuna delle condizioni descritte sopra porti alla terminazione della fase di inoltro del messaggio, è utilizzato il livello sottostante di OW per recuperare le informazioni sull'immediato successore a cui è inviato un messaggio come quello descritto nel paragrafo precedente al quale sono state aggiunte le strutture dati modificate.

Capitolo 6

Risultati sperimentali

In questo capitolo sono presentati una serie di test sperimentali. Nel primo paragrafo sono descritti gli esperimenti fatti per confrontare le prestazioni di Overlay Weaver con quelle di Bamboo.

Successivamente sono stati effettuati dei test con Remedy per verificare la correttezza dell'implementazione e i possibili benefici, derivanti dalla diminuzione del numero di messaggi durante la fase di pubblicazione

L'ultima parte del capitolo prova a dare una valutazione, basata su rilevazioni di workstation, di uno scenario di lavoro plausibile per Remedy. Lo scopo è quello di capire che prestazioni dobbiamo aspettarci utilizzando le ottimizzazione con condizioni reali.

6.1 Confronto fra Overlay Weaver e Bamboo

Dopo aver studiato a fondo il codice di Overlay Weavr e gli strumenti che mette a disposizione, in questo paragrafo sono studiate le sue prestazioni. Come punto di riferimento sono stati considerati test effettuati con Bamboo [13].

6.1.1 Organizzazione dei test

Sono stati utilizzati due diversi ambienti di sperimentazione:

- Pianosa: è un cluster omogeneo composto di 32 nodi, in cui si sono svolti i test su scala più piccola.
- Grid5000 [7]: è una Grid riconfigurabile, controllabile e monitorabile composta da cluster eterogenei. Grid5000 è geograficamente distribuita sulla superficie della Francia su nove siti, ognuno composto da uno o più clusters. Ogni sito è accessibile attraverso un nodo di interfaccia su cui si utilizzano tool per riservare, utilizzare e monitorare i cluster. Più dettagli sulle modalità di accesso a grid5000 sono presenti nell'appendice B.0.1.

I test sono stati svolti studiando le prestazioni su di un cluster omogeneo connesso da una LAN Ethernet (quale è Pianosa), in seconda istanza sono stati ripetuti su Grid5000 per osservare i comportamenti su una scala più ampia.

Da sottolineare il fatto che Grid5000 non è una piattaforma dedicata in quanto altre applicazioni possono essere eseguite su nodi differenti ma dello stesso cluster. Essa è anche un banco di prova impegnativo per un applicazione distribuita, data la sua estensione geografica, la grandezza e la sua non omogeneità.

Il fatto di aver scelto una piattaforma come Grid5000 si è rivelato prezioso, non solo perché ha permesso una verifica del sistema su larga scala ma ha anche dato la possibilità di scontrarsi con i problemi relativi al deployment di software nel mondo reale. Un esempio è stato quello di assicurare il corretto funzionamento di OW su tutti i cluster di Grid5000. OW è scritto in Java ed essendo un progetto molto recente (ed aggiornato spesso) ha bisogno della ultima versione di Java per poter essere eseguito. I siti che formano Grid5000 sono gestiti distintamente: la disponibilità software varia da sito a sito, non solo per quanto riguarda le versioni, ma anche la loro presenza. Risolvere il problema non è stato difficile, in quanto è bastato installare l'ultima versione della JVM (Java Virtual Machine) su tutti i siti, ma dimostra come l'avvalersi di una piattaforma reale renda più macchinosi i test.

Su ogni nodo coinvolto nella fase di test sono in esecuzione:

- Un istanza di Overlay Weaver.

- Un istanza di Monitor, un demone che pubblica lo stato della macchina ogni 30 secondi.

In particolare con OW si è utilizzato come algoritmo Pastry, in modo da rendere più significativo il confronto con Bamboo sul quale erano in esecuzione istanze di programmi nella sostanza analoghi a quelli da me utilizzati con OW.

L'obiettivo del primo test di scalabilità è misurare il tempo di risposta (latenza) di una GET fatta da un singolo nodo, chiamato *Requester*. Quello del secondo è valutare il tempo di risposta della GET aumentando il numero di Requester. Nel terzo test si valuta l'affidabilità del sistema, facendo una richiesta di tutte le chiavi possibili, dopo aver ucciso un certo numero di nodi casuali. In tutti i test eseguiti il grado di replicazione dei dati è impostato a 4.

6.1.2 Primo test di scalabilità

Come detto prima lo scopo del test è quello di misurare le variazioni nel tempo di risposta (latenza) della GET al variare della dimensione della rete. Il Requester invoca periodicamente il metodo GET, con la chiave scelta casualmente fra l'insieme di nodi connessi alla rete.

Questo esperimento è stato ripetuto in due modalità, variando la distanza in termini di tempo tra una richiesta e la successiva da 11 a 27 secondi. Si sono scelti questi valori in quanto minori di 30 secondi (l'intervallo del Monitor) e consoni alle dimensioni della rete di overlay.

Dalla figure 6.1 e 6.2 si può notare come l'andamento medio della latenza mostri un risultato sulla scalabilità pressoché identico per entrambe le overlay. Tuttavia nei primi test su scala ridotta overlay weaver ha mostrato prestazioni leggermente migliori, in media circa 10 millisecondi sulla latenza della GET.

Le figure 6.3 e 6.4 mostrano i risultati ottenuti nelle stesse condizioni su Grid5000, utilizzando un numero più elevato di nodi.

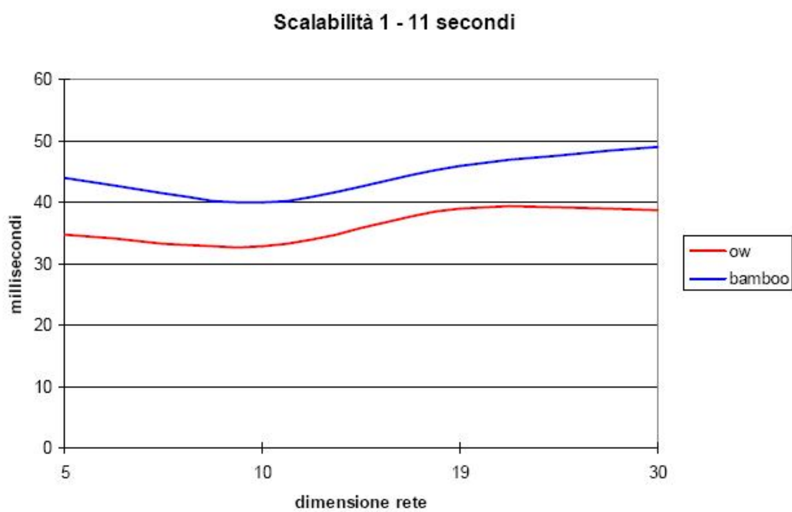

Figura 6.1: Primo test di scalabilità su Pianosa con GET ogni 11 secondi

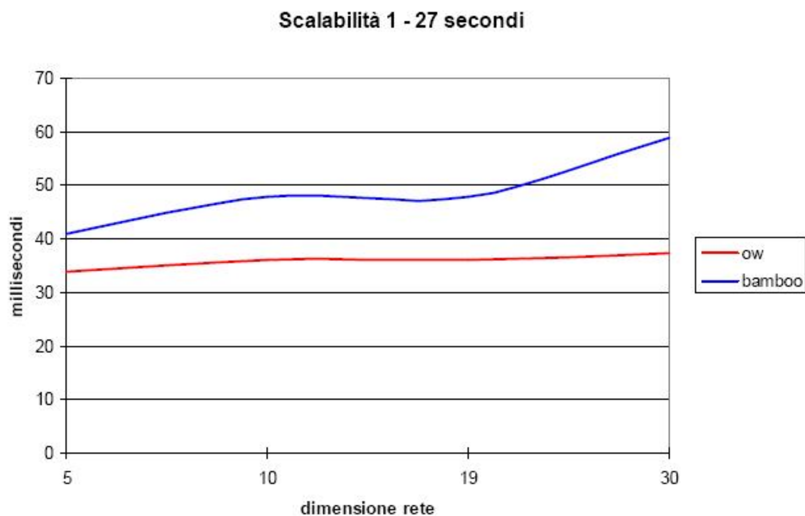

Figura 6.2: Primo test di scalabilità su Pianosa con GET ogni 27 secondi

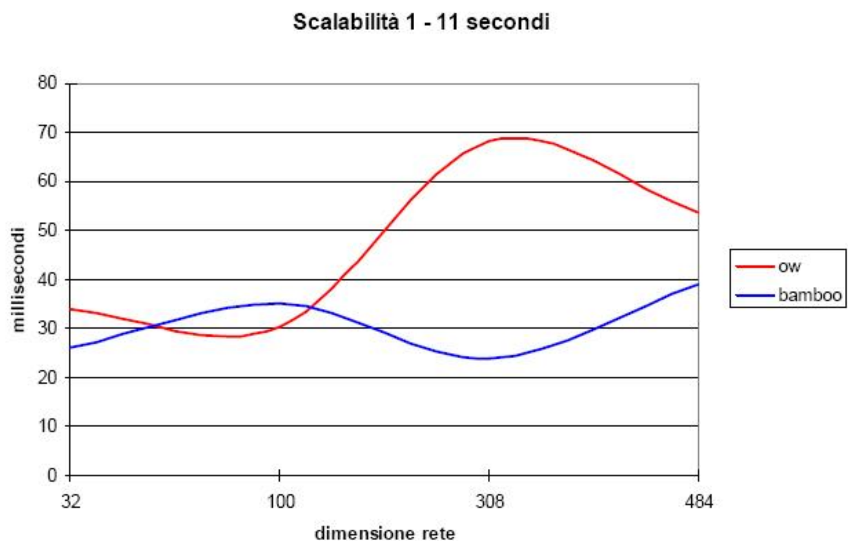

Figura 6.3: Primo test di scalabilità su Grid5000 con GET ogni 27 secondi

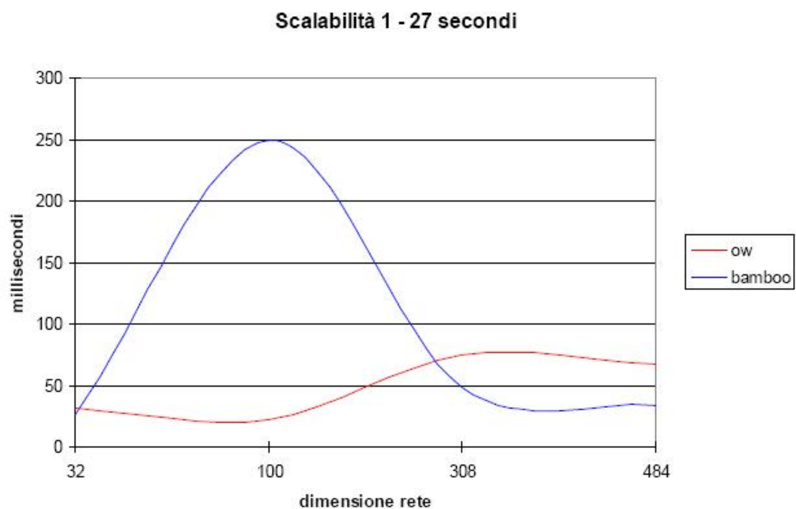

Figura 6.4: Primo test di scalabilità su Grid5000 con GET ogni 27 secondi

Con una rete di dimensione maggiore si nota come OW scali meno rispetto a Bamboo, ottenendo risultati, sia nelle misure con intervalli di 27 sia di 11 secondi, dell'ordine dei 60-70 millisecondi contro i circa 50 di Bamboo.

Inoltre è da sottolineare come le rilevazioni su Grid5000 risultino meno lineari rispetto alle corrispettive su Pianosa: la motivazione è da ricercarsi nelle caratteristiche stesse dell'ambiente (cluster non omogenei) e nel maggior numero di nodi coinvolto nell'esperimento.

