Università degli Studi di Padova

Facoltà di Ingegneria

Corso di Laurea Triennale in Ingeneria Informatica

tesi di laurea

PariDiESeL

Hash Table Implementation

Relatore: prof. Enoch Peserico Stecchini Negri De Salvi

Correlatore: dott. Vincenzo-Maria Cappelleri

Laureando: Matteo Gervasi

22 novembre 2013

alla mia famiglia, alla mia capra,
e a chi aspettava questo momento da tanto tempo.

Indice

Capitolo 1

Introduzione

L'oggetto di questa tesi riguarda il lavoro svolto all'interno del progetto PariPari. Nello specifico verrà analizzato il contributo fornito alla progettazione della libreria DiESeL, tramite l'implementazione di una tabella hash "dedicata".

La parte iniziale del lavoro si è focalizzata sullo studio del funzionamento delle tabelle hash, esaminando nel dettaglio le caratteristiche di tale struttura dati e le tecniche possibili da utilizzare per una sua implementazione. Successivamente si è passati a una valutazione delle scelte migliori di implementazione tra quelle possibili, e infine, si è proceduto all'implementazione vera e propria in linguaggio Java.

Prima di ripercorrere nel dettaglio le fasi del lavoro, nel capitolo 2 viene fornita una breve introduzione riguardo il progetto PariPari e le caratteristiche fondamentali della sua architettura. Il capitolo viene poi concluso con una descrizione della struttura e del funzionamento della libreria DiESeL.

Il terzo capitolo contiene una descrizione dell'approfondimento teorico che è stato svolto per lo sviluppo del codice della tabelle hash. Nel corso del capitolo vengono analizzate la struttura dati e le tecniche di implementazione possibili.

Nel capitolo 4 si fornisce una spiegazione della motivazione che ha portato il team di DiESel alla decisione di fornire un'implementazione dedi-

cata di una tabella hash. Inoltre vengono illustrate le scelte fatte prima dell'effettiva realizzazione.

Nel capitolo 5 viene analizzato il codice elaborato in relazione a tali scelte.

Infine il capitolo 6 presenta le considerazioni finali sul lavoro svolto ed eventuali possibili sviluppi futuri.

Capitolo 2

Il progetto PariPari

Introduzione

Il progetto *PariPari* prevede la realizzazione di un' applicazione peer-to-peer multifunzionale e multipiattaforma. Il linguaggio scelto per la sua realizzazione è Java, la cui caratteristica principale è l'indipendenza della piattaforma di esecuzione, che si traduce in un'elevata portabilità. Java infatti è un linguaggio semi-interpretato; le istruzioni cioè non vengono lette da un compilatore ma gestite dalla JVM (Java Virtual Machine) prima di essere eseguite. E' proprio questa caratteristica che garantisce la portabilità del codice, il quale non necessita la compilazione su ogni macchina prima di essere eseguito. In questo modo *PariPari* può essere lanciato direttamente dal browser dell'utente tramite Java Web Start; questa tecnologia permette di scaricare ed eseguire le applicazioni direttamente dal web, nascondendo all'occhio dell'utente le procedure di installazione ed aggiornamento. Quando l'applicazione viene avviata per la prima volta, viene memorizzata localmente dal software Java Web Start. In questo modo ogni ulteriore avvio, anche se attraverso il browser, avviene in modo pressoché immediato, dal momento che tutte le risorse necessarie sono disponibili localmente. Ad ogni avvio dell'applicazione inoltre, *PariPari* verifica se è disponibile una nuova versione e, in tal caso, la scarica e l'avvia automaticamente; quest'ultima funzione, per essere più precisi, è fornita dal

componente chiave del software *PariPari*, il Core, le cui mansioni saranno descritte nei prossimi paragrafi.

Un altro aspetto cardine del linguaggio di programmazione usato è la sicurezza; a differenza di altri linguaggi Java integra un certo numero di strategie dedicate. Ad esempio le dimensioni dei dati di input sono tenute costantemente sotto controllo per impedire attacchi tramite buffer overflow; questo tipo di vulnerabilità consentirebbe infatti di ottenere accesso al sistema inviando più dati rispetto alla capienza del buffer dell'applicazione che, a quel punto, li memorizzerebbe andando a sovrascrivere una parte del programma stesso.

L'impiego del garbage collector poi, garantisce un certo livello di "pulizia": i dati non più utilizzati vengano eliminati dalla memoria. In tal modo si impedisce a un eventuale codice malevolo di sovraccaricarla ed impossessarsi del controllo dell'applicazione.

Si ricorda infine che Java è un linguaggio di programmazione orientato agli oggetti e, in quanto tale, permette di utilizzare l' incapsulamento; tale principio consiste nella proprietà dell'oggetto di incorporare al suo interno metodi e attributi, agendo quindi da contenitore sia di strutture dati sia di procedure che li utilizzano. In questo modo l'oggetto è visto come una sorta di scatola nera (o black-box) che permette il mascheramento delle informazioni desiderate; questo risultato è ottenuto differenziando, all'interno del codice, una sezione pubblica (informazioni visibili all'esterno) da una privata (informazioni non accessibili ad altri oggetti e quindi invisibili all'esterno).

2.1 Architettura di rete

La rete di *PariPari* è stata progettata tenendo in considerazione soprattutto gli svantaggi che presenterebbe l'utilizzo di un'architettura centralizzata. Ricorrendo ad una architettura centralizzata, infatti, i server rappresenterebbero dei punti critici e nel caso di indisponibilità di questi, la rete collasserebbe. Una rete centralizzata inoltre è facilmente afflitta da problemi di scalabilità; all'aumentare del numero di utenti si rende necessario

aggiungere dei nuovi server per evitare un sovraccarico. Infine la presenza di server renderebbe la rete più vulnerabile ad attacchi di tipo DOS (Denial Of Service), che mirano a rendere inutilizzabile un servizio tramite un elevato numero di richieste inviate alla macchina che lo esegue, fino alla saturazione delle risorse disponibili.

Per ovviare a questi problemi si è scelto di utilizzare per *PariPari* una rete priva di server. Una delle caratteristiche basilari di una rete P2P è infatti che tutti i nodi sono considerati alla pari, senza distinzioni di tipo gerarchico. La struttura generale si basa su una variante dell'algoritmo Kademlia. Di seguito si fornisce una breve spiegazione del funzionamento dell'algoritmo in questione, per poi illustrare le modifiche che sono state apportate in *PariPari*.

Kademlia è un protocollo di rete P2P che indicizza nodi e risorse attraverso la realizzazione di una tabella di hash distribuita e permette di ricercare entrambi in maniera totalmente decentralizzata. Ogni risorsa e ogni nodo ricevono dall'algoritmo un ID calcolato tramite una funzione di hash. Nel caso delle risorse l'ID e' calcolato con una funzione che utilizza come input una stringa caratteristica della risorsa (ad esempio l'autore per un file musicale), che ci si aspetta che venga utilizzata come chiave per le ricerche per quella risorsa. Nel caso dei nodi l'ID e' calcolato con una funzione di hash che utilizza informazioni caratteristiche del nodo con l'obiettivo di minimizzare la probabilita' che due nodi ricevano lo stesso ID. Nodi e risorse vengono supposti appartenenti ad un albero binario, le cui posizioni sono determinate dal loro ID; si assegna infatti al ramo sinistro uscente da ciascun elemento dell'albero l'identificativo 1, e al ramo destro lo 0. L'ID di un nodo/risorsa coincide con la sequenza ordinata di bit che si trovano lungo il cammino dalla radice al nodo/risorsa in esame. Il protocollo si assicura anche che ciascun nodo conosca almeno un nodo in ciascun sottoalbero della rete diverso da quello che lo contiene.

Il concetto fondamentale su cui si basa l'algoritmo è quello della distanza, che viene calcolata utilizzando la metrica XOR, ovvero eseguendo l'operazione di OR esclusivo tra due ID.

Al momento della richiesta di una risorsa da parte di un host, grazie

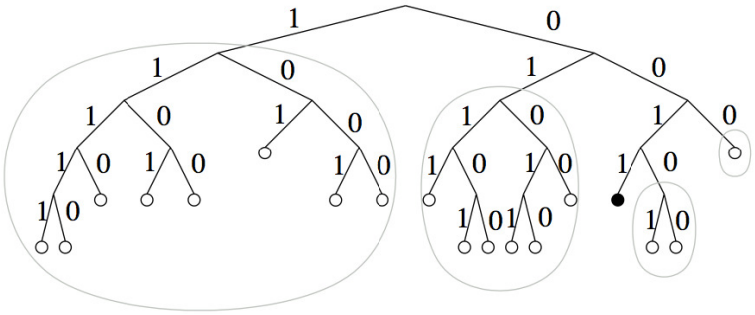

Figura 2.1: Albero binario di una rete Kademlia. In nero è evidenziata la posizione di un nodo con prefisso unico 0011. Cerchiati in grigio sono i sottolaberi in cui in cui ci deve essere almeno un nodo conosciuto dal nodo 0011

al fatto che nodi e risorse vengono identificati con lo stesso formato, la distanza tra i due può essere determinata confrontando i loro ID. Se un nodo vuole condividere una risorsa ne calcola l'ID (tramite la funzione di hash), ricerca gli host con ID più prossimo (in metrica XOR), e infine richiede a tali host di memorizzare le informazioni necessarie per raggiungerla. Quando invece un host vuole ricercare una risorsa, invia una richiesta a 3 nodi, tra quelli da esso conosciuti con ID più vicino all'ID della risorsa cercata. Da ciascuno dei 3 nodi può poi ricevere al massimo 20 riferimenti ai loro nodi conosciuti con ID più prossimo a tale risorsa. I riferimenti che il richiedente può ricevere sono quindi al più 60 e il passo successivo consiste nello scegliere tra questi i 3 più vicini all'ID cercato. Il richiedente a questo punto inoltra la richiesta ai 3 nodi scelti e ripete i passi svolti al punto precedente. Il procedimento continua finchè viene raggiunto il riferimento al nodo che dispone della risorsa cercata.