Tuttavia la migliore connettività garantita da Grid5000, unità a dei cluster sicuramente più avanzati a livello di architettura rispetto a Pianosa, permetto alla Grid di avere prestazioni in assoluto migliori quando confrontabili: il tempo di risposta medio di una GET su Grid5000 con un rete di 32 nodi è migliore rispetto a quello con 30 nodi su Pianosa.

6.1.3 Secondo test di scalabilità

Su ogni nodo utilizzato nel secondo test di scalabilità, come nel precedente esperimento, è attivo il demone Monitor ma questa volta è aumentato il numero dei Requester.

- Pianosa: 5, 10, 15 Requesters con dimensione della rete fissa a 32 nodi.
- Grid5000: il 33% dei nodi che appartengono all'overlay funzionano come Requester

Le figure 6.5 e 6.6mostrano il tempo di risposta della GET su Pianosa. In entrambi i casi la latenza media è per entrambe le DHT sui 50 millisecondi. La dimensione della rete è troppo contenuta per osservare differenze rilevanti.

La figure 6.7 e 6.7 mostrano i tempi ottenuti su Grid5000.

Anche questo secondo test evidenzia la migliore scalabilità ottenuta da Bamboo rispetto a OW con reti più ampie. A riprova questo fatto partecipano in maniera molto significativa i risultati ottenuti su Grid5000, che mostrano chiaramente che le distanze aumentano con un numero di peers (e quindi di Requester) superiore alle centinaia.

Figura 6.5: Secondo test di scalabilità su Pianosa con GET ogni 11 secondi

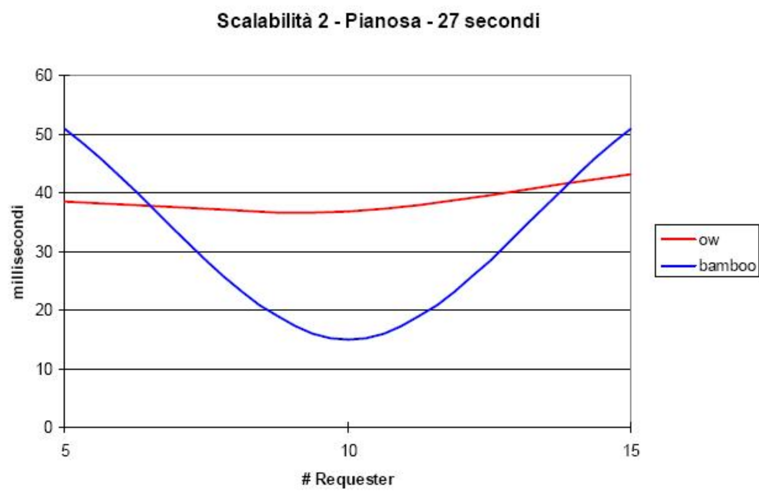

Figura 6.6: Secondo test di scalabilità su Pianosa con GET ogni 27 secondi

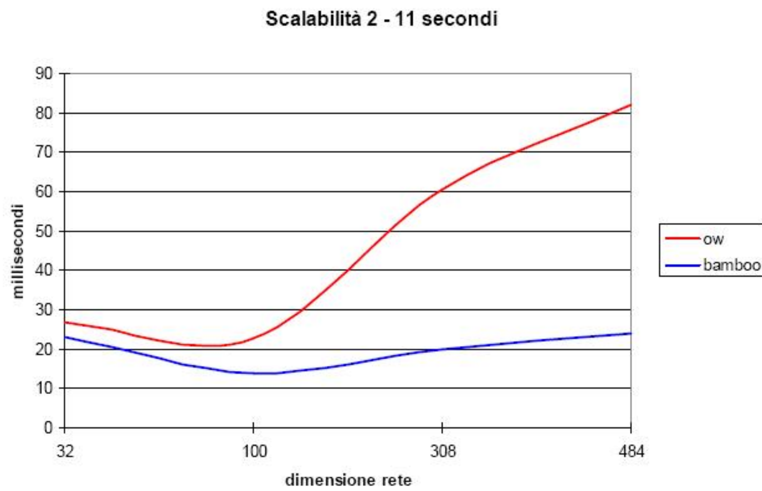

Figura 6.7: Secondo test di scalabilità su Grid5000 con GET ogni 11 secondi

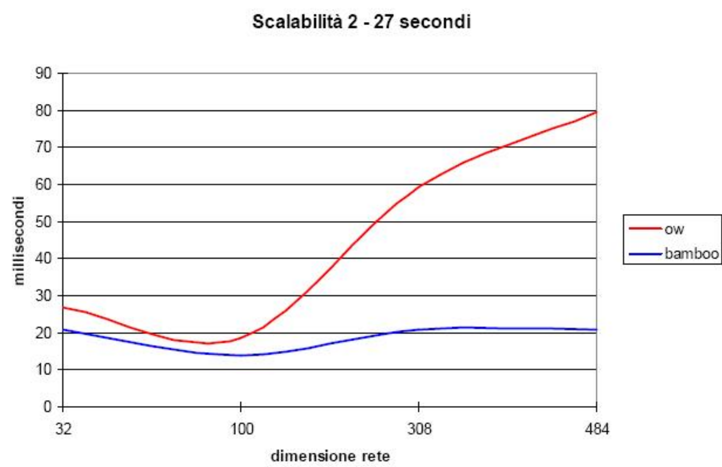

Figura 6.8: Secondo test di scalabilità su Grid5000 con GET ogni 27 secondi

6.1.4 Test di Affidabilità

L'approccio al test di affidabilità è differente rispetto ai precedenti, in quanto si vuole considerare il comportamento dell'overlay simulando la caduta di alcuni peers.

In una situazione reale questo è uno scenario plausibile. Con reti in larga scala, uno o più cluster possono interrompere il funzionamento per, ad esempio, un improvviso blackout o lavori sulla rete elettrica.

La procedura di verifica consiste nel provare a recuperare tutti i valori (lo stato di tutti i peers) sparsi per al rete, in altre parole si chiede di eseguire una GET per ogni chiave. Questa richiesta è una prima volta per verificare la correttezza e il funzionamento normale della rete. Successivamente una parte dei nodi, scelti casualmente, vengono uccisi. Si esegue quindi una altra richiesta immediatamente, per verificare la percentuale di chiavi perse subito dopo la caduta e si aspetta l'eventuale aggiornamento da parte di nodi che sono ancora vivi (in accordo con le specifiche di monitor circa 30 secondi) prima di ripetere l'ultima richiesta.

Il numero di nodi uccisi varia a seconda della dimensione della rete

- Pianosa: 5, 13, 20 nodi uccisi (dimensione della rete fissa a 32);
- Grid5000: uccisi il 16%, 50% e il 66% dei nodi appartenenti alla rete (32, 100, 308 e 484).

Dato che il grado di replicazione impostato sulle DHT è pari a 4, la risorsa associata alla chiave si perde solamente se tutti i nodi che contengono una copia sono uccisi, e se viene ucciso anche il nodo che pubblicava la risorsa. Per questo motivo l'ultima richiesta di tutte le chiavi ne recupera in numero maggiore, in quanto l'origine di alcune è ancora in esecuzione.

I test di affidabilità (Figure 6.9, 6.10, 6.11, 6.12 e 6.13) hanno un comportamento simile sia su Pianosa che su Grid5000 con entrambe le DHT mostrando un buon livello di affidabilità, impostando un grado di replicazione relativamente basso (4). Da sottolineare che in tutti i test qualunque sia il numero di nodi uccisi, le chiavi disponibili restano comunque sopra il 70% del totale.

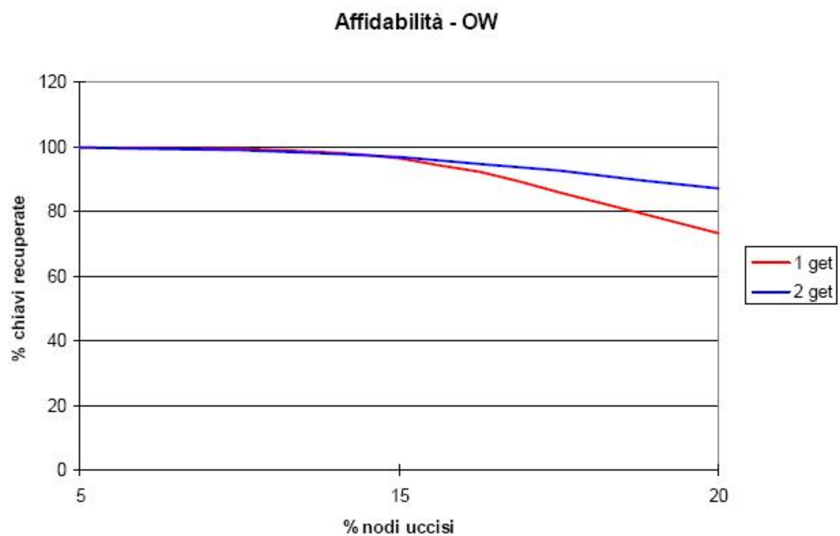

Figura 6.9: Test di affidabilità su Pianosa con Overlay Weaver

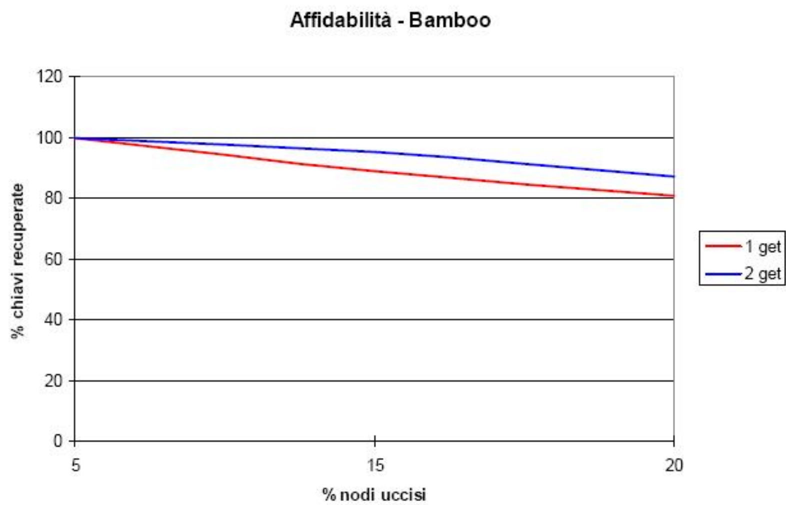

Figura 6.10: Test di affidabilità su Pianosa con Bamboo

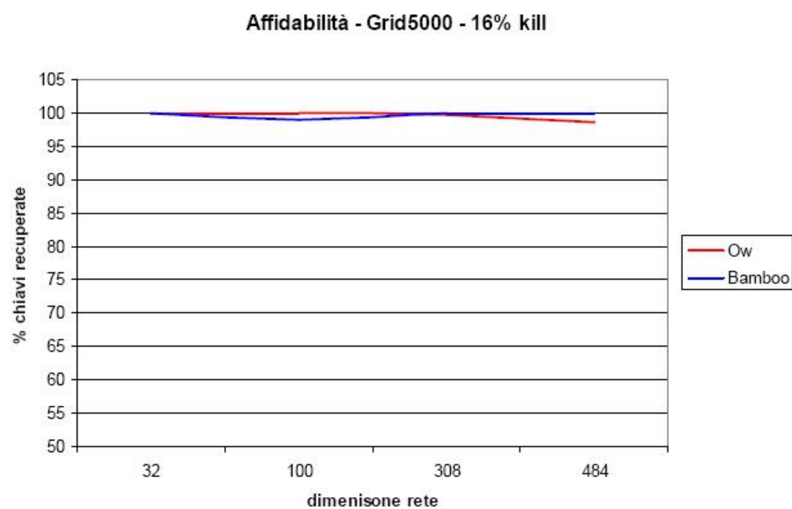

Figura 6.11: Test di affidabilità su Grid5000 con il 16% di nodi uccisi. Si è considerata solo l'ultima serie di GET.

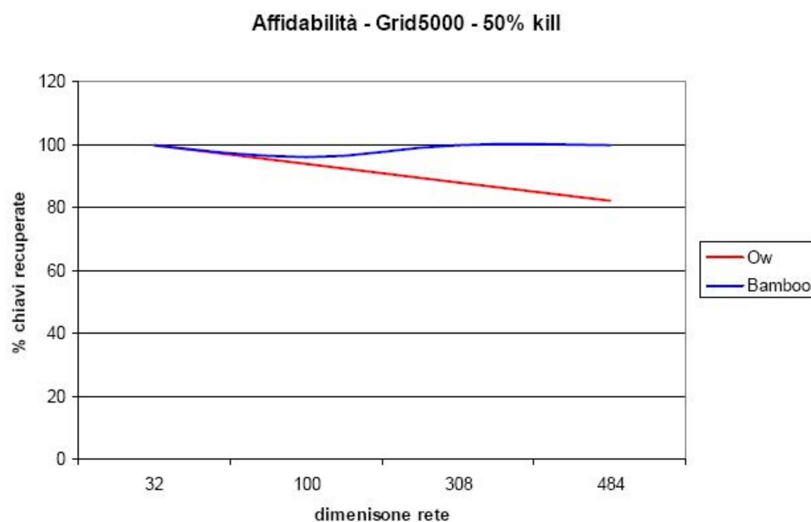

Figura 6.12: Test di affidabilità su Grid5000 con il 50% di nodi uccisi. Si è considerata solo l'ultima serie di GET.

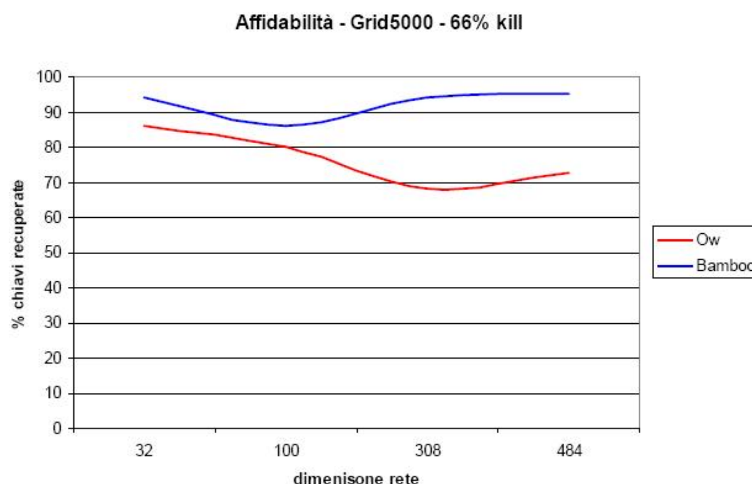

Figura 6.13: Test di affidabilità su Grid5000 con il 66% di nodi uccisi. Si è considerata solo l'ultima serie di GET.

6.1.5 Conclusioni

In questo capitolo abbiamo valutato le prestazioni di due differenti implementazioni di overlay: OveralyWeaver e Bamboo.

I risultati mostrano che Bamboo può essere ritenuto scalabile sia aumentando la dimensione della rete, che il numero dei Requester. OW dalla sua, pur avendo prestazioni confrontabili con Bamboo nel primo caso, ha un degrado in termini di latenza quando si tratta di dover sopportare un numero maggiore di traffico (GET) fra i suoi nodi.

La differenza in termini di prestazioni, se pur presente e misurabile, permette comunque un confronto alla pari in quanto i risultati rimangono sullo stesso ordine di grandezza (decine di millisecondi).

Su fronte dell'affidabilità gli esperimenti dimostrano che entrambe le DHT capaci di sopportare relativamente bene una grossa mancanza della rete. Il fatto che in tutte le configurazioni uccidendo il 66% dei nodi non si recupera mai meno di circa il 70% del numero totale di chiavi, è indicativo.

Rispondere alla domanda su quale tra Bamboo e OW sia migliore non è immediato, ognuno ha delle frecce al proprio arco. In generale a livello

di prestazioni Bamboo sembra un gradino superiore a OW. Quest'ultimo ha però un architettura modulare che consente lo sviluppo di nuove soluzioni o l'implementazione di vecchie con uno sforzo più basso, è più recente e viene aggiornato spesso. Questo aspetto è comunque molto rilevante, specie quando lo strumento deve essere personalizzato per venire incontro a esigenze specifiche. Il costo di una modifica è molto più alto su Bamboo, cosa che ne pregiudica l'utilizzo in situazioni in cui l'adattabilità e la modularità sono requisiti essenziali

Entrambe le overlay comunque non forniscono funzionalità utili a risolvere le range query. Questo problema sarà approfondito nei capitoli successivi.

6.2 Remedy: risultati sperimentali

Dopo aver studiato l'approccio in teoria, in questo paragrafo studiamo le prestazioni di Remedy su una rete reale. Le prove sono state eseguite utilizzando l'implementazione, descritta in 5, di Remedy su Overlay Weaver.

6.2.1 Organizzazione dei test

I test sono stati eseguiti nei due ambienti già descritti in precedenza: Pianosa e Grid5000.

Il modello di prestazione utilizzato per i test è la latenza, in particolare quella relativa alla risoluzione di query con selettività variabile. Nei paragrafi successivi, con il termine generale *prestazione* si indicherà questo parametro.

In particolare le prove effettuate hanno l'obiettivo di valutare le prestazioni di MAAN in base a due parametri significativi: la selettività di una query, e il numero di attributi utilizzati. Il numero di nodi da contattare ha sicuramente una certa influenza sul tempo di risposta di una query. Un altro fattore è la quantità di messaggi che vengono scambiati tra i nodi per pubblicare le risorse, i quali inevitabilmente rallentano anche le prestazioni delle query. Inoltre è stato eseguito un ulteriore test di misura del numero di hops nella fase di risoluzione di una query, per verificare l'aderenza dell'implementazione con i risultati teorici.

Su ogni nodo della rete è in esecuzione, oltre ad un istanza di OW, un demone che periodicamente esegue la pubblicazione di una risorsa composta da coppie attributo valore. L'intervallo fra due pubblicazioni successive è impostato a 30 secondi. Questo intervallo è decisamente breve se paragonato a quello impostato su reti che affrontano il problema del Resource Discovery, ad esempio 5 minuti in [10], ma permette di studiare le prestazioni di una rete sotto pressione simulando una dinamicità elevata dell'ambiente.

6.2.2 Numero di nodi visitati nella risoluzione di query

Questo primo test ha come scopo quello di verificare il numero di nodi contattati al variare della selettività della query, per capire come l'implementazione si avvicini alle prestazioni studiate in teoria. Ricordiamo che utilizzando la risoluzione con singolo attributo il numero di nodi da attraversare è pari a $(logN + N \times s_k)$, dove con N si indica la dimensione della rete e con s_k il grado di selettività del k-esimo attributo.

Questo test è stato effettuato utilizzando l'emulatore presente in overlay weaver. Un singolo nodo sottomette una query al sistema. La selettività di questa query è di volta in volta variabile in base alle esigenze dei test.

La prima prova è quella di verificare, all'aumentare della dimensione della rete, se il numero di nodi contatti varia come ci si aspetta. Sono state fatte due prove, una con selettività di circa il 10% e la seconda con una selettività vicina allo 0%.

I risultati sono visibili in figura 6.14. Come si può verificare, a meno di qualche imperfezione nelle curve dovuta agli arrotondamenti, l'andamento teorico è stato rispettato in entrambi i casi.

Il secondo test misura il numero di nodi contattati al variare della selettività con la dimensione della rete fissa a 256 nodi. I risultati sono visibili in figura 6.15.

Anche in questo test i risultati sperimentali rispecchiano le previsioni studiate in teoria.

In conclusione, si può affermare che, dal punto di vista dei nodi contattati, l'implementazione fornita è in accordo con i risultati teorici.

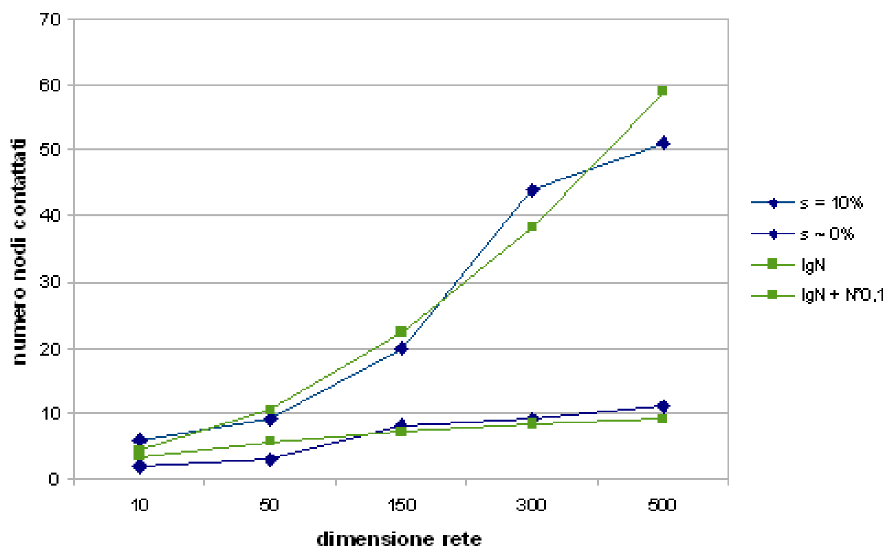

Figura 6.14: Numero di nodi contattati con selettività fissa all'aumentare della rete.

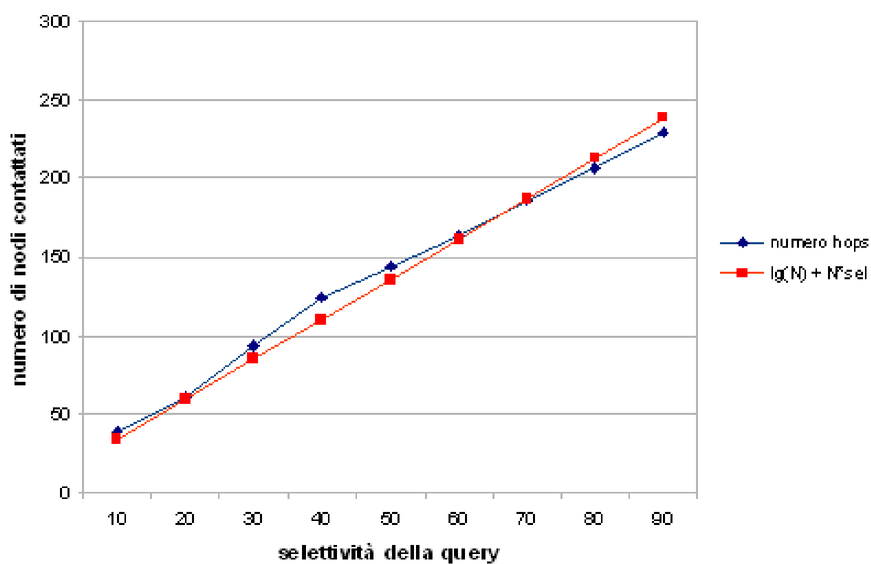

Figura 6.15: Numero di nodi contattati all'aumentare della selettività con rete da 256 nodi.

6.2.3 Test sulla selettività

Questo test ha come obiettivo quello di studiare la latenza al variare della selettività della query, che ricordiamo essere quella dell'attributo dominate (quello con selettività minore).

Sono state considerate reti con un diverso numero di nodi: 5, 15, 30, 50, 100 e 300 nodi. Per ognuna di queste configurazioni sono stati effettuati test con query con selettività variabile dal 10% al 70%. Il numero di attributi per la descrizione di una risorsa è impostato fisso a 5.