La concezione ad albero di Kademlia implica che le richieste al nodo con ID più prossimo vengano effettuate su sottoalberi via via più piccoli. Ad ogni salto la dimensione del sottoalbero di ricerca si dimezza e in questo modo si ha la certezza che l'algoritmo converga dopo un numero di passi finiti. La ricerca può essere vista come un cammino all'interno di un albero e quindi la sua complessità è dell'ordine del numero di salti necessari per passare da una foglia all'altra, ovvero proporzionale all'altezza dell'albero

che è $O(\log n)$.

In *PariPari*, per ridurre il numero delle comunicazioni tra nodi per la condivisione/ricerca di una risorsa, è stata introdotta una modifica all'algoritmo originale di Maymounkov e Mazières. Il primo nodo sceglie, tra i nodi che deve contattare, un nodo preferito. Questo successivamente inoltra la richiesta di ricerca ad uno dei nodi che è in procinto di trasmettere come risposta al primo nodo. Il comportamento appena descritto avviene ad ogni iterazione: il nodo preferito sceglie a sua volta un preferito tra i suoi nodi candidati (che come visto prima sono quelli da esso conosciuti con ID più vicino all'ID della risorsa desiderata), mentre trasmette la propria risposta all'origine. Se per un problema di connessione la comunicazione fallisce la ricerca continua comunque in parallelo seguendo il protocollo standard.

Con il metodo standard, per permettere all'origine di conoscere l'indirizzo del nodo cercato sono necessarie un numero di comunicazioni pari a:

$$comunicazioni = (distanza - 1) * 2$$

Con il nuovo sistema, potrebbe invece bastare

$$comunicazioni = distanza - 1$$

Se assumiamo che tutte le comunicazioni hanno la medesima durata, con questa modifica si può ottenere un risparmio del 50% in termini di tempo.

2.2 Struttura del client PariPari

Per la stesura del codice di *PariPari* si è scelta un'architettua a plugin, applicativi non autonomi che interagiscono con un programma per ampliarne le funzioni. Nella struttura del software si possono differenziare due diversi tipi di Plugin: quelli della cosiddetta cerchia interna (Core, Crediti, Connettività, Storage e DHT) e quelli appartenenti alla cerchia esterna (IRC, DNS, Voip, Emule, Torrent, etc...). I primi si occupano di

gestire la coordinazione tra i vari plugin e di fornire i servizi basilari per il loro funzionamento. I secondi offrono invece servizi specifici per l'utente sfruttando le funzioni offerte dai plugin della cerchia interna. Quando un plugin vuole offrire un servizio ad altri plugin, deve esporre almeno una Interfaccia di Programmazione di Applicazione API (Application Programming Interface). I plugin però non comunicano direttamente tra loro; le comunicazioni devono essere filtrate dal Core, che si occupa anche di impedire richieste da plugin maligni e di evitare che questi possano assumere il controllo di risorse destinate ad altri. Quando un plugin intende richiedere una risorsa ad un altro deve inviare una richiesta al Core, specificando la risorsa desiderata e il plugin da cui vuole ottenerla. Il Core esamina la richiesta e, se valutata legittima, la passa al plugin destinatario, che risponderà fornendo la risorsa richiesta. Questo approccio è la chiave per consentire ad ogni sviluppatore di utilizzare i plugin come delle black-box, ovvero senza conoscerne i dettagli realizzativi.

2.3 Il sistema dei crediti

In una rete peer-to-peer la gestione dei crediti è uno dei punti cruciali poiché regola i rapporti di collaborazione tra i nodi. In molte reti l'assegnazione dei crediti è strutturata in modo tale da indurre gli utenti a rimanere connessi alla rete il più a lungo possibile e a condividere il maggior numero di risorse possibili.

Il sistema dei crediti in *PariPari* si ispira invece al libero mercato. Di seguito si fornisce una breve spiegazione del concetto utilizzando un piccolo esempio. Assumiamo che, connessi alla rete, ci siano tre utenti, che chiameremo Alice, Bob e Charlie. Alice conclude usualmente affari con Bob, così come con Charlie; quest'ultimo però non ha mai avuto nessun contatto diretto con Bob. Si immagini ora che Charlie sia interessato ad acquisire una risorsa di Bob. Purtroppo, Charlie non vanta un credito con Bob e, secondo il sistema dei crediti,non ha modo per ottenere tale risorsa. Si supponga però che Alice sia fortemente indebitata con Charlie. Charlie procederà quindi ad acquisire la risorsa grazie alla mediazione di Alice, che

risulterà essere in debito con Charlie e in credito con Bob. In questo modo Alice si troverà ad avere crediti verso diversi altri utenti che le verranno richiesti con frequenze e quantità sempre diverse. Il suo scopo sarà quello di riuscire a compiere queste mediazioni, guadagnando comunque sulla transazione, non rimanendo mai sprovvista di quei crediti che potrebbero servirle in futuro. I crediti accumulati dall'utente possono essere poi gestiti distribuendoli tra i vari plugin a seconda delle sue necessità; così macchine particolarmente "ricche" di una specifica risorsa (come ad esempio la memoria disponibile), possono guadagnare i crediti necessari per usufruire di tutti gli altri servizi. Al momento dell'ingresso di un nuovo nodo nella rete, viene stabilito che questo non possa indebitarsi, senza prima aver investito una certa quantità delle sue risorse. Nell'ipotesi che il nuovo utente voglia acquisire una risorsa della rete, questo non avrebbe ancora nulla con cui scambiarla e neppure altri crediti per procedere a una mediazione. Potrebbe allora sprecare ad esempio la sua banda, come garanzia della sua buona volontà per acquisire la risorsa desiderata.

Nella progettazione dei singoli Plugin bisogna dunque tener conto della limitata disponibilità di risorse, e cercare di agire col massimo della parsimonia. Nel caso in cui un Plugin non abbia sufficienti crediti per eseguire una richiesta, questa viene direttamente respinta dal core, fino all'eventuale sospensione del Plugin stesso. Questo accorgimento impedisce ad eventuali Plugin maligni di inondare il core con continue richieste non soddisfabili al fine di interrompere il funzionamento degli altri Plugin.

2.4 DiESeL

DiESeL è l'acronimo di Distributed Extensive Server Layer ed è una libreria che permette la gestione di applicazioni distribuite di tipo server sulla rete di *PariPari*. Più precisamente la libreria crea un substrato al di sotto del livello applicativo del server garantendone la distribuzione su più nodi della rete.

Il modulo *DiESeL* è composto da una classe principale che gestisce al suo interno una serie di thread. La classe principale del modulo è stata

Figura 2.2: Logo della libreria *DiESeL*

denominata Distributore e in essa vengono gestiti i thread e le comunicazioni tra di essi. In particolare, tramite la classe *ServerLayer* vengono gestite le operazioni per la ricerca e l'attivazione di un nodo. Quando un nuovo nodo entra a far parte della rete, può essere avviato in modalità passiva o in modalità attiva. Nel primo caso rimane in attesa di essere contattato da un server; nel secondo caso invece, avvia il server distribuito che successivamente verifica ad intervalli regolari la presenza di altri nodi. Se il nodo attivo si disconnette, il nodo che ne rileva la disconnessione si autoproclama attivo, ereditando i compiti del suo predecessore.

Si fornisce una breve spiegazione delle principali classi del modulo *DiESeL*, la cui struttura è rappresentata nella figura 2.3.

FatherListener è il thread responsabile delle comunicazioni con il *ServerLayer*. Si limita a inoltrare i messaggi che vengono depositati dal livello superiore su una coda di Object a tutti i nodi del server.

CKASender e *CKARecieve*r sono i thread che realizzano l'algoritmo CKA (Cooperative Keep Alive); questo sistema riconosce le disconnessioni dei nodi e le comunica agli altri, prendendo le misure necessarie affinché non si verifichino intoppi nelle comunicazioni attraverso la rete. Più precisamente ogni nodo possiede una lista che contiene i nodi ai quali è

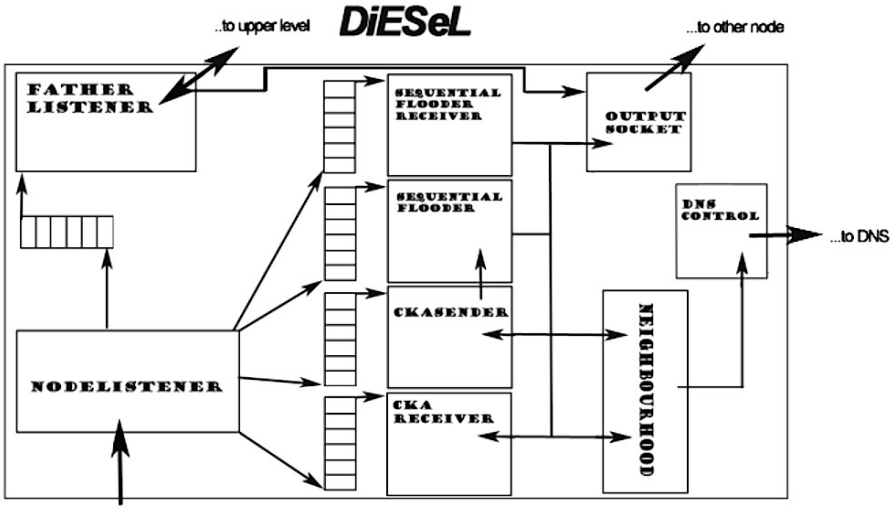

Figura 2.3: Schema di interazione delle classi del modulo *DiESeL*

connesso e ogni X secondi un thread contatta il nodo in testa a tale lista per verificare il mantenimento di tale connessione.