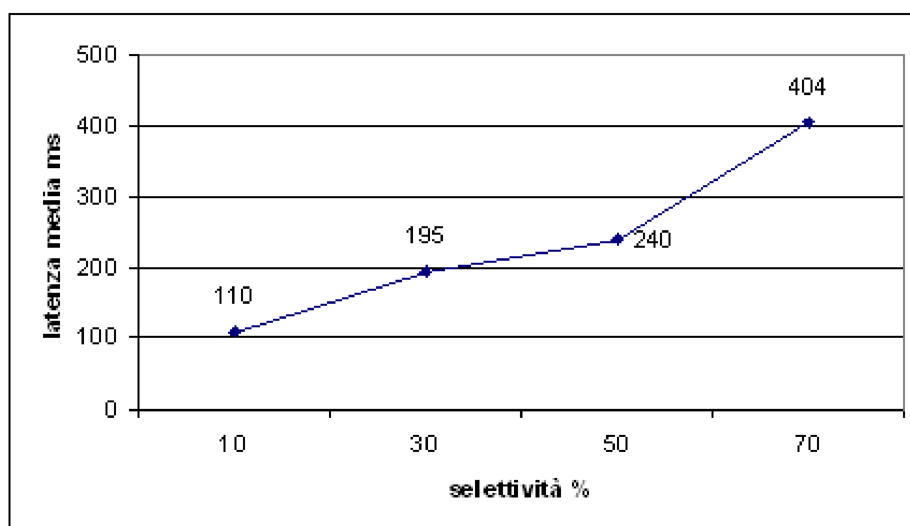

Figura 6.16: Latenza media su Pianosa, con rete da 30 nodi

In figura 6.16 sono sono presenti le latenze medie per la risoluzione di query su Pianosa, utilizzando tutti e 30 i nodi disponibili. Come si vede le prestazioni rimangono in generale accettabili, ma il decadimento quando si interrogano più del 50% dei nodi è evidente.

Verosimilmente, all'aumentare della selettività le prestazioni tendono a peggiorare in quanto sale linearmente il numero di nodi interrogati.

A dispetto del maggior numero di nodi, le prestazioni su Grid5000 (figura 6.17) hanno lo stesso andamento rispetto a quelle eseguite su Pianosa. Da un punto di vista quantitativo sono comunque migliori rispetto a

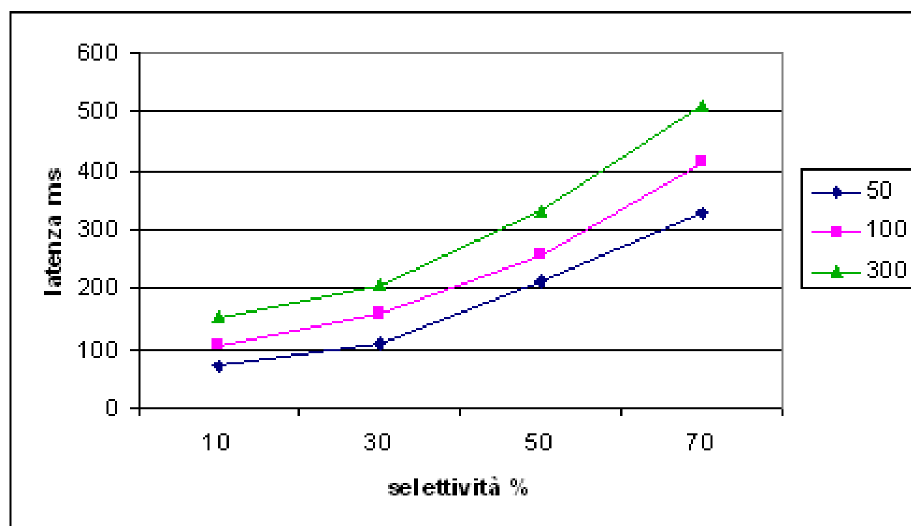

Figura 6.17: Latenza media di una query su Grid5000 con reti da 50,100 e 300 nodi

quelle di Pianosa, in quanto influenzate dal maggior livello tecnologico della piattaforma.

6.2.4 Test sul numero di attributi

In MAAN il numero di messaggi necessari a pubblicare una risorsa è direttamente proporzionale al numero di attributi necessari a descrivere una risorsa. In altre parole tanto più alto è il numero degli attributi, tanto maggiore è il traffico di rete generato. Lo scopo di questo test è verificare l'impatto che una riduzione del traffico dovuto alla fase di pubblicazione ha sulle prestazioni.

Le dimensioni delle reti utilizzate sono analoghe al test precedente. In questa prova a variare sono il numero di attributi utilizzati da uno fino a cinque. La selettività è costante al 20%.

In figura 6.18 è rappresentato l'andamento della latenza su Pianosa. Rispetto al caso iniziale di cinque attributi si ha un miglioramento delle prestazioni del 20% passando ad una risorsa con quattro attributi. Riducendo di un unità ancora il numero degli attributi le prestazioni migliorano di un ulteriore 10%, per poi nelle successive riduzioni stabilizzarsi su valori costanti.

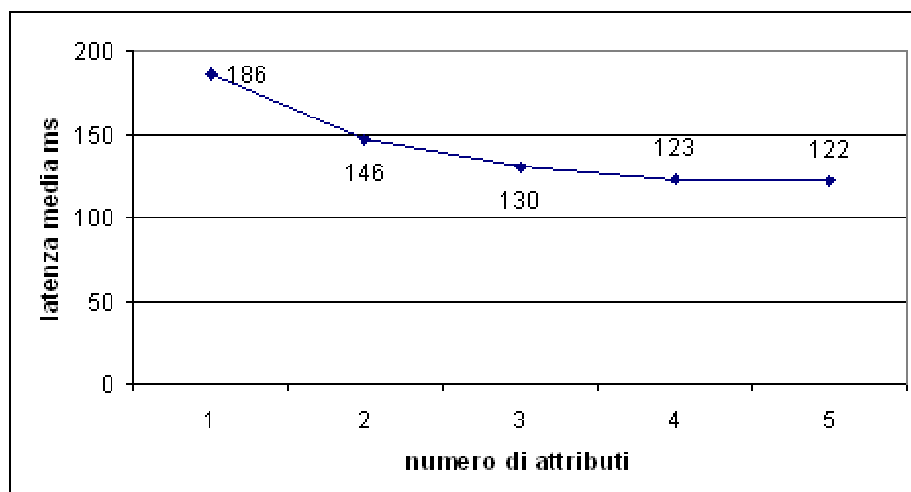

Figura 6.18: Latenza al variare del numero di attributi con un rete da 30 nodi su Pianosa

Lo stesso andamento è visibile su una rete più ampia in figura 6.19. Si noti che più la rete cresce, e di conseguenza, più il traffico dovuto alla pubblicazione aumenta, la riduzione del numero degli attributi ha un impatto maggiore sulle prestazioni.

6.2.5 Conclusioni

In questi test ho analizzato come fattori quali la selettività e il numero di attributi influiscono sulle prestazioni di MAAN. Per quanto riguarda la selettività, MAAN è un sistema che funziona al meglio quando la selettività di una query non è troppo elevata. Per reti di grandi dimensioni è necessario limitare al più possibile la selettività o quantomeno limitare il numero di nodi interrogati, basandosi sulle dimensioni presunte della rete. Questo può essere ottenuto limitando esplicitamente il numero di nodi contattati e/o limitando superiormente il numero di risultati che si vogliono restituire. Operazioni di questo tipo sono strettamente legate alle applicazioni in cui il RD è utilizzato e, per questo, difficilmente generalizzabili.

Il numero di attributi per la descrizione di una risorsa influenza le prestazioni, specie quando il traffico generato è relativamente alto. La diminuzione

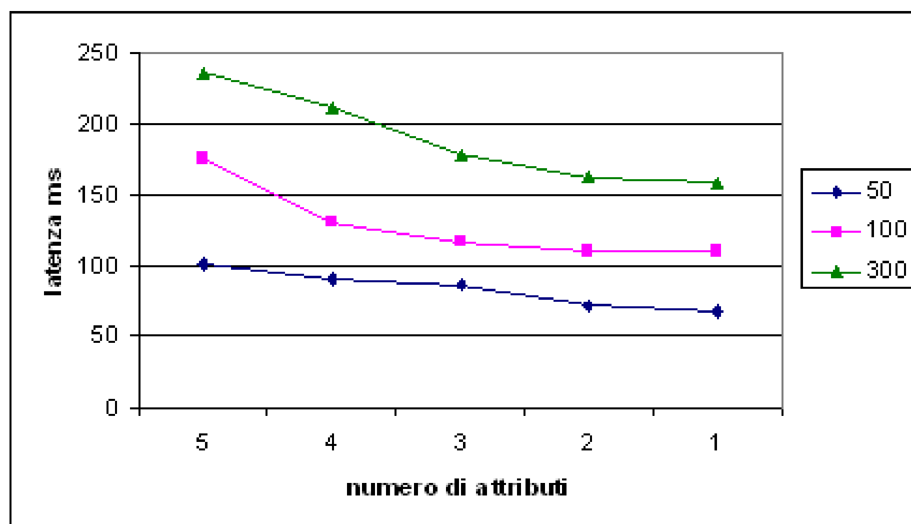

Figura 6.19: Latenza al variare del numero di attributi con reti da 50, 100 e 300 nodi su Grid5000

del numero degli attributi ha un effetto migliorativo e permette di ridurre in alcuni casi le latenze di un 50%. Dai grafici è anche chiaro come la latenza media per la risoluzione di una query abbia il suo limite inferiore. Questo dipende sicuramente dall'algoritmo utilizzato e dal livello tecnologico della piattaforma ed è indipendente dal traffico generato per gli aggiornamenti.

6.3 Analisi di dati reali

Remedy, come abbiamo visto, è un approccio per l'ottimizzazione del traffico dovuto alla pubblicazione di risorse caratterizzate da attributi dinamici e basato sulla valutazione dei seguenti parametri:

- la probabilità che due misurazioni successive di un attributo abbiano un valore gestito dalla stesso nodo della DHT.
- la probabilità che una chiave sia gestita dallo stesso nodo fra due misurazioni successive.
- la probabilità che un attributo abbia bassa popolarità in un determinato istante.

L'obiettivo dei prossimi paragrafi è di provare a dare una valutazione quantitativa a questi parametri.

6.3.1 PlanetLab

Iniziamo prendendo in considerazione il primo parametro. Per darne una valutazione quantitativa sono necessari dati sul comportamento di un insieme di hosts appartenenti ad una rete reale per un certo periodo di tempo. Questa misurazioni non possono essere fatte su Grid5000. Esso è un sistema in cui l'utilizzo di risorse, tranne la rete, è esclusivo per l'utente che le ha prenotate è quindi non può rappresentare in modo ideale lo stato dei nodi.

Ho scelto quindi di utilizzare dati provenienti PlanetLab. PlanetLab [19] è un insieme di macchine disponibili come piattaforma per il testing e il lavoro di ricerca su applicazioni di rete distribuite. Ogni progetto è rappresentato da uno *slice*, cioè un accesso tramite macchina virtuale ad un sottoinsieme di nodi; è importante sottolineare che per ogni macchina possono esserci più slices attivi, rendendo di fatto le misurazioni più eterogenee, ma per questo più realistiche, possibili. Oltre che per scopi di ricerca su PlanetLab sono in esecuzione servizi pubblici come OpenDHT [23] o CoDeeN [22]. In particolare quest'ultimo è un Content Distribution Network, in altre parole una rete di computer per la distribuzione di contenuti. All'interno di questo progetto è presente CoMon [21] un sotto progetto che si occupa di fornire informazioni sullo stato delle macchine che compongono la rete di PlanetLab. Le rilevazioni fatte da CoMon riguardano tutti i nodi presenti su PlanetLab e sono eseguite con un intervallo di cinque minuti. Ho utilizzato quindi alcuni delle rilevazioni di questo progetto disponibili in rete.

Altri studi si sono soffermati nello studiare questi dati, in particolare [20]. Questo studio ha esaminato le risorse di PlanetLab dalla prospettiva di chi progetta un sistema di resource discovery. In particolare si sono soffermati sulla variabilità dei dati nel tempo e le loro correlazioni fra i nodi. Questi dati sono stati ottenuti con strumenti tra cui CoMon e Ganlia [31]. L'obiettivo finale è quello di valutare l'importanza di un sistema di resource discovery per l'esecuzione di una applicazione. Per alcune risorse, la quantità disponibile in

un certo momento varia significativamente fra i nodi. D'altra parte, per molti nodi, la disponibilità di risorse varia nel tempo, rendendo plausibile soluzioni in cui le istanze di applicazioni *migrano* verso nodi che ne garantiscono una migliore esecuzione. Uno dei risultati ottenuti da questa ricerca afferma che nel breve periodo la variazione degli attributi dinamici è minore rispetto al lungo periodo. In altre parole per periodi relativamente brevi gli attributi, pur essendo dinamici, non cambiano. Questa è, a tutti gli effetti, una delle ipotesi su cui si basa Remedy.

Elaborazione dei dati

Ho utilizzato le misurazioni di sei diversi attributi dinamici: la disponibilità di memoria, lo spazio di swap disponibile, il numero di bytes al secondo ricevuti ed inviati sulla rete ed infine il carico e l'utilizzo del processore.

In figura 6.20 è presente un esempio di come le misurazioni sono memorizzate. Il carico del processore e la disponibilità di memoria sono indicati rispettivamente da *Loads* e *MemPress* (zona 1). Loads è la misurazione equivalente a quella dei sistemi Unix in cui è espresso il carico in termini di threads runnable a 1, 5 e 15 minuti. La disponibilità di memoria è l'output di un demone che periodicamente cerca di allocare 100MB di memoria sul nodo. Valori alti, ad esempio sopra i 90MB, indicano che il nodo non è sotto pressione dal punto di vista della memoria.

La quantità di in entrata ed in uscita sull'interfaccia di rete sono indicati rispettivamente da *Tx* e *Rx* (zona 2). I valori rappresentano i Kbps in uscita e in entrata. L'utilizzo del processore e le informazioni sulla memoria virtuale sono indicati da *CPUUse* e *VMStat* (zona 3). CPUUse differisce da Loads in quanto rappresenta, in termini percentuali, il tempo in cui il processore no si trova in stato di idle. Il primo valore (nell'esempio 10) è il tempo utilizzato dai processi utente, il secondo (21) il tempo totale, sommando quello utilizzato dai processi utente a quello dei processi di sistema. In VMStat sono presenti molte informazioni, corrispondenti all'analogo comando nei sistemi Unix. I tre valori evidenziati rappresentano rispettivamente: il totale della memoria virtuale usata, il totale della memoria virtuale libero, il totale della memoria

```
Start: 1212465902 Tue Jun  3 00:05:02 2008
Name: 146-179.surfsnel.dsl.internl.net
RespTime: 1.265235
VMStat: [..] 715044 106480 298728 [..]      3
CPUUse: 10 21
Uptime: 28701.20
Loads: 1.38 0.63 0.36      1
MemPress: 100
MemInfo: 1.97149 17.8528 0
KernVer: 2.6.12
Burp: 79.7%
CPUSpeed: 3.2014
TxRate: 170.000000      2
RxRate: 96.000000
NumSlices: 33 mostrecentcotop
LiveSlices: 3
IPAddr: 145.99.179.146
BootState: boot
NodeType: Prod
SiteName: cwi
BWLimit: none
Latitude: 52.22
Longitude: 4.53
Location: Europe
Drift: -0.894305
```

Figura 6.20: Alcuni dei dati rilevati da CoMon

virtuale usata come buffer. Per la mia analisi ho utilizzato il secondo di questi valori.

Oltre agli attributi presi in considerazione per il mio studio ne sono presenti molti altri, che si riferiscono a caratteristiche diverse, anche statiche. Fra questi cito, a titolo di esempio, attributi statici quali la frequenza espressa in Mhz del processore, il numero di GB di memoria presenti e la latitudine e longitudine in cui è fisicamente locato il nodo.

Per elaborare i dati necessari a Remedy, ho preso in esame le rilevazioni effettuate da CoMon nell'arco temporale di una settimana di giugno 2008.

Per ogni nodo soggetto ad esame ho calcolato, per ognuno dei sei attributi, la frequenza con la quale la differenza fra due misurazioni non supera una certa soglia come accennato nel paragrafo 3.5.2. Per questo si valuta se qualitativamente la differenza fra due misurazioni successive rimane inferiore rispetto ad un valore soglia. Il significato è che piccole variazioni del valore di un attributo corrispondono sempre allo stesso nodo all'interno di una DHT. Nei miei test questa soglia ha assunto un valore di $\epsilon = 0,5\%$.

Successivamente ho calcolato la media di queste frequenze. I risultati sono visibili in figura 6.21.

Come si vede la disponibilità di memoria e l'utilizzo del processore sono attributi che presentano una frequenza di ripetizione molto alta. Questo è dovuto principalmente al fatto che i valori sono espressi in termini percentuali senza decimali, il che implica che il livello di granularità della misurazioni risulti relativamente alto.

Per altri attributi, ad esempio lo spazio swap disponibile, pur avendo fra misurazioni successive valori molto vicini in termini numerici, è altamente improbabile che i valori siano esattamente uguali. La frequenza di ripetizione di questi ultimi è infatti sensibile al cambiamento di ϵ, a differenza di attributi i cui valori sono espressi in percentuale.

La media globale è calcolata sommando tutte le frequenze e dividendo per il numero degli attributi. In conclusione il valore medio della frequenza

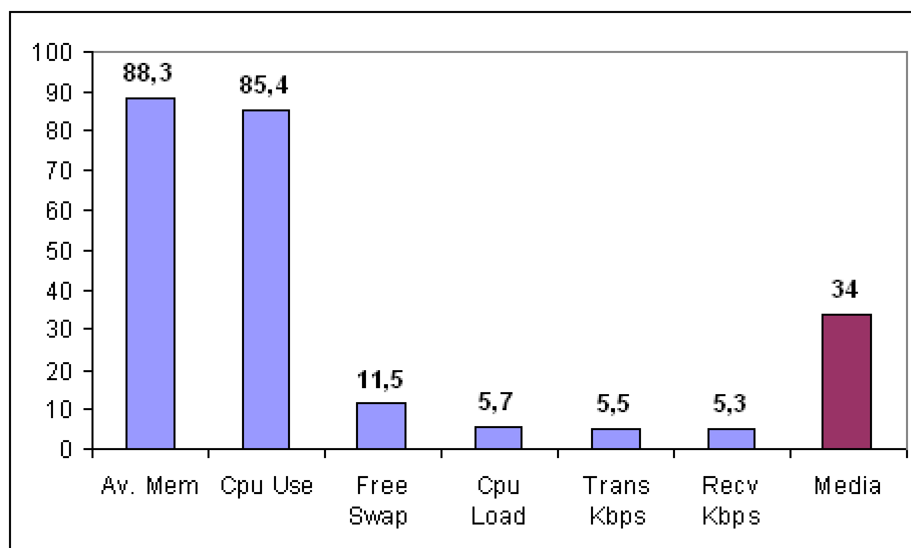

Figura 6.21: Frequenza di ripetizione degli attributi

di ripetizione di un attributo è di circa il 30%. In altre parole una volta su tre un attributo ha lo stesso valore rispetto alla misurazione precedente.

6.3.2 Distribuzione dei dati

Come discusso nel paragrafo 3.3 una funzione di hash specifica per ogni parametro e basata sulla sua distribuzione, aiuta a distribuire il carico in modo migliore sulla rete.

I dati di CoMon quindi sono stati utilizzati anche per ricavare la distribuzione dei valori di alcuni attributi. Gli attributi considerati sono gli stessi sei del paragrafo precedente, quindi il carico del processore sia in termini di processi che in termini di percentuali di utilizzo, una stima della disponibilità di memoria, lo spazio swap disponibile e la banda utilizzata in entrata ed in uscita.

In figura 6.22 è mostrato il grafico relativo alla distribuzione del tempo del processore. È interessante notare come i valori sia concentrati in un intervallo relativamente ampio all'inizio della curva (processore poco carico), mentre quando il processore è carico lo è sempre al massimo. Quest'ultima è di certo la situazione di gran lunga più frequente: per ragioni di presentazione

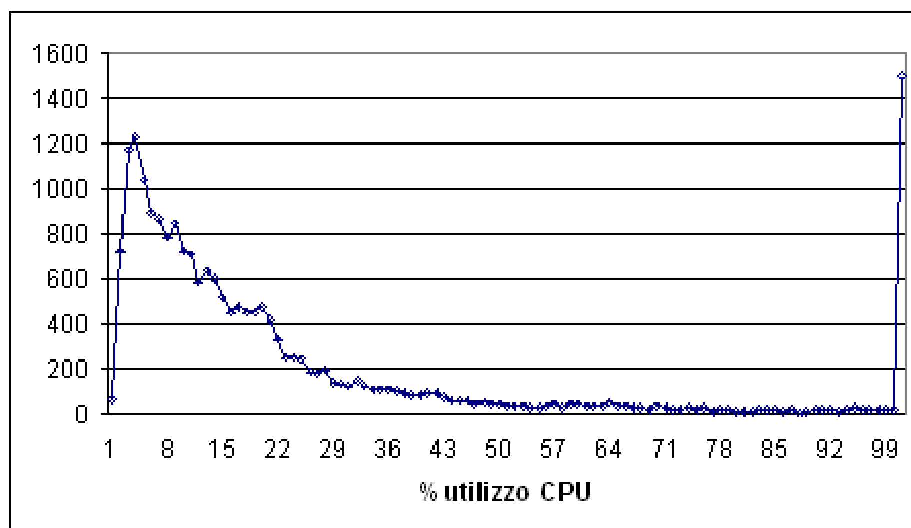

Figura 6.22: Distribuzione del tempo di utilizzo della CPU.

il numero di occorrenze relative al valore 100 è stato diminuito di due ordini di grandezza.

Sempre in relazione al processore in figura 6.23 è presentato il carico medio in termini di processi. Nella maggior parte dei casi il numero di processi attivi non supera le dieci unità.

Per quanto riguarda la disponibilità di memoria, i risultati sono presentati in figura 6.24. I dati sono tutti concentrati verso la parte finale della curva, il che indica sempre una immediata disponibilità di memoria, cosa plausibile per come è strutturato MemPress. Anche in questo caso, come per il carico del processore, il numero di occorrenze per il valore 100 è stato diminuito per rendere il grafico presentabile.

La distribuzione del numero di dati in uscita e in entrata è presente in figura 6.25.

Come si nota dai grafici l'andamento è piuttosto simile. La curva dei dati trasmessi si abbassa più rapidamente, ed è più irregolare rispetto a quella dei dati inviati.

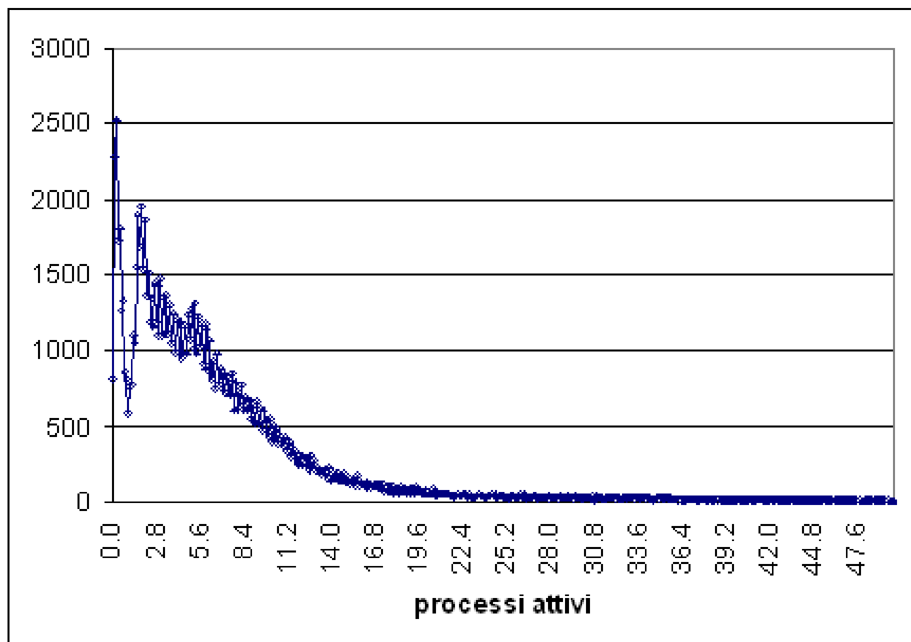

Figura 6.23: Distribuzione del numero di processi attivi sulla CPU.