SequentialFlooder e *SequentialFlooderRecieve*r sono i due thread che eseguono l'algoritmo di Sequential Flooding per l'outage detection; ovvero si preoccupano di avvisare gli altri nodi della rete che un nodo è andato perso, aggiornando in maniera sequenziale le liste dei vicini a ciascun nodo.

NodeListener è il thread che una volta ricevuto il PingMessage lo ricompone dal flusso di byte e lo identifica; una volta individuato il tipo di messaggio il thread lo deposita nella rispettiva coda in modo che gli altri thread possano leggerlo.

OutputSocket è il thread che si occupa di prendere i messaggi destinati ad un altro nodo, convertirli in un flusso di byte e inviarli.

Neighborhood è la tabella in cui vengono mantenute le informazioni della rete: ogni elemento di tale tabella associa ad un nodo i suoi "vicini". La struttura dati utilizzata è la *DiESeLHashTable*, oggetto di questa tesi, che verrà analizzata nello specifico nel capitolo 5.

DNSController è un thread che controlla periodicamente lo stato dei nodi sulla rete e genera una lista ordinata di indirizzi per comunicarla al modulo DNS di *PariPari*.

DiESeL, durante il suo funzionamento, interagisce costantemente con il plugin DHT (Distributed Hash Table) di *PariPari*. E' infatti quest'ultimo che fornisce alla libreria i nodi pubblicati sulla tabella distribuita che sono disponibili alla connessione ad un server.

Quando un nodo A viene avviato come attivo, riceve da DHT i candidati alla connessione; ne sceglie allora uno (nodo B) e prova a contattarlo con un messaggio; in esso vengono inseriti i riferimenti temporali del server (fondamentali per la sincronizzazione dei successivi ping), l'ID del nodo attivo e la lista dei nodi a cui il nodo B deve connettersi. A questo punto il nodo B sceglie, tra i nodi indicati dal nodo comandante (nodo A), uno a cui connettersi (ad esempio il nodo C); da questo riceve la sua lista dei vicini. Allora il nodo B aggiorna la sua tabella dei vicini e invia in broadcast a tutta la rete le parti modificate. Nella fase successiva il nodo B invia al nodo C un ping, al quale il nodo C risponde con un altro ping in cui specifica l'instante in cui il nodo B dovrà pingarlo nuovamente. Questa serie di ping e risposte continua quindi fino a quando uno dei due nodi non si disconnette ed è il meccanismo che consente di rilevare un'eventuale disconnessione.

Capitolo 3

Tabelle Hash

Introduzione

Le tabelle hash sono strutture dati utilizzate per l'implementazione di dizionari. La loro efficienza consiste nel fatto che, anche se nel caso peggiore la ricerca di un elemento della tabella di dimensione N può richiedere un tempo $O(N)$, il tempo medio di ricerca, è $O(1)$. Una tabella hash, consiste di due elementi specifici: un array di bucket e una funzione hash. Il primo è un array A di dimensione N, in cui ogni cella contiene una collezione di coppie chiave-valore. Un elemento con chiave k verrà inserito nel bucket $A[k]$. Se le chiavi sono numeri interi appartenenti all'intervallo $[0, N-1]$, allora ogni bucket memorizza al suo interno al più un elemento. In questo modo le operazioni di inserimento, ricerca e rimozione richiedono tempo costante. Se da un lato questo sembra essere un ottimo risultato, dall'altro è opportuno tenere conto di due svantaggi. Innanzitutto lo spazio occupato dalla tabella è proporzionale a N; inoltre, se N è molto più grande del numero di elementi n, si rischia di incorrere in uno spreco di memoria. Un altro svantaggio è poi dato dal fatto che le sole chiavi utilizzabili in questo scenario, sono quelle che hanno valori compresi nell'intervallo $[0, N-1]$. Per questi motivi, abbinata al bucket di array, si utilizza una funzione che possa "tradurre" in maniera efficace le chiavi in numeri interi compresi nell'intervallo.

3.1 Funzioni hash

La funzione hash, come detto, ha il compito di associare a ogni chiave k un valore intero appartenente all'intervallo $[0, N-1]$. L'idea è infatti di usare il valore della funzione hash come indice dell'array di bucket. Se si utilizzasse infatti come indice la chiave stessa, si dovrebbe utilizzare un valore N molto grande o limitare l'intervallo di chiavi utilizzabili, approcci che, come visto nel paragrafo precedente, si rivelerebbero poco efficienti.

Possiamo suddividere l'operazione di calcolo del valore della funzione in due parti: la prima consiste nel tradurre la chiave in un valore intero che la identifichi, chiamato codice hash; nella seconda si trasforma il suddetto codice in un altro valore intero contenuto nell'intervallo $[0, N-1]$, in modo da poter assegnare ad un elemento l'indice corretto appartenente al bucket array. Quest'ultima operazione è calcolata dalla funzione di compressione.

I codici hash e le funzioni di compressione saranno analizzati nel dettaglio rispettivamente nel terzo e quarto paragrafo di questo capitolo.

3.1.1 Requisiti di una buona funzione hash

Una buona funzione di hash cerca di soddisfare al meglio l'ipotesi di uniformità semplice; in questo modello, formalizzato da W.W. Peterson nel 1957, si assume che ogni chiave venga inserita in una posizione casuale della tabella, così che le possibili combinazioni delle N celle occupate $\binom{N}{n}$ si presentino con la stessa probabilità delle N-n celle vuote. In questo modo il riempimento di ogni posizione della tabella è indipendente da tutte le altre. Si assume cioè che ogni chiave sia estratta in modo indipendente dall'universo delle chiavi U secondo la distribuzione di probabilità P, tale che, sia *P(k)* la probabilità che la chiave k venga estratta, si abbia:

$$\sum_{k:h(k)=j} P(k) = \frac{1}{n} \qquad per\ j = 0, 1, ..., n-1$$

.

Sfortunatamente però, in generale non è possibile agire su questa condizione perché P non è nota.

Quello che si può fare è usare delle tecniche euristiche per creare una funzione hash che si comporti bene con una buona probabilità. Si consideri ad esempio la tabella dei simboli di un compilatore, in cui le chiavi siano stringhe qualsiasi di caratteri che rappresentano gli identificatori presenti in un programma. E' frequente trovare identificatori molto simili, come *pt* e *pts*, in uno stesso programma. Una buona funzione hash dovrebbe minimizzare la probabilità che i due identificatori corrispondano alla stessa locazione della tabella. Un approccio comune è quello di derivare il codice hash in modo che sia indipendente da qualunque configurazione possa esistere nei dati. Nell'espressione della divisione a modulo che si vedrà in seguito ad esempio, si utilizza un numero primo; a meno che questo non sia correlato con qualche particolarità nella distribuzione di probabilità, questo metodo da in genere buoni risultati.

3.2 Codici hash in Java

Data una funzione hash $A[k]$ e due chiavi distinte *k1* e *k2*, quando $h(k1) = h(k2)$ si dice che si è verificata una collisione. Di conseguenza non è possibile inserire semplicemente un nuovo elemento *(k,v)* direttamente alla posizione $A[k]$ del bucket di array. Anche se si sviluppa un metodo efficiente per la gestione delle collisioni, come verrà illustrato nel paragrafo 4, è importante assicurarsi che queste non avvengano già in prima fase, ovvero in occasione della generazione del codice hash; senza questo accorgimento infatti, nessuna funzione di compressione, per quanto sia efficace, riuscirà ad evitare il verificarsi di collisioni. Questo perché la funzione, per come è definita, rialloca semplicemente i codici hash all'interno dell'intervallo desiderato, mantenendo collidenti due chiavi che lo erano già prima del suo intervento. Inoltre, anche se apparentemente ovvio, è fondamentale che il codice hash sia lo stesso per due chiavi distinte con lo stesso valore , in quanto appartenenti alla stessa classe di equivalenza.

3.2.1 Il metodo hashCode()

In Java il metodo *hashCode()* appartiene alla classe generica Object e restituisce un intero a 32 bit. Questo metodo viene ereditato da ogni oggetto e può essere quindi invocato sull'oggetto stesso per generare il codice hash corrispondente. Questa soluzione, anche se apparentemente vantaggiosa perché immediata, nasconde al suo interno un'insidia che può portare a risultati indesiderati. Per eseguire il calcolo, il metodo utilizza l'*ObjectID*; tale parametro è legato alla specifica istanza dell'oggetto e quindi alla sua locazione in memoria. Se utilizzato ad esempio per delle stringhe di caratteri, il metodo non fornisce i risultati sperati perché due stringhe uguali che occupano due locazioni diverse in memoria darebbero luogo a due codici hash diversi. E' quindi necessario estendere il metodo predefinito, sovrascrivendone il comportamento, in modo che soddisfi le nostre necessità. (Ad esempio la classe *String* in Java fa lo stesso).

Esistono diversi tipi di dati e diversi metodi per generare dei codici hash a partire da oggetti di questo tipo, presenteremo ora i principali.

3.2.2 Casting in valore intero

Per cominciare si può osservare, banalmente, che ogni tipo di dato in informatica è rappresentato in codice binario; si ipotizzi che il numero di bit necessari per codificare un elemento non superi il numero di bit utilizzati per codificare in binario il codice hash (che ad esempio può essere un valore intero a 32 bit); il codice dell'elemento in questione può essere allora calcolato come una semplice interpretazione come valore intero della sua codifica binaria.

Per dati java di tipo *byte*, *short* e *char* possiamo fare quindi il cast a *int*, mentre per dati di tipo *float* in virgola mobile possiamo convertirli in interi usando ad esempio il metodo *Float.floatToIntBits()*, e poi utilizzare il valore così ottenuto come codice hash.

3.2.3 Somma di componenti

Per i valori come *long* e *double*, la cui rappresentazione in bit è doppia di quella di un codice hash, bisogna agire in modo diverso. Una possibilità è quella di considerare solo una parte del dato lunga quanto il codice hash e fornire la sua interpretazione come valore intero di quella porzione di bit. In questo modo però si perde una quantità di informazione perché non si considerano tutti i bit del dato iniziale; con valori che differiscono solo per i bit non considerate, si incorre quindi in collisioni. Un'alternativa che tiene conto di tutti i bit è quella di sommare la rappresentazione intera dei bit più significativi alla rappresentazione intera dei bit meno significativi. Per un valore di tipo long ad esempio, in Java si potrebbe scrivere questo codice:

```
static int hashCode(long i) {
    return (int) ( (i>>>32) + (int) i);
}
```

3.2.4 Codice hash polinomiale

Per oggetti di lunghezza variabile che possono essere visti come tuple nella forma $(x_0, x_1, ..., x_{k-1})$, dove l'ordine degli x_i è caratterizzante, sarà necessario trovare una soluzione diversa. Si utilizza come codice hash il valore

$$x_0 a^{k-1} + x_1 a^{k-2} + ... + x_{k-2} a + x_{k-1}$$

in cui a è una costante non nulla diversa da 1; questa tecnica prende il nome di codice hash polinomiale.

3.2.5 Shift ciclico

È una variante del codice polinomiale che per ogni addendo, sostituisce l'operazione di moltiplicazione per una costante con uno spostamento di un certo numero di bit; in realtà anche in questo caso si tratta di una moltiplicazione: in rappresentazione binaria infatti lo shift verso sinistra è una moltiplicazione (divisione nel caso dello shift verso destra). Per chiarezza si fornisce un esempio di come si potrebbe utilizzare in Java lo shift ciclico per ricavare il codice hash da una stringa di caratteri.

```
static int hashCode (String s) {
    int h=0;
    for (int i=0; i<s.lenght(); i++) {
        h = (h<<5) | (h>>>27);
    h += (int) s.charAt(i);
    }
return h;
}
```

3.3 Funzioni di compressione

Come evidenziato all'inizio del secondo paragrafo, una volta generato un codice hash, è necessario trovare un nuovo valore intero ad esso associato, che però sia contenuto nell'intervallo $[0, N-1]$ che, come detto in precedenza, è l'intervallo degli indici dell'array di bucket.

La funzione che calcola questo valore e fornisce il risultato finale della funziona hash è detta funzione di compressione. Verranno ora esaminati i metodi principali di compressione

3.3.1 Divisione a modulo

Con la divisione a modulo si fa corrispondere a un intero i il valore

$$i \bmod N$$

dove N è la dimensione dell'array di bucket. Se tale dimensione è data da un numero primo, allora la funzione di compressione così definita distribuisce uniformemente i codici hash. Se N non è un numero primo però, è molto probabile che dei pattern di ripetizione nella distribuzione dei codici hash vengano replicati nella distribuzione dei codici hash, causando collisioni. Per comprendere meglio si prenda come esempio l'utilizzo di una potenza di 2 per il valore di N; il valore $A[k]$ rappresenterebbe la parte più significativa di k, indipendentemente dalla configurazione dei bit meno significativi. In generale si vuole evitare valori di N che dividano $r^k \pm a$, dove k e a sono delle costanti piccole e r è la base dell'alfabeto di parole usato; questo perché il resto della divisione modulo N non conterrebbe le informazioni sulle cifre più significative. Queste considerazioni portano a stabilire che sia opportuno scegliere N tra i numeri primi, tale che $r^k \neq \pm a \bmod N$.

3.3.2 Il metodo di moltiplicazione

Il metodo di moltiplicazione opera in due passi. In prima istanza si procede alla moltiplicazione del valore intero i per una costante A, con A appartenente all'intervallo $[0, 1]$, e dal risultato viene estratta la parte frazionaria. Successivamente si moltiplica questo valore per N e, come al punto precedente, si estrae la parte frazionaria dal risultato. L'espressione è quindi

$$\lfloor N(iA \bmod 1) \rfloor$$

Il vantaggio di questo metodo è che, a differenza del metodo di divisione, la scelta di N non è critica. Solitamente viene scelta una potenza di 2 come valore di N.

La scelta determinante invece, è quella che riguarda la costante A: anche se infatti il metodo funziona per qualsiasi valore di A (compreso nell'intervallo sopra citato), per alcuni valori di A, il metodo risulta più efficiente. Donald Knuth suggerisce che il valore migliore di A sia:

$$A \approx (\sqrt{5} - 1)/2 = \phi^{-1}$$

Tale valore corrisponde al reciproco del rapporto di sezione aurea. Knuth ha osservato infatti il fenomeno seguente: si consideri un segmento [0...1]; si evidenzino all'interno del segmento i punti corrispondenti ai valori $\phi^{-1}, 2\phi^{-1}, ...$, dove con la notazione $\{x\}$ si denota la parte frazionaria di x. Facendo riferimento alla figura 3.1 si può osservare allora che i punti si distribuiscono in maniera uniforme all'interno del segmento; nello specifico ogni punto che viene aggiunto cade all'interno di uno degli intervalli più grandi rimasti, dividendolo in sezione aurea. Applicando questo ragionamento all'intervallo dei valori ammissibili prodotti dalla funziona di compressione, si ottiene il medesimo tipo di distribuzione dei codici hash.

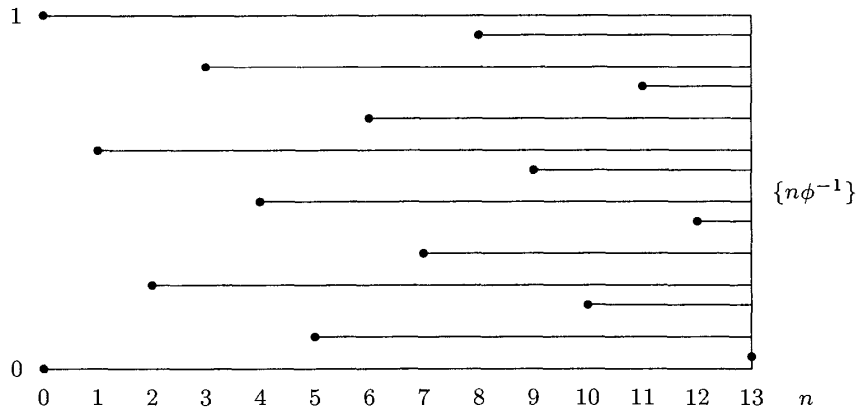

Figura 3.1: Suddivisione del segmento [0...1]

3.3.3 Il metodo MAD

Una funzione di compressione ancora più sofisticata è quella del metodo MAD ("multiply add and divide"). L'espressione è data da:

$$|Ai + B| \bmod N$$

dove, come di consueto, N è la dimensione dell'array e viene scelta tra i numeri primi; A e B sono due costanti intere ($A > 0, B > 0$) che vengono chiamate rispettivamente fattore di scala e shift; A e N sono scelte in modo che A mod N=/0 .

Come suggeriscono Goodrich e Tamassia, le prove effettuate con il metodo MAD testimoniano che questa funzione di compressione ci permette di evitare di incorrere in schemi di ripetizione per l'insieme dei codici hash generati e quindi di avvicinarci all'obiettivo di raggiungimento di una "buona" funzione hash.

3.4 Gestione delle collisioni

Per quanto efficienti si scelgano il codice hash e la funzione di compressione, il fenomeno delle collisioni è sempre possibile.

Intuitivamente si potrebbe ipotizzare che la soluzione ideale a questo problema sia la scelta di una funzione hash h appropriata; ovvero si può scegliere h in modo che distribuisca uniformemente i codici hash all'interno dell'intervallo desiderato, evitando così collisioni. Naturalmente occorre anche tenere presente che una funzione di hash deve essere deterministica: per una data chiave k in ingresso deve produrre in uscita sempre lo stesso valore $A[k]$. Inoltre occorre considerare anche che la cardinalità dell'insieme universo dei valori U è maggiore di N (dimensione dell'array di bucket); se non valesse questa condizione, e fosse invece $N > |U|$, allora non avrebbe senso utilizzare una tabella hash, perché si avrebbe uno spreco di spazio (ci sarebbero sicuramente delle posizioni libere all'interno dell'array di bucket).

Poiché $|U| > N$, vi sono sicuramente due chiavi con lo stesso codice hash; evitare del tutto le collisioni è quindi impossibile e seppure una funzione hash possa minimizzarne il numero, è necessario poterle gestire opportunamente.

Fortunatamente esistono tecniche efficaci per la risoluzione di questi conflitti; verranno ora analizzate le due principali: quella per concatenazione e quella tramite indirizzamento aperto.

3.4.1 Concatenazione

Il metodo più semplice ed efficiente per gestire le collisioni è quello di inserire tutti gli elementi che collidono nella stessa posizione dell'array di bucket, in una lista concatenata. La posizione $j = A[k]$ dell'array contiene un puntatore alla testa della lista che contiene tutti gli elementi memorizzati che corrispondono alla posizione j. In questo modo le operazioni di inserimento e rimozione che interessano una chiave k vengono delegate alla lista concatenata memorizzata alla posizione j dell'array.