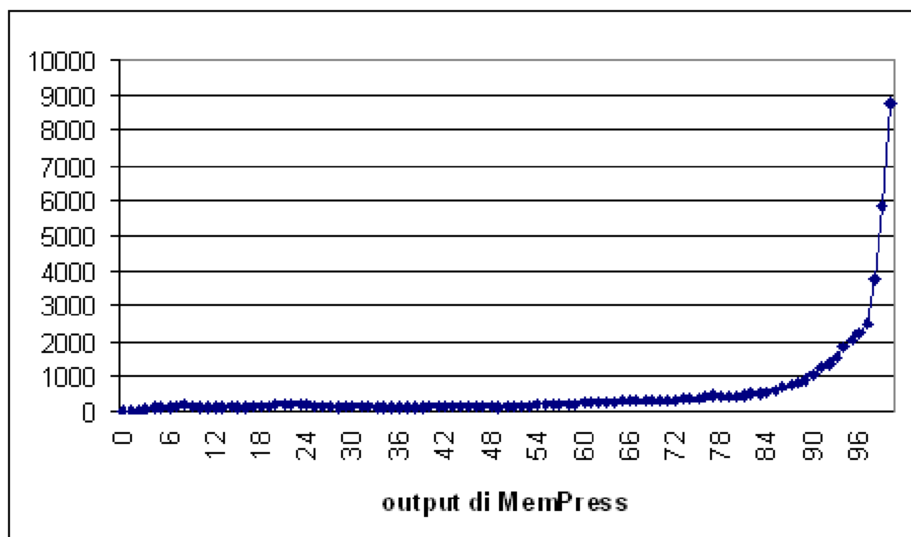

Figura 6.24: Distribuzione dell'output del programma MemPress.

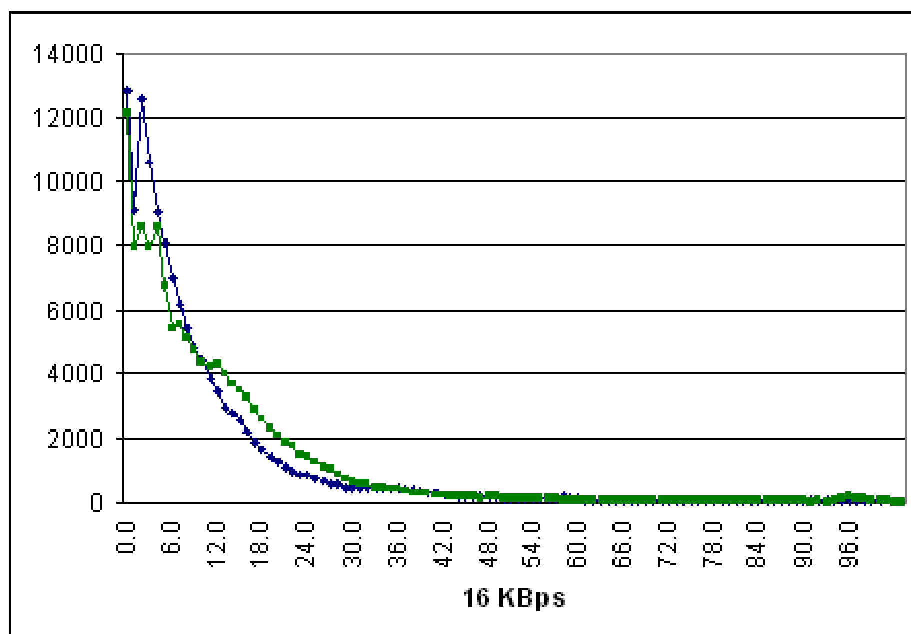

Figura 6.25: Distribuzione dei dati in uscita e in entrata con una risoluzione di 16 KBps

6.3.3 Churn

Con il termine *churn* si intende la modifica della topologia di una DHT dovuta all'inserimento o all'uscita di nodi dalla rete. In presenza di churn la DHT deve sostenere degli overhead, a partire dalla modifica delle tabelle di routing fino all'assegnare nuove risorse al nodo entrante (a toglierle al nodo uscente). Tutti questi fattori possono peggiorare le prestazioni della DHT.

Sono stati proposti modelli statistici [24] per calcolare in modo analitico il costo di questi overhead. Inoltre si è cercato di mitigare il problema studiando algoritmi per adattarsi dinamicamente al livello di churn [25] e DHT che provano a mitigarne l'effetto quando il flusso di nodi è troppo elevato [13].

Attualmente, esistono poche DHT utilizzate in applicazioni su grande scala ed è quindi difficile ottenere informazioni sui possibili livelli di churn in una rete. Tuttavia, come punto di partenza, una possibilità è quella di utilizzare come riferimento le reti P2P per la condivisione di file. Queste architetture utilizzano un sistema per indicizzare le risorse che può essere

paragonato a quello delle DHT.

Queste informazioni possono essere utili per trovare un limite superiore al livello di churn, in quanto in griglie computazionali i nodi sono più stabili e sono utilizzati esclusivamente per uno scopo preciso.

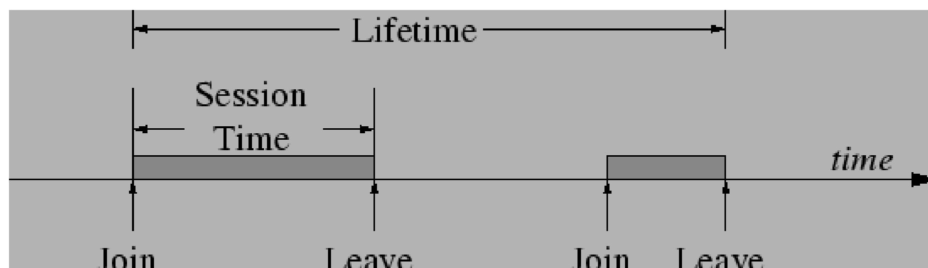

Figura 6.26: Rappresentazione grafica del churn

Con *session* si intende il tempo che intercorre fra un ingresso e un uscita della DHT, mentre con *lifetime* è inteso il tempo che intercorre fra il primo ingresso nella rete fino alla definitiva uscita.

Il parametro più importante è il tempo di sessione, in quanto anche una perdita temporanea di un collegamento può risultare negativa sulle prestazioni e sulla correttezza di una DHT, ad esempio forzando una richiesta a seguire un percorso non ottimale per arrivare a destinazione.

Ai fini di Remedy, è importante valutare la probabilità che nell'intervallo fra due misurazioni successive il nodo che gestisce una certa chiave nella DHT non cambi. Una chiave è gestita da un nodo diverso quando un nodo lascia (volontariamente o meno) o quando un nodo si unisce alla rete.

Supponiamo che il nodo è in qualche modo 'dedicato, cioè nel caso di uscita dalla rete (forzata o meno) si colleghi appena ha ripreso le funzionalità che ne hanno determinato l'uscita. Utilizzando lo schema in figura 6.26, significa avere un rapporto fra sessiontime e lifetime, vicino ad uno. Essendo gli ID dei nodi equiprobabili, un nodo cambia due volte chiavi gestite nel corso del session time, una volta entrando nella rete ed una volta uscendo.

Uno studio sui sistemi P2P di condivisione dei file [30] raggiunge la conclusione che in media in questo tipo di sistemi la durata del sessiontime è di circa 60 minuti.

Supponiamo di memorizzare una risorsa sempre alla stessa chiave, ripetendo l'operazione ogni 5 minuti (intervallo fra le misurazioni di CoMon). Se il tempo di sessione dura 60 minuti, ho in questo periodo che la chiave cambia di gestore due volte (una volta per l'uscita e una per l'entrata del nodo). Questo significa che per 1 aggiornamento su 6 la chiave è gestita da un nodo diverso. In termini percentuali questo si verifica quindi nel 16,6% dei casi. In generale questa probabilità potrebbe essere anche più bassa in quanto il cambio di gestione dovuto alla caduta e quello dovuto all'inserimento di un nuovo nodo potrebbero verificarsi nel corso dello stesso intervallo.

Dal punto di vista di Remedy interessava misurare la probabilità che una chiave fosse gestita sempre dallo stesso nodo, cioè l'inverso del valore appena calcolato che corrisponde all'83,4% dei casi.

Come già detto questo stima è per difetto in quanto basata su una rete P2P di condivisone di file per sua natura più dinamica di reti utilizzate per altri scopi, ma comunque utile per avere un'idea di quelli che possono essere comportamenti plausibili.

6.3.4 Funzioni Hash e bilanciamento del carico

Per poter risolvere le range query multi attributo Remedy sfrutta particolari funzioni di hashing che permettono di mantenere la località dei dati.

Come spiegato nel paragrafo 3.3 l'utilizzo di funzioni diverse può avere effetti su dove gli oggetti immessi nella rete siano effettivamente memorizzati, in altre parole effetti sul *bilanciamento del carico*. I tipi di funzione descritti sono sostanzialmente due:

1. $$H(v) = (v - v_{min}) \times (2^m - 1)/(v_{max} - v_{min})$$

2. $$H_i(v) = D_i(v) \times (2^m - 1)$$

I dati di CoMon offrono lo spunto per verificare quali sono realmente le distribuzioni dei principali attributi dinamici. Grazie a questi dati è stato

possibile implementare funzioni specifiche per ogni attributo, basandosi sulla sua distribuzione.

Dei test sono stati svolti per verificare l'impatto dei due tipi di funzioni hash sul bilanciamento del carico. Queste prove sono state effettuate considerando una rete di 100 nodi in cui, tramite una PUT, sono state inserite in cinque sessioni differenti, 100 risorse composte da un solo attributo. L'elenco degli attributi, uno per ogni sessione, è il seguente: la disponibilità di memoria calcolata tramite il demone MemPress, il numero di processi attivi (Load), i KBps trasmessi in uscita, lo spazio di swap libero e il tempo percentuale in cui il processore è occupato (CpuUse). I valori dell'attributo scelto di volta in volta sono stati calcolati in accordo con la sua distribuzione, indipendentemente dal tipo di funzione hash utilizzata.

Lo scopo di questo test è quello di verificare se effettivamente la distribuzione delle risorse utilizzando una funzione hash in cui compare la distrbuzione dell'attributo aiuta a bilanciare il carico. Come modello di prestazione si è utilizzata la *deviazione standard* rispetto al numero di risorse che ogni nodo della rete memorizza. Più la deviazione standard è bassa più le risorse sono distribuite uniformemente sulla rete.

Un altro test è stato poi svolto memorizzando nella stessa rete risorse composte da tutti e cinque gli attributi. I risultati sono mostrati in figura 6.27.

Per gli attributi i cui valori variano in un range relativamente basso ([0, 100]) e, come si note dalla loro distribuzione (figure 6.24 e 6.22), hanno la maggior parte delle occorrenze concentrate in pochi valori ($CpuUse$ e $Av.Mem$) l'utilizzo di un hashing basato sulla distribuzione non apporta particolari benefici.

Negli altri casi si ha una distribuzione migliore dei valori abbastanza evidente. In particolare si nota come nel caso di risorse con 5 attributi il bilanciamento (nel grafico *globale*) è migliore utilizzando una funzione di hashing derivata dalla distribuzione.