Per ottimizzare lo spazio occupato e le prestazioni, se un bucket non contiene nessuna chiave, il puntatore corrispondente punterà al valore *null*; quando invece il bucket contiene un solo elemento, invece di puntare a una lista concatenata che contiene un solo elemento, punterà direttamente all'elemento stesso.

Assumendo di aver utilizzato una buona funziona hash, ci aspettiamo che ogni bucket sia di dimensione n/N. Questo valore prende il nome di fattore di carico della tabella hash (e viene di norma indicato con il simbolo λ).

3.4.2 Indirizzamento aperto

Nell'indirizzamento aperto tutti gli elementi sono memorizzati nella tabella stessa, senza l'utilizzo di strutture dati ausiliarie. Con Questo approccio si ha al più un elemento per posizione, quindi $n \leq N$, che implica $\lambda \leq 1$.

Per eseguire le operazioni comuni su una tabella hash in cui le collisioni vengono gestite in questo modo, diventa fondamentale sapere come eseguire la scansione della tabella (ad esempio quando si vuole trovare una posizione vuota in cui poter inserire un elemento). Vengono qui analizzati i principali metodi.

Scansione lineare

Al momento dell'inserimento di un nuovo valore nella tabella, data una funzione hash h, il metodo di scansione lineare usa la funzione:

$$h'(k,i) = (h(k) + i) \bmod N \qquad con\ i = 0, 1, ..., N-1$$

.

Se si cerca quindi di inserire un elemento nella posizione dell'array già occupata, l'algoritmo procederà cercando di inserirlo alla posizione $(h(k)+1) \bmod N$; se anche questa posizione risulta occupata, si procederà con quella successiva $(h(k)+2) \bmod N$, e così via, fino a trovare una posizione libera per inserire il nuovo elemento.

La scansione lineare è di facile realizzazione, ma presenta un problema conosciuto come fenomeno di agglomerazione primaria: le posizioni occupate si accumulano in lunghi tratti, aumentando il tempo medio di ricerca. Per capire meglio il motivo si consideri la situazione rappresentata in figura 3.2.

La dimensione dell'ipotetica tabella hash rappresentata è $N = 19$ e 9 posizioni di questa sono già occupate. Si prendano in esame le posizioni 8 e 11; una nuova chiave k viene inserita in quest'ultima posizione se $11 \leq h(k) \leq 15$, mentre viene inserita in corrispondenza dell'ottava se $A[k] = 8$. E' facile stabilire quindi che, al prossimo inserimento, è 5 volte più probabile che venga occupata la posizione 11 rispetto alla posizione 8, con un conseguente accumulo di posizioni consecutive occupate. La situazione si aggrava inoltre quando viene occupata la posizione 4 o 16, evento che porta all'unione di due sequenze. E' quindi evidente che le prestazioni della scansione lineare diminuiscono man mano che n si avvicina a N.

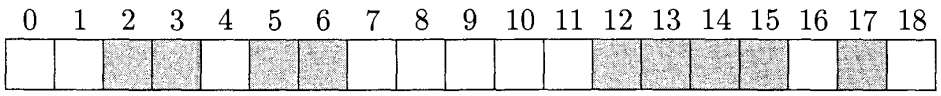

Figura 3.2: Tabella hash con alcune posizioni occupate

Scansione quadratica

Come per la scansione lineare, data una funzione hash h, la scansione quadratica usa la funzione:

$$h'(k,i) = (h(k) + ai + bi^2) \bmod N \qquad con\ i = 0, 1, ..., N-1$$

.

dove a e b sono constanti ausiliarie non nulle. Se si cerca quindi di inserire un elemento nella posizione $A[k]$ dell'array già occupata, le posizioni esaminate successivamente sono distanziate di un valore che dipende in modo quadratico dal numero i di tentativi effettuati fino a quel momento.

Con questo metodo si evita di incorrere nel fenomeno di agglomerazione primaria sopra citato, ma l'agglomerazione permane in una forma più lieve, denominata agglomerazione secondaria.

Hashing doppio

Nel caso dell'hashing doppio, la funzione utilizzata è invece:

$$h'(k,i) = (h_1(k) + ih_2(k)) \bmod N \qquad con\ i = 0, 1, ..., N-1$$

.

dove *h1* e *h2* sono due funzioni ausiliarie. La prima posizione esaminata sarà quella all'indice $h1(k)$, ma le scansioni successive, diversamente dalla scansione lineare e da quella quadratica, dipendono dalla chiave k in due modi: a seconda di essa infatti può variare sia la posizione iniziale, sia la distanza.

$h1(k)$ produce un valore compreso nell'intervallo $[0, N-1]$; $h2(k)$ invece deve generare un numero compreso nell'intervallo $[1, N-1]$ che sia relativamente primo a N. Se però, ad esempio, N è primo, $h2(k)$ può generare qualsiasi numero compreso tra 1 e $N-1$; oppure se N è una potenza m-esima di due, allora $h2(k)$ può assumere qualsiasi valore dispari maggiore di 1 e minore di 2^{m-1}. Se N e $h2(k)$ invece avessero un massimo comun divisore $d > 1$ per qualche chiave k, allora una ricerca di tale chiave

andrebbe ad esaminare solo una frazione $(1/d)$ della tabella hash.

L'hashing doppio rappresenta un ulteriore miglioramento rispetto alle scansioni lineare e quadratica dato dal fatto che viene utilizzato un numero $O(N^2)$ di sequenze di scansioni, piuttosto che $O(N)$. Questo perché, come già detto, a seconda della chiave, la posizione iniziale $h1(k)$ e la distanza $h2(k)$ possono variare in modo indipendente e ogni possibile coppia $(h1(k), h2(k))$ implica una diversa sequenza di scansione.

3.5 Analisi e confronto delle prestazioni

Vengono ora analizzati i due metodi di gestione delle collisioni, confrontandoli in base al fattore di carico $n/N = \lambda$ introdotto precedentemente. Si fa inoltre uso della notazione di *O-grande*, già utilizzata in precedenza, della quale però per chiarezza si ricorda la definizione:

Siano f e g due funzioni reali definite sul medesimo intervallo $(a, b) \setminus \{x_0\}$. Si dice che $f = O(g)$ (e si legge "f è *O-grande* di g") se esiste $M \in \mathbb{R}$, con $M \neq 0$, tale che

$$\lim_{x \to x_0} \frac{f(x)}{g(x)} = M$$

Dire che $f = O(g)$ significa circa che : "la funzione f si comporta allo stesso modo della funzione g, per x che tende a x_0".

3.5.1 Prestazioni della concatenazione

Nell'utilizzo della concatenazione il caso peggiore si manifesta quando tutte le n chiavi collidono nella stessa posizione. Il tempo di ricerca nel caso peggiore è così $O(n)$, dove n rappresenta il numero di chiavi, più il tempo utilizzato per calcolare la funzione hash. In queste condizioni il metodo in questione non sembra di certo vantaggioso. Tuttavia l'analisi che da un punto di vista statistico ci interessa maggiormente è quella del comportamento nel caso medio: anche se questo dipende da come la funzione hash distribuisce le chiavi da memorizzare, ovvero dalla funzione di compressio-

ne utilizzata, assumiamo per semplicità che la funzione verifichi l'ipotesi di uniformità semplice. Si ipotizzi inoltre che il valore della funzione hash possa essere calcolato in un tempo costante, così che il tempo richiesto per ricercare un elemento con chiave k dipenda esclusivamente dalla lunghezza della lista che si trova alla posizione $h(k)$. Consideriamo quindi il numero medio di elementi esaminati dall'algoritmo di ricerca, ovvero il numero degli elementi della lista in esame che vengono controllati per verificare se le loro chiavi sono uguali a k. Si verificano due casi, che prendiamo in esame: nel primo la ricerca non ha successo, ovvero nessun elemento della tabella ha chiave k, nel secondo la ricerca ha successo e trova un elemento con chiave k.

Si può dimostrare che in una tabella hash con collisioni risolte per concatenazione e funzione hash che verifica l'uniformità semplice, una ricerca (con o senza successo) richiede in media un tempo $O(1 + \lambda)$. Di seguito si presenta la dimostrazione del risultato appena citato per una ricerca con successo.

Dimostrazione

Nell'ipotesi di uniformità semplice della funzione hash è equamente probabile che qualunque chiave k corrisponda a una qualunque delle N posizioni dell'array. Il tempo medio di ricerca senza successo di una chiave k è così il tempo medio richiesto per analizzare per intero una delle N liste. La lunghezza media di una tale lista è data dal fattore di carico n/N, per cui mediamente il numero di elementi esaminati è λ; e il tempo totale richiesto per l'operazione (incluso il tempo necessario per calcolare $h(k)$) è $O(1 + \lambda)$.

Si consideri ora il caso della ricerca con successo.

Dimostrazione

Per trovare il numero medio di elementi esaminati, si consideri la media, sugli n elementi della tabella, di 1 più la lunghezza media della lista in cui l elemento i-esimo è aggiunto; tale lunghezza è $(i-1)/N$ e quindi il numero medio di elementi esaminati è:

$$\frac{1}{n}\sum_{i=1}^{n}\left(1+\frac{i-1}{N}\right) = 1+\frac{1}{nN}\sum_{i=1}^{n}(i-1)$$

$$= 1+\left(1+\frac{1}{nN}\right)+\left(1+\frac{(n-1)n}{2}\right)$$

$$= 1+\frac{\lambda}{2}-\frac{1}{2N}$$

Di conseguenza il tempo totale per una ricerca con successo (incluso il tempo per calcolare la funzione hash) è $O(2+\lambda/2-1/2N) = O(1+\lambda)$.

L'analisi appena esposta porta a dedurre che se il numero di posizioni nella tabella è almeno proporzionale al numero di elementi contenuti, allora si ha che il numero di chiavi n è $O(N)$ (dove, ricordiamo, N è la dimensione dell'array di bucket). Conseguentemente $\lambda = n/N = O(N)/N = O(1)$. La ricerca richiede quindi in media tempo costante.

3.5.2 Prestazioni dell'indirizzamento aperto

Per quanto riguarda l'indirizzamento aperto, al massimo si può avere un elemento per posizione, quindi $n \leq N$ che implica $\lambda \leq 1$. Assumendo l'utilizzo di una funzione hash uniforme la sequenza di scansione per ogni chiave k $\langle h(k,0), h(k,1), ..., h(k,N-1)\rangle$ è, in modo equiprobabile, una qualunque permutazione di $\langle 0, 1, ..., N-1\rangle$. Ogni possibile sequenza di scansione è cioè usata in modo equiprobabile per un'inserzione o una ricerca.

Si può inoltre dimostrare che data una tabella hash a indirizzamento aperto e assumendo l'uniformità della funzione hash, il numero medio di una ricerca senza successo vale al massimo $1/(1-\lambda)$.

Dimostrazione

In una ricerca senza successo ogni accesso tranne l'ultimo è a una posizione occupata che non contiene la chiave desiderata e l'ultima posizione considerata è vuota. Si definisca $p_i = Pr$ {esattamente i accessi a posizioni occupate}

per $i = 0, 1, 2, ...$ per $i > n$ si ha $p_i = 0$, poichè si possono trovare al più n posizioni già occupate. Per cui il numero medio di accessi è

$$1 + \sum_{i=0}^{\infty} ip_i$$

Definendo $q_i = Pr$ {almeno i accessi a posizioni occupate}, si può scrivere:

$$\sum_{i=0}^{\infty} ip_i = \sum_{i=1}^{\infty} q_i$$

Analizziamo allora il valore che può assumere q_i per $i \geq 1$: la probabilità che la prima posizione cui si accede sia occupata è n/N, per cui

$$q_1 = \frac{n}{N}$$

Usando una funzione hash uniforme, un secondo accesso, se necessario, è sulle rimanenti $N - 1$ posizioni ancora non esaminate, $n - 1$ delle quali sono occupate. Si fa un secondo accesso allora solo se il primo è ad una posizione occupata; per cui

$$q_2 = \left(\frac{n}{N}\right)\left(\frac{n-1}{N-1}\right)$$

In generale, l'i-esimo accesso è fatto solo se i primi $i - 1$ accessi sono a posizioni occupate e la posizione esaminata è in modo equamente probabile una qualunque delle restanti $N - i + 1$ posizioni, $n - i + 1$ delle quali sono occupate. Per cui

$$q_i = \left(\frac{n}{N}\right)\left(\frac{n-1}{N-1}\right)...\left(\frac{n-i+1}{N-i+1}\right)$$

$$\leq \left(\frac{n}{N}\right)^i$$

$$= \lambda^i$$

per $i = 1, 2, ..., n$, poichè $(n - j)/(N - j) \leq n/N$ se $n \leq N$ e $j \geq 0$. Dopo

n accessi tutte le n posizioni occupate sono state esaminate e non saranno più scandite, quindi $q_i = 0$ per $i > n$.

Tornando all'uguaglianza tra le due sommatorie, data l'ipotesi che $\lambda < 1$, il numero medio di accessi di una ricerca senza successo è

$$1 + \sum_{i=0}^{\infty} ip_i = 1 + \sum_{i=1}^{\infty} q_i$$

$$\leq 1 + \lambda + \lambda^2 + \lambda^3 + ...$$
$$= \frac{1}{1-\lambda}$$

Al risultato possiamo fornire un'interpretazione intuitiva: un accesso viene sempre effettuato, con probabilità λ ne è necessario un secondo, con probabilità λ^2 ne è necessario un terzo ... e così via.

Per una ricerca con successo invece, ipotizzando di usare una funzione hash uniforme e assumendo che ogni chiave sia ricercata in modo equiprobabile, il numero medio di accessi può essere al massimo $\frac{1}{\lambda} ln \frac{1}{1-\lambda}$.

Prima della dimostrazione di tale risultato si ontroduce un corollario:

Corollario

L'inserzione di un elemento in una tabella hash ad indirizzaento aperto con fattore di carico λ richiede al massimo $1/(1-\lambda)$ accessi in media, nell'ipotesi dell'utlizzo di una funzione hash uniforme.

Dimostrazione

La ricerca di una chiave k segue la medesima sequenza di scansione seguita per l'inserzione dell'elemento con chiave k. Dal corollario precedente se k è la $(i+1)$-esima chiave inserita nella tabella, il numero medio di accessi compiuti in una ricerca non può superare il valore $1/(1-i/N) = N/(N-i)$. Eseguendo la media su tutte le n chiavi nella tabella si ottiene il numero medio di accessi in una ricerca senza successo come limite superiore:

$$\frac{1}{n} \sum_{i=0}^{n-1} \frac{N}{N-i} = \frac{N}{n} \sum_{i=0}^{n-1} \frac{1}{N-i}$$

$$= \frac{1}{\lambda}(H_N - H_{N-n})$$

dove $H_i = \sum_{j=1}^{i} \frac{1}{j}$ è l'i-esimo numero armonico. Per le sue proprietà si ha:

$$\frac{1}{\lambda}(H_N - H_{N-n}) = \frac{1}{\lambda} \sum_{k=N-n+1}^{N} \frac{1}{k}$$

$$\leq \frac{1}{\lambda} \int_{N-n}^{N} \frac{1}{x}\, dx$$

$$\frac{1}{\lambda} ln(\frac{N}{N-n})$$

$$\frac{1}{\lambda} ln \frac{1}{1-\lambda}$$

3.5.3 Funzione hash universale

Dal punto di vista dell'analisi prestazionale, il caso peggiore si presenta quando ci sono n chiavi che corrispondono alla stessa posizione dell'array, provocando un tempo medio di ricerca $O(N)$. L'unica soluzione possibile è quella di scegliere una funzione hash in modo casuale, in maniera che sia indipendente dalle chiavi effettivamente memorizzate. Questo approccio è chiamato hashing universale e consiste nel selezionare, al momento del calcolo, la funzione hash da una classe di funzioni opportunamente definite.

Si può dimostrare che, se h è scelta da un insieme universale di funzioni hash ed è usata per l'inserimento di n chiavi in una tabella di dimensione N, dove $n \leq N$, allora il numero medio di collisioni che coinvolge una particolare chiave x è minore di 1.

Dimostrazione

Per ogni coppia di chiavi distinte y,z, sia c_{yz} una variabile casuale che è uguale a 1 se y e z collidono usando h (cioè $h(y) = h(z)$), e 0 altrimenti. Per definizione una coppia di chiavi collide con probabilità $1/N$; si può scrivere quindi:

$$E[c_{yz}] = 1/N$$

Sia C_x il numero totale di collisioni che coinvolgono la chiave x in una tabella hash T di dimensione N contenente n chiavi; allora

$$E[C_x] = \sum_{y \in T, y \neq x} E[c_{xy}] = \frac{n-1}{N}$$

Poiché $n \leq N$, si ha che $E[C_x] < 1$.

3.6 Rehashing

Nel corso dei paragrafi precedenti è emerso più volte che per un funzionamento efficiente della tabella hash deve essere $\lambda = n/N \leq 1$. M. Goodrich e R. Tamassia, dopo diverse prove, hanno fornito una stima migliore di questo limite superiore: per l'indirizzamento aperto è conveniente avere $\lambda < 0.5$, mentre per la concatenazione $\lambda < 0.9$.

In entrambe le strategie è quindi necessario controllare che il fattore di carico si mantenga sotto un certo valore, nonostante gli inserimenti di nuovi elementi in tabella. Per consentire ciò, quando λ supera il valore richiesto si ridimensiona la tabella. In genere si sceglie che la nuova dimensione della tabella sia almeno il doppio di quella precedente. Una volta allocato il nuovo array di bucket si definisce una nuova funzione hash calcolando i parametri adatti per la nuova dimensione. Si reinserisce quindi ogni valore del vecchio array in quello nuovo utilizzando la funzione appena calcolata.

Anche se le operazioni di ridimensionamento e riallocamento possono sembrare laboriose e sconvenienti dal punto di vista prestazionale, l'utilizzo del rehashing è comunque una scelta conveniente. Il costo dell'operazione di rehash viene infatti ammortizzato dalla ridistribuzione ottimizzata dei valori nel nuovo array.

Capitolo 4

Obiettivi del lavoro

Introduzione

La scelta dell'implementazione di una tabella hash per *DiESeL* deriva sostanzialmente da una necessità ben precisa: la gestione ottimale dei "nodi di *DiESeL*".

Questa astrazione rappresenta un nodo della rete servendosi della struttura dati *IDiESeLNode*; essa contiene i riferimenti all'*InetAddress* e ai numeri delle 2 porte utilizzate dal nodo per comunicare; il primo fornisce l'indirizzo IP ed eventualmente il corrispondente nome host; i rimanenti permettono di differenziare due nodi nella medesima rete (quindi con medesimo indirizzo IP, ma porte differenti).

Come illustrato nel capitolo 2, DiESeL gestisce una tabella in cui vengono memorizzate le informazioni della rete; in queste informazioni ciascun *IDiESeLNode* della rete viene messo in relazione con la lista degli *IDiESeLNode* a lui vicini. Come si vedrà nel capitolo 5 quindi, ciascun elemento della tabella non deve essere rappresentato da un nodo, bensì dalla coppia costituita da un nodo e dall'insieme dei nodi conosciuti. Per consentire le operazioni di inserimento, ricerca e rimozione degli elementi di tale tabella è quindi necessario utilizzare una struttura dati che consenta un valido confronto di tali elementi.