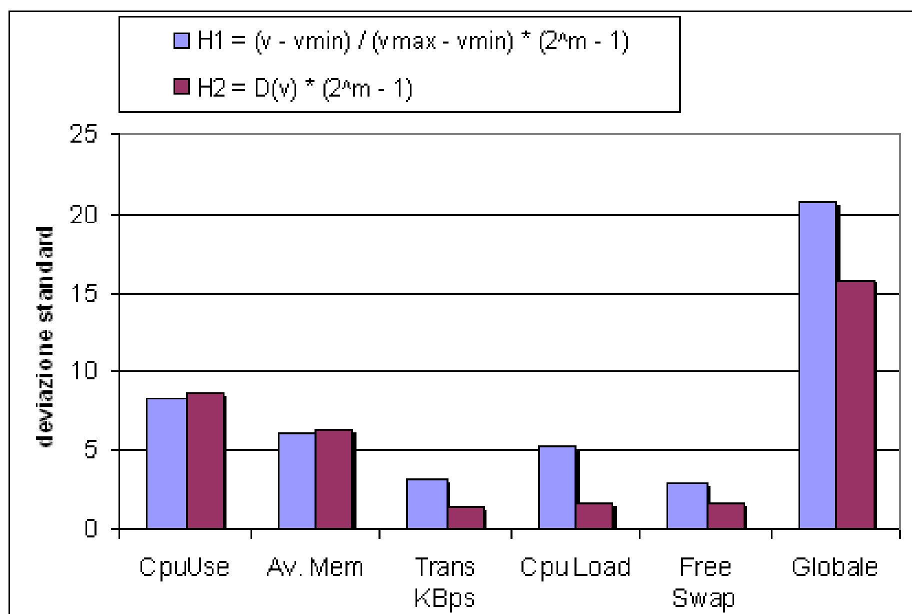

Figura 6.27: Deviazione standard dei vari attributi utilizzando funzioni di hash differenti.

Conclusioni

L'obiettivo di questa tesi è stato lo studio e l'implementazione di un sistema di risoluzione di range query multi attributo basato su DHT. Il supporto realizzato, Remedy, prende spunto da alcuni lavori presenti in letteratura e in particolare dall'approccio proposto in [3].

Il framework utilizzato per l'implementazione di Remedy è stato Overlay Weaver. I primi test effettuati hanno evidenziato come le prestazioni di Overlay Weaver come supporto per DHT siano paragonabili a quelle di un'altra implementazione di DHT come Bamboo [13].

Ulteriori prove sono state effettuate per verificare l'aderenza dell'implementazione di Remedy con i risultati teorici, in particolare per quanto riguarda il numero di nodi contattati durante la risoluzione di una query.

L'aspetto innovativo introdotto da Remedy è un insieme di tecniche introdotte per limitare il traffico generato dalla pubblicazione di risorse dinamiche, rendendo di conseguenza più scarica la rete in modo da diminuire il tempo di risposta nella fase di risoluzione delle query.

Sono state proposte due ottimizzazioni distinte per la riduzione del numero di messaggi in fase di pubblicazione. La prima riguarda la possibilità di evitare il routing quando la destinazione di una risorsa è la stessa per due misurazioni successive. La seconda introduce il concetto di popolarità di un attributo, cioè una stima, locale ad ogni nodo, della frequenza con cui un attributo viene scelto come dominante durante la risoluzione di una query. L'idea è quella di evitare del tutto l'aggiornamento per un attributo con popolarità bassa. Il modello analitico studiato per queste ottimizzazioni mostra un'effettiva diminuzione del numero di messaggi.

I test effettuati inoltre, dimostrano come diminuendo il traffico relativo

alla fase di pubblicazione, il tempo di risposta alle query diminuisce anche se in modo non lineare. Questo è dovuto al fatto che la risoluzione di una query ha comunque un costo minimo che rimane fisso, cioè non relativo al traffico sulla rete.

L'analisi dei dati di PlanetLab ha permesso di valutare la prima ottimizzazione con riferimento a dati riferiti ad uno scenario reale.

Alcuni aspetti rimangono tuttora aperti. In primo luogo la possibilità di dare una valutazione dell'impatto della seconda ottimizzazione, oltre ad altri aspetti riguardanti il bilanciamento del carico. Nel paragrafo successivo illustriamo alcuni di questi aspetti, che saranno oggetto di studio in un lavoro futuro.

Lavori futuri

Come abbiamo visto Remedy propone due serie di ottimizzazioni per limitare l'impatto degli aggiornamenti di risorse dinamiche. Una prima ottimizzazione consiste nel ricordare i percorsi di routing effettuati in precedenza, la seconda si basa sulla *popolarità* degli attributi. Per dare una valutazione di questa seconda ottimizzazione basata su dati strumentali è necessario possedere dati sulle distribuzioni delle query approssimate da funzioni di tipo *power-law* [29]. Utilizzando questo tipo di funzioni, in particolare le zipf [28], è possibile dare una valutazione di quanto la popolarità abbia un impatto sulle prestazioni, basandosi sulla distribuzioni nei valori delle query.

Un altro aspetto aperto è la situazione riguardante il bilanciamento del carico. Come si vede dagli esperimenti effettuati 6.3.4 si ha un cattivo bilanciamento nelle situazioni in cui la maggior parte delle occorrenze di un attributo è concentrato un range minimo di valori.

Una situazione di questo tipo è risolvibile facendo in modo che la parte finale dell'identificatore che rappresenta la risorsa sia calcolata in modo casuale. In questo modo lo stesso valore non è sempre memorizzato sullo stesso nodo, ma in un intorno vicino la cui dimensione è proporzionale alle

cifre dell'identificatore che si vogliono casuali. Un approccio di questo tipo è utilizzato da SWORD [10].

Bibliografia

[1] A. S. Cheema , M. Muhammad , I. Gupta, *Peer-to-Peer Discovery of Computational Resources for Grid Applications* Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing, p.179-185, November 13-14, 2005

[2] Cristina Schmidt , Manish Parashar, *Flexible Information Discovery in Decentralized Distributed Systems* Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03), p.226, June 22-24, 2003

[3] Min Cai, Martin Frank, Jinbo Chen, Pedro Szekely, *MAAN: A Multi-Attribute Addressable Network for Grid Information Services*, grid, p. 184, Fourth International Workshop on Grid Computing, 2003.

[4] Frank Dabek, Ben Zhao, Peter Druschel, John Kubiatowicz e Ion Stoica, *Towards a Common API for Structured Peer-to-Peer Overlays*, Proc. the 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03), February 2003.

[5] K. Shudo et al., *Overlay Weaver: An overlay construction toolkit*, Comput. Commun. (2007).

[6] Berkeley DB Java Edition. http://www.sleepycat.com/products/bd-bje.html.

[7] Grid5000 web site. <https://www.grid5000.fr/>.

[8] XML-RPC Home Page. <http://www.xmlrpc.com/>.

[9] Resource Management System for High Performance Computing. http://oar.imag.fr/

[10] D. Oppenheimer, J. Albrecht, D. Patterson, and A. Vahdat. *Scalable Wide-Area Resource Discovery.* Technical Report CSD04 -1334, University of California Berkeley, Berkeley, CA, USA, 2004

[11] I. Stoica, R. Morris, D. Karger, F. Kaashoek, e H. Balakrishnan. *Chord: A Scalable Peer to Peer Lookup Service for Internet Applications.*

[12] A. Rowstron e P. Druschel. *Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-Peer Systems*, 2001.

[13] S. Rhea, D. Geels, T. Roscoe, e J. Kubiatowicz. *Handling churn in a dht*, 2004.

[14] I. Foster, C. Kesselman e S. Tuecke. *The Anatomy of the Grid: Enabling scalable virtual organizations.*, 2001.

[15] Petar Maymounkov e David Mazières. *Kademlia: A Peer-to-peer Information System Based on the XOR Metric.*

[16] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. *A scalable content-addressable network.* In Proc. ACM SIGCOMM 2001, August 2001

[17] Heinz Johner, Larry Brown, Franz-Stefan Hinner, Wolfgang Reis, and Johan Westman. *Understanding LDAP.*

[18] XtreemOS : A Linux-based Operating System to support Virtual Organizations for next generation Grids. http://www.xtreemos.eu/

[19] PlanetLab, An open platform for developing, deploying, and accessing planetary-scale services. http://www.planet-lab.org/

[20] D. Oppenheimer, D. A. Patterson, and A. Vahdat, *A case for informed service placement on planetlab* PlanetLab Consortium, Tech. Rep. PDN04025, December 2004.

[21] Park, K. and V. S. Pai, *Comon: A Mostly-Scalable Monitoring System For Planetlab.* SIGOPS Operating Systems Review, Vol. 40, Num. 1, pp. 65-74, 2006.

[22] CoDeeN - A CDN on PlanetLab. http://codeen.cs.princeton.edu/

[23] Sean Rhea, Brighten Godfrey, Brad Karp, John Kubiatowicz, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, and Harlan Yu. *OpenDHT: A Public DHT Service and Its Uses.* Proceedings of ACM SIGCOMM 2005, August 2005.

[24] Praveen Kumar, G. Sridhar, V. Sridhar, *Bandwidth and Latency Model for DHT Based Peer-to-Peer Networks under Variable Churn,* icw, pp. 320-325, 2005 Systems Communications (ICW'05, ICHSN'05, ICMCS'05, SENET'05), 2005

[25] J. Li et.al, *Bandwidth-efficient management of DHT routing tables,* In USENIX 2nd Symposium on Networked Systems Design and Implementation (NSDI'05), May 2-4, 2005, Boston, MA.

[26] Huebsch, R., Hellerstein, J. M., Lanham, N., Loo, B. T., Shenker, S., and Stoica, I. 2003. *Querying the internet with PIER.* In Proceedings of the 29th international Conference on Very Large Data Bases - Volume 29 (Berlin, Germany, September 09 - 12, 2003)

[27] Rajiv Ranjan, Aaron Harwood, Rajkumar Buyya, *Peer-to-Peer Based Resource Discovery in Global Grids: A Tutorial,* IEEE Communications Surveys and Tutorials, IEEE Communications Society Press, USA, 2008.

[28] V. Ramasubramanian and E. G. Sirer. *eehive: O(1) lookup performance for power-law query distributions in peer-to-peer overlays.* In NSDI, 2004.

[29] M. E. J. Newman, *Power laws, Pareto distributions and Zipf 's law,* Contemporary Physics 46(5), 323-351 (2005).

[30] S. Saroiu, P. K. Gummadi, and S. D. Gribble. *A measurement study of peer-to-peer file sharing systems*. In Proc. MMCN, Jan. 2002

[31] M. Massie, B. Chun, and D. Culler. *The Ganglia distributed monitoring system: design, implementation, and experience*. Parallel Computing, Vol. 30, Issue 7, July 2004.