Precedentemente agli interventi di modifica oggetto di questa tesi, i

cui dettagli saranno esaminati nel capitolo successivo, *DiESel* utilizzava la classe *Neighborhood* come astrazione della tabella di informazione. All'interno di tale classe veniva creata e utilizzata una tabella hash e, per tale struttura dati, ci si serviva dell'implementazione fornita da Java: la classe *Hashtable*, estensione della classe *Dictionary*, appartenente al pacchetto *java.util*.

Con questo approccio ogni volta che la tabella hash doveva compiere un'operazione su un nodo, veniva invocato il metodo *hashCode()* generico ereditato dalla classe *IDiESeLNode*, in quanto estensione della classe *Object*. Il comportamento del metodo però era simile a quello descritto al paragrafo 3.2.1 (comportamento, come visto, da evitare): il calcolo del codice hash si basava sull'*ObjectID* anzicchè sul significato semantico dell'oggetto *IDiESeLNode*. In questo modo quando era necessario confrontare due nodi, esaminando i due codici hash corrispondenti si ottenevano risultati non veritieri. Capitava spesso cioè che due nodi con le stesse caratteristiche (stesso indirizzo IP e stesse porte utilizzate) avessero associato un codice hash differente. Questo naturalmente portava a risultati non desiderati ogni qual volta si effettuavano operazioni di inserimento o rimozione (ma anche ricerca) all'interno della tabella.

4.1 Le scelte effettuate

Per fornire quindi agli sviluppatori del team PariPari dei metodi di servizio affidabili, si è deciso di implementare una nuova tabella hash. E' stata così creata una nuova interfaccia *IDiESeLHashTable* da utilizzare al posto della tabella hash di Java, con la rispettiva classe che la implementava; naturalmente poi la classe *Neighborhood* è stata adattata alla nuova implementazione. Considerando quanto esposto al capitolo 3 si è deciso di usare, per l'implementazione della nuova tabella hash, la tecnica di concatenazione. In questo modo ad ogni posizione dell'array (che rappresenta l'ossatura della tabella) viene inserita una lista concatenata in cui inserire gli oggetti con il medesimo codice hash. Per quanto riguarda la funzione si è scelto di ricavare il codice hash utlizzando i numeri delle porte e le

quattro cifre dell'indirizzo IP di ciascun *IDiESeLNode*. In questo modo con il codice hash si è sicuri di poter indentificare il nodo corrispondente in maniera univoca e di poter gestire opportunamente i confronti tra nodi diversi. Per la funzione di compressione invece, si è optato per il metodo "Multiply and Divide" in quanto, al momento della realizzazione del progetto, è sembrato il miglior compromesso tra efficienza e difficoltà di implementazione. Infine alla tabella è stata fornita l'abilità di ridimensionarsi automaticamente tramite la procedura di rehashing, così da poter mantenere efficienti i metodi di accesso anche dopo diversi inserimenti. Nel capitolo successivo sarà analizzato in dettaglio il codice sviluppato secondo le scelte appena esposte.

Capitolo 5

Realizzazione

Introduzione

Le considerazione esposte al capitolo precedente hanno portato allo sviluppo di 2 classi (con relative interfacce), necessarie per implementare opportunamente una tabella hash:

paripari.utilities.diesel.DiESeLHashTableEntry

paripari.utilities.diesel.DiESeLTashTable

Le classi in esame hanno richiesto però che si fornisse una versione modificata della classe *Neighborhood*. Si è deciso allora di creare anche la classe

paripari.utilities.diesel.NeighborhoodV2,

la quale condivide assieme a *Neighborhood* l'interfaccia implementata *INeighborhood*.

Nel corso di questo capitolo verrà esaminato nel dettaglio il codice delle classi sopracitate, spiegandone il funzionamento e evidenziando le tecniche utilizzate per applicare le considerazioni fatte al capitolo precedente.

5.1 DiESeLHashTableEntry

La prima classe che prendiamo in esame è anche la più semplice e ha la scopo di rappresentare, in maniera astratta, gli elementi della tabella hash. In essa si trovano 2 variabili globali; la variabile node è di tipo *IDiESeLNode*, che rappresenta l'astrazione di un nodo all'interno della rete *DiESeL*; la variabile list invece utilizza la classe *ArrayList* della libreria *java.util*, la quale permette di rappresentare la lista di nodi conosciuti. Segue il costruttore della classe, che inizializza le due variabili globali sopracitate, in modo da creare un elemento che andrà a far parte della tabella hash è conterrà il riferimento al nodo desiderato e ai nodi da esso conosciuti.

```
public class DiESeLHashTableEntry implements
    IDiESeLHashTableEntry {

    private ArrayList<IDiESeLNode> list;
    private IDiESeLNode node;
    public DiESeLHashTableEntry(IDiESeLNode n ,
        ArrayList<IDiESeLNode> l){
        node=n;
        list=l;
    }
    public ArrayList<IDiESeLNode> getList() {
        return list;
    }
    public IDiESeLNode getNode() {
        return node;
    }
}
```

La classe è completata da due metodi di accesso che permettono di ottenere i valori delle due variabili globali, altrimenti inaccessibili da una classe esterna perché dichiarate private.

5.2 DiESeLHashTable

DieselHashTable rappresenta la vera e propria tabella hash e viene .dichiarata in questo modo:

```
public class DiESeLHashTable implements
    IDiESeLHashTable
```

Come tale quindi necessità di due variabili globali, *size* e *bucketArr*.

```
private int size;
private LinkedList<DiESeLHashTableEntry>[] bucketArr;
```

La prima, come si può intuire, fornisce la dimensione della tabella; la seconda rappresenta un array che per ogni posizione contiene un riferimento a una lista concatenata. La lista viene rappresentata tramite la classe *LinkedList* della libreria *java.util*

Seguono le 3 variabili utilizzate dal il metodo MAD per la funzione di compressione. La prima rappresenta la dimensione dell'array di bucket, mentre le rimanenti sono le 2 costanti intere positive utilizzate nella formula, che in questo caso sarà quindi $|ai + b|$ mod bas

```
private int bas;
private int a=0;
private int b=0;
```

Il costruttore predefinito inizializza le 3 variabili sopracitate: a e b vengono scelti casuali, utilizzando il metodo *random()* della classe *Math* di java; la dimensione dell'array viene inizializzata a 3.

Si noti come *bas* definisca la dimensione dell'array, che viene inizializzata quando si invoca il costruttore della tabella; tale dimensione rimane

invariata fino a che la tabella non viene ridimensionata dal metodo *rehash()*, i cui dettagli saranno discussi più avanti nel corso del capitolo. La variabile *size* invece è una variabile che indica il numero di elementi presenti nella tabella e viene aggiornata ad ogni operazione di inserimento o rimozione.

Successivamente si inizializza l'array di bucket, che, come detto in precedenza, contiene un puntatore a una lista concatenata per ogni sua posizione. Ecco come risulta nel complesso il metodo costruttore:

```
public DiESeLHashTable(){

    bas=3;
    while ( (a==0) || (a % bas == 0) )
        a = (int) (Math.random()*10);
    b = (int)Math.random()*10;
    size=0;
    bucketArr = new LinkedList[bas];
    }
```

Per consentire l'operazione di rehashing si introduce un secondo costruttore, questa volta privato; può quindi essere invocato solo dagli altri metodi di questa classe (il costruttore predefinito invece è dichiarato public); il metodo è pressoché identico al precedente, fatta eccezione per l'utilizzo del parametro intero *newbas*, che viene passato quando si invoca il metodo: esso determina la dimensione desiderata per l'array della nuova tabella hash. L'utilità di questo metodo diverrà più chiara più avanti quando si esaminerà il metodo *rehash()*.

```
private DiESeLHashTable(int newbas){

    bas = newbas;
    while ( (a==0) || (a % bas == 0) )
```

```
            a = (int) (Math.random()*10);
        b = (int)Math.random()*10;
        size=0;
        bucketArr = new LinkedList[bas];
}
```

Il metodo successivo è usato per inserire i nuovi elementi all'interno della tabella. Essi sono coppie chiave-valore, dove la chiave è rappresentata da un nodo, mentre il valore è costituito dalla lista dei nodi da esso conosciuti.

Il codice delle prime 5 righe del metodo si ripresenta in più punti all'interno della classe; si fornisce quindi di seguito una precisazione su tali istruzioni, che naturalmente non verrà ripetuta ogni qual volta si trovino le stesse righe di codice più avanti nel corso del capitolo.

```
IDiESeLNodeExtended nodeExt;
    if (node instanceof DiESeLNode)
        nodeExt = new DiESeLNodeExtended(node);
    else
        nodeExt = (DiESeLNodeExtended) node;
```

Come detto in precedenza la libreria *DiESeL* è stata concepita in modo da essere in grado di gestire oggetti di tipo *IDiESelNode*. Succcessivamente però sono stati introdotti anche gli *IDiESelNodeExtended*; questi oggetti come si può intuire estendono la classe *IDiESelNode* arricchendone le funzionalità. Infatti sono costiutiti da un riferimento a un nodo *IDiESelNode*, e da un certo numero di attributi aggiuntivi, chiamati *Features*. Queste servono sostanzialmente a fornire la possibilità di specificare delle caratteristiche aggiuntive che un certo nodo possiede, che possono variare da nodo a nodo: ad esempio un server casalingo può avere diverse prestazioni e funzionalità rispetto a un server di una banca. Al momento della stesura del codice di queste classi, le funzionalità offerte dalle *Features* erano anco-

ra in fase di testing; di conseguenza era necessario poter gestire all'interno della tabella sia i "nodi semplici" che i "nodi estesi". In realtà si è scelto di gestire, all'interno della tabella solo oggetti di tipo *IDiESeLNodeExtended*; quando si ha a che fare con un *IDiESeLNode*, prima di effettuare qualsiasi operazione su tale nodo, lo si "converte" in un nodo esteso. Analizziamo meglio il listato esposto precedentemente.