Elenco delle figure

2.1 Relazioni fra entries, attributi e valori in LDAP 12
2.2 Architettura ad alto livello di SWORD 19
2.3 Esempio di query in sword 21
2.4 Mapping tra il valore di un attributo e la chiave nello spazio della DHT. 23
2.5 Nodi gestori(M) e sensori (S) su InfoEye. 25
2.6 Traduzione attributi dinamici e statici in ID 28

3.1 Una rete Chord con identificatori di 6 bit. 31
3.2 Hashing in Remedy 34
3.3 Un esempio di risoluzione di query utilizzando l'attributo dominante. 39
3.4 MPA al variare della dimensione della rete, con K = 5. 41
3.5 MPA al variare del numero di attributi, con una rete di 100 nodi. 42
3.6 MPA al variare di x, in una rete di 100 nodi e con K=5. . . . 45
3.7 M_1 al variare di x, in una rete di 100 nodi e con K=5. 46
3.8 M_2 al variare di z, in una rete di 100 nodi e con K=5. 49

4.1 Architettura di Overlay Weaver 51
4.2 Routing iterativo e ricorsivo 53
4.3 Routing layer di Overlay Weaver 61
4.4 Esempio di file scenario 66
4.5 Interfaccia del Visualizer 67

5.1 Struttura della classe *Resource* 69
5.2 File XML contenente la descrizione degli attributi 70
5.3 Struttura della classe IDCache 71
5.4 Pseudo codice della pubblicazione sul nodo provider 73
5.5 Pseudocodice della pubblicazione sul nodo destinatario 75
5.6 La generica i-esima entrata della directory in cui sono memorizzate le risorse. 75
5.7 Struttura della classe Query 76
5.8 Pseudo codice della risoluzione di una query multi attributo sul nodo provider 77
5.9 Pseudo codice della risoluzione di una query multi attributo sul nodo destinatario 79

6.1 Primo test di scalabilità su Pianosa con GET ogni 11 secondi . 84
6.2 Primo test di scalabilità su Pianosa con GET ogni 27 secondi . 84
6.3 Primo test di scalabilità su Grid5000 con GET ogni 27 secondi 85
6.4 Primo test di scalabilità su Grid5000 con GET ogni 27 secondi 85
6.5 Secondo test di scalabilità su Pianosa con GET ogni 11 secondi 87
6.6 Secondo test di scalabilità su Pianosa con GET ogni 27 secondi 87
6.7 Secondo test di scalabilità su Grid5000 con GET ogni 11 secondi 88
6.8 Secondo test di scalabilità su Grid5000 con GET ogni 27 secondi 88
6.9 Test di affidabilità su Pianosa con Overlay Weaver 90
6.10 Test di affidabilità su Pianosa con Bamboo 90
6.11 Test di affidabilità su Grid5000 con il 16% di nodi uccisi. Si è considerata solo l'ultima serie di GET. 91
6.12 Test di affidabilità su Grid5000 con il 50% di nodi uccisi. Si è considerata solo l'ultima serie di GET. 91
6.13 Test di affidabilità su Grid5000 con il 66% di nodi uccisi. Si è considerata solo l'ultima serie di GET. 92
6.14 Numero di nodi contattati con selettività fissa all'aumentare della rete. 95
6.15 Numero di nodi contattati all'aumentare della selettività con rete da 256 nodi. 95

6.16 Latenza media su Pianosa, con rete da 30 nodi 96
6.17 Latenza media di una query su Grid5000 con reti da 50,100 e 300 nodi 97
6.18 Latenza al variare del numero di attributi con un rete da 30 nodi su Pianosa 98
6.19 Latenza al variare del numero di attributi con reti da 50, 100 e 300 nodi su Grid5000 99
6.20 Alcuni dei dati rilevati da CoMon 102
6.21 Frequenza di ripetizione degli attributi 104
6.22 Distribuzione del tempo di utilizzo della CPU. 105
6.23 Distribuzione del numero di processi attivi sulla CPU. 106
6.24 Distribuzione dell'output del programma MemPress. 106
6.25 Distribuzione dei dati in uscita e in entrata con una risoluzione di 16 KBps 107
6.26 Rappresentazione grafica del churn 108
6.27 Deviazione standard dei vari attributi utilizzando funzioni di hash differenti. 111

B.1 Infrastruttura di rete di Grid5000 127

Elenco delle tabelle

4.1 Lista dei packages in Overlay Weaver e loro descrizione. 54
4.2 Principali parametri dell'interfaccia DHTConfiguration. 56
4.3 Principali parametri in RoutingServiceConfiguration. 58
4.4 Principali parametri in ChordConfiguration. 58
4.5 Corrispondenze fra i metodi dell'interfaccia RoutingAlgorithm e vari algoritmi di routing. 64

Appendice A

Emulatore

A.1 Comandi

In questo capitolo è presentata una breve descrizione della sintassi e della semantica dei principali comandi per il controllo dell'emulatore in locale.

- **class** <nome classe>

  La successiva chiamata del comando *invoke* istanzia la classe ed invoca il suo metodo *main*.

- **arguments** [<argomento> ...]

  Al momento dell'invocazione passa questi argomenti al *main* della classe definita con *class*.

- **invoke** [<ID of app instance>]

  Istanzia la classe precedentemente definita ed invoca il suo metodo *main*. Se la classe implementa l'interfaccia *EmulatorControllable* viene invocato invece il metodo *start*. Se non è presente il parametro ID, viene invocata l'ultima classe definita.

- **priority** <priorità>

  I thread creati dal comando *invoke* successivo, avranno questa priorità. Il valore specificato è utilizzato come priorità del processo dell'Emulatore.

- **control** <ID of app instance> <commando sullo stdin>

  Scrive il comando specificato sullo *standard input* di una particolare istanza attraverso l'interfaccia *EmulatorControllable*. Gli ID delle istanze partono da zero e seguono l'ordine di invocazione. Ad esempio la prima istanza che esegue l'host *emu0* ha ID 0. L'ID speciale 'all' indica tutte le istanze.

- **timeoffset** <offset>

  Il successivo comando *schedule* aggiunge il valore dell'offset specificato alla sua schedulazione.

- **schedule** <time>[,<intervallo>[,<cicli>]] <comando> [...]

  Schedula il comando specificato come secondo argomento (invoke and control) al tempo specificato (in millisecondi). Se è specificato un intervallo si aspetta il suo valore fra una esecuzione del comando specificato e la successiva. Se viene specificato un numero di cicli il comando è eseguito quel numero di volte, altrimenti è ripetuto all'infinito.

- **include** <nome del file>

  Parsa il contenuto del file specificato come una lista di comandi.

- **quit**

  Termina l'esecuzione dell'emulatore.

I comandi seguenti vengono utilizzati negli scenari in cui si prevede un utilizzo distribuito dell'emulatore.

- **remote connect**
  Invoca i worker sui computer remoti.

- **remote directory**[<path sulla quale viene eseguita la JVM>]
  Per invocare il worker, la shell va al path specificato.

- **remote javapath** [<path del comando java sul computer remoto>]
  Per invocare il worker si cerca il comando *javar* al path specificato.

- **remote jvmoption** [<lista di opzioni>]
  Le opzioni specificate sono passate al comando *java* per l'invocazione dei worker.

Appendice B

Grid5000

B.0.1 Accesso a GRID5000

GRID5000 è composta da siti di cluster locati sul territorio francese. In figura B.1 è mostrata l'infrastruttura di rete. Ogni sito ha uno o più cluster con architetture diverse (AMD o Intel) ma generalmente a 64 bit con più (2-4) core.

Ogni sito ha una macchina denominata *frontend* per permettere l'accesso ai suoi cluster.

Il punto di accesso a grid5000 è unico: *access.rennes.grid5000.fr*. Da qui si può accedere ai frontend di tutti i cluster tramite *ssh*.

Esempio 6. *ssh frontend.lyon.grid5000.fr è utilizzato per accedere ai clusters di Lione.*

Ogni sito mantiene il proprio filesystem, quindi è necessario sincronizzare manualmente gli stessi, se si prevede di utilizzare gli stessi file su più clusters.

B.0.2 Utilizzo di grid5000

La gestione dei *job* in grid5000 è controllata tramite il resource manager OAR [9]. OAR mette a disposizione una serie di comandi (interfaccia testuale) con i quali è possibile utilizzare la piattaforma.

I principali metodi di controllo dei job sono due:

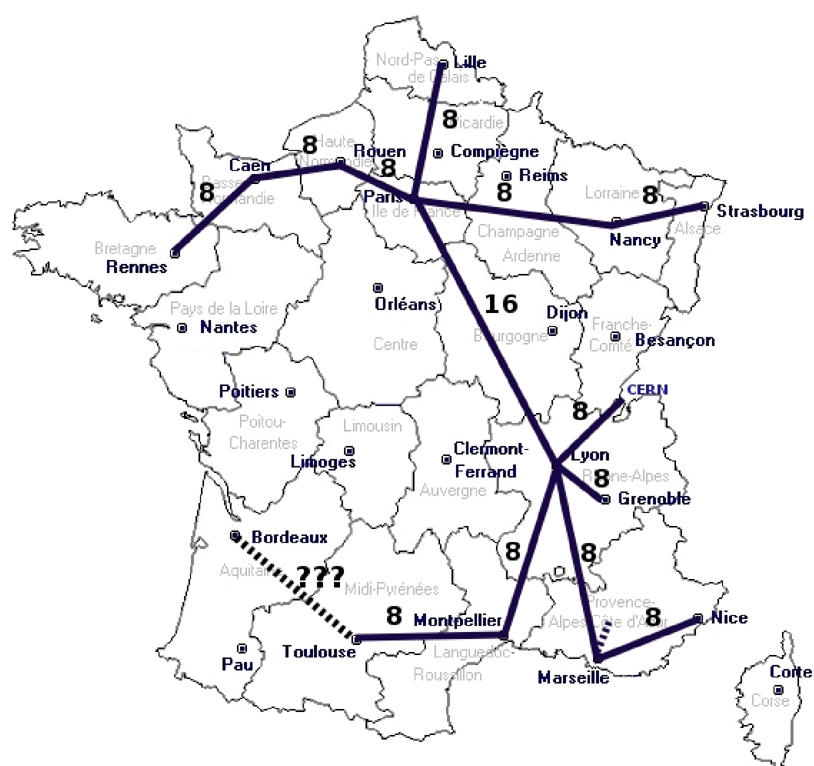

Figura B.1: Infrastruttura di rete di Grid5000

- Interattivo: controllo diretto dei nodi da parte dell'utente tramite shell.
- Batch: l'utente stabilisce a priori quale script/programma deve essere eseguito

Per OAR un job è composto dal programma con le sue risorse e dall'ambiente di esecuzione. La prenotazione di risorse coincide con la creazione di un ambiente.

B.0.3 Panoramica dei comandi

oarsub

Permette di prenotare i nodi e mandare in esecuzione script, è il comando principale.

Con la prenotazione dei nodi si crea anche un ambiente. OAR fornisce il riferimento al primo nodo di quelli prenotati. Per conoscere il suo ambiente questo nodo ha a disposizione alcune variabili d'ambiente. Le più significative:

- $OAR_NODE_FILE: è il nome di un file che contiene la lista di tutti i nodi riservati. É importante notare che se un nodo ha più di un core viene ripetuto più volte nel file.
- $OAR_JOBID: l'id del job corrente.
- $OAR_NB_NODES: il numero di nodi riservati.

Il comando oarsub è invocato con alcune opzioni:

- -I : attiva la modalità interattiva.
- -l<arg> : descrive le risorse che si vogliono prenotare. Esempio successivamente.
- -r "2008-05-11 13:32:03" : la prenotazione comincia dalla data in argomento.

- -C : carica l'ambiente di un job precedentemente riservato.
- -n : assegna un nome al job.

Nota: l'opzione -r e la modalità interattiva (-I) sono incompatibili.

Esempio 7. *oarsub -r "2008-01-17 12:00:00" -l/nodes=45,walltime="02:00:00"*

prenota 45 nodi, per due ore (walltime) il 17 gennaio alle 12:00. L'output del comando fornisce l'ID del job.

Esempio 8. *oarsub -C jobID*

Si usa per richiamare l'ambiente (shell) la mattina del 17 gennaio.

Esempio 9. *oarsub -I -l/nodes=10/cpu=2*

Fornisce immediatamente, previa disponibilità dei nodi, una shell al primo nodo (modalità interattiva) da cui è possibile accedere a 10 nodi con almeno 2 cpu.

Esempio 10. *oarsub -l/nodes=4 test.sh*

Modalità batch. Esegue sul primo dei 4 nodi lo script *test.sh*

oarsh

Sostituisce il comando ssh. Utilizzato nella modalità interattiva e negli script per accedere ai nodi riservati.

Esempio 11. *oarsh pastel-9.toulouse.grid5000.fr <comando>*

Esegue <comando> sul nodo specificato.

oardel

Il comando utilizzato per per cancellare i job.

Esempio 12. *oardel jobID*

Cancella il job con ID jobID.