In prima fase, tramite il comando *instanceof*, si controlla il tipo di nodo passato al metodo; se questo non è un' "nodo esteso" allora ne viene creato uno nuovo utilizzando il costruttore della classe *DiESeLNodeExtended*; viene passato come parametro il nodo semplice e nella variabile nodeExt viene salvato un nodo esteso con il riferimento al nodo semplice e le feature impostate a *null*. In caso contrario, in ingresso il metodo ha ricevuto un oggetto *DiESeLNodeExtended*; si noti che la dichiarazione del metodo prevede tra i parametri un oggetto *DiESeLNode*, ma essendo la classe *DiESeLNodeExtended* una sottoclasse di *DiESeLNode*, può essere passato come parametro appunto un oggetto *DiESeLNodeExtended*. In questo modo è sufficiente un semplice cast a *DiESeLNodeExtended*, che è lecito in quanto in realtà l'oggetto *node* è proprio di tipo *DiESeLNodeExtended*.

Successivamente viene calcolato il codice hash dell'elemento, tramite l'invocazione del metodo *hashCode()*, che verrà analizzato più avanti nel corso del capitolo. Si noti che il codice hash viene calcolato sulle informazioni contenute del "nodo semplice", in quanto il metodo *getNode()* invocato sull'oggetto *nodeExt*, restituisce il nodo privo di *features*; queste, come spiegato in precedenza, sono opzionali e non vanno a modificare l'identità del nodo all'interno della rete.

```
public void put(IDiESeLNode node, ArrayList<
    IDiESeLNode> list){

    IDiESeLNodeExtended nodeExt;
    if (node instanceof DiESeLNode)
        nodeExt = new DiESeLNodeExtended(node);
    else
```

```
        nodeExt = (DiESeLNodeExtended) node;
    int index = hashCode(nodeExt.getNode());
```

Il metodo prosegue nella sua esecuzione controllando che la posizione *index* dell'array generata dal metodo *hashCode()* sia vuota; in tal caso viene creata, in corrispondenza di tale posizione, una nuova lista concatenata di *DiESeLHashTableEntry* (astrazione degli elementi della tabella hash).

```
if (bucketArr[index] == null){
    bucketArr[index] = new LinkedList<
        DiESeLHashTableEntry>();
}
```

In caso contrario è necessario inserire l'elemento all'interno della lista concatenata esistente. Prima però, tramite il metodo *find* (di cui si parlerà successivamente), si cerca in tale lista il nodo che si vuole inserire; se il metodo non restituisce *null*, allora significa che tale nodo è già presente nella tabella. Comincia quindi il processo di aggiornamento dei nodi conosciuti: si copiano i nodi presenti nella "vecchia" lista all'interno di una lista temporanea di tipo *ArrayList*, che viene aggiornata con i nodi "nuovi" da inserire tra quelli conosciuti. Successivamente si rimuove l'elemento di tipo *DiESeLHashTableEntry* oggetto dela modifica (la modifica diretta non è possibile) e lo si sostituisce con un nuovo *DiESeLHashTableEntry* che contiene il nodo passato come parametro al metodo *put* (cioè il nuovo nodo da inserire in tabella) e la lista aggiornata dei nodi da esso conosciuti. Infine si stampa a video la lista aggiornata dei nodi conosciuti

```
DiESeLHashTableEntry tmpentry = find(nodeExt,
    bucketArr[index]);
if (tmpentry != null){
    ArrayList<IDiESeLNode> tmpflist = tmpentry.
        getList();
```

```
        for(int i=0;i<list.size();i++){
            IDiESeLNode tmp = list.get(i);
            if (!tmpflist.contains(tmp) )
                tmpflist.add(tmp);
        }
        bucketArr[index].remove(tmpentry);
        DiESeLHashTableEntry e = new
            DiESeLHashTableEntry(nodeExt,tmpflist);
        bucketArr[index].addLast(e);
        String output = new String("[DiESeL] 
            DiESeLHashTable >>  Updated Friends List 
            of node "+nodeExt.getNode().getHostAddress
            ()+" is : #1: "+tmpflist.get(0).
            getHostAddress()+"\n");
        for(int i=1; i<tmpflist.size();i++){
            output +=                "
                                    
                                    
                #"+(i+1)+": "+
                tmpflist.get(i).getHostAddress()+"\n";
        }
        System.out.println(output);
    }
```

Nel caso invece in cui il nodo non venga trovato all'interno della lista concatenata, si procede al suo inserimento invocando il metodo *addLast* sulla lista stessa. Dopo aver incremento la variabile size si procede al controllo del fattore di carico λ; si verifica cioè se è necessario un rehashing della tabella. Questo si ottiene con l'impiego dei metodi *needsRehash()* e *rehash()*, dei quali si rimanda la spiegazione. Si noti come il controllo su λ e l'eventuale rehashing non venga effettuato nella porzione di codice vista in precedenza; in quel caso infatti, con "solo" l'aggiornamento della lista dei nodi conosciuti, λ non subisce nessuna variazione (il numero di oggetti di tipo *DiESeLHashTableEntry* nella tabella non varia.

```
        else{
            DiESeLHashTableEntry e = new
                DiESeLHashTableEntry(nodeExt, list);
            bucketArr[index].addLast(e);
            size++;
            if(needsRehash())
                rehash();
        }
}
```

Il successivo metodo put, a differenza di quello precedente, è privato e accetta come parametro un elemento di tipo *DiESeLHashTableEntry* ed è un metodo ausiliario utilizzato dal metodo *rehash()*.

```
private void put(DiESeLHashTableEntry entry){

    put(entry.getNode(),entry.getList());
}
```

Segue il metodo *find*, impiegato per la ricerca di un nodo all'interno della lista concatenata presente in corrispondenza di una posizione del bucket di array; esso riceve come input un nodo *DiESeL* e una lista di elementi *DiESeLHashTableEntry*; restituisce come output un oggetto di tipo *DiESeLHashTableEntry*. In prima fase si verifica che *list* non punti a *null*; se così fosse infatti, significherebbe che si vuole effettuare una ricerca in una posizione vuota dell'array (dove appunto non c'è nessuna lista concatenata); è quindi ovvio che se tale lista non è presente, il nodo cercato non è presente all'interno della tabella. Una volta compiuta questa verifica, tramite l'utilizzo di un iteratore, si scandisce la lista fino a trovare l'elemento *DiESeLHashTableEntry* il cui nodo equivale a quello cercato.

```
private DiESeLHashTableEntry find(IDiESeLNode node,
    LinkedList<DiESeLHashTableEntry> list){

    IDiESeLNodeExtended nodeExt = (DiESeLNodeExtended
        )node;
    if (list != null){
        ListIterator<DiESeLHashTableEntry> iter =
            list.listIterator();
        while (iter.hasNext()){
            DiESeLHashTableEntry temp = iter.next();
            IDiESeLNodeExtended tempExt = (
                DiESeLNodeExtended)temp.getNode();
            if (tempExt.getNode().equals(nodeExt.
                getNode()))
                return temp;
        }
    }
    return null;
}
```

Oltre all'inserimento di un nodo, la tabella deve essere in grado di effettuare anche le operazioni di rimozione; il compito è svolto dal metodo *remove*, il quale, una volta ottenuto il codice hash per conoscere la posizione all'interno dell'array del nodo da rimuovere, verifica che il nodo sia realmente presente nella lista concatenata alla posizione corrispondente. Il valore che il metodo restituisce è di tipo booleano: vero se la rimozione è avvenuta, falso viceversa (ovvero il nodo non è stato trovato); nel primo caso viene quindi decrementata la variabile *size*, nel secondo il metodo fallisce silenziosamente.

```
public void remove(IDiESeLNode node){

    IDiESeLNodeExtended nodeExt;
```

```
        if (node instanceof DiESeLNode)
            nodeExt = new DiESeLNodeExtended(node);
        else
            nodeExt = (DiESeLNodeExtended) node;
        int index = hashCode(nodeExt.getNode());
        boolean tmp = bucketArr[index].remove(find(
           nodeExt, bucketArr[index]));
        if ( tmp == true)
            size--;
}
```

Il metodo *get* riceve un nodo come parametro e restituisce la lista dei nodi conosciuti; per fare ciò genera il codice hash corrispondente per individuare la posizione dell'array in cui ricercare il nodo, operazione, come visto in precedenza, che viene delegata al metodo *find*. Sul risultato fornito da quest'ultimo metodo infine, si invoca il metodo *getList()* che restituisce la lista di nodi conosciuti.

```
public ArrayList<IDiESeLNode> get(IDiESeLNode node){

        IDiESeLNodeExtended nodeExt;
        if (node instanceof DiESeLNode)
            nodeExt = new DiESeLNodeExtended(node);
        else
            nodeExt = (DiESeLNodeExtended) node;
        int index = hashCode(nodeExt.getNode());
        return find(nodeExt, bucketArr[index]).getList();
}
```

Il metodo *hashCode()* seguente sovrascrive il metodo ereditato dalla classe Object, per i motivi esaminati nel capitolo 2. Come anticipato nel capitolo 4, per il calcolo del codice hash è impiegata la tecnica del-

la concatenazione in combinazione con il metodo MAD e il valore viene calcolato utlizzando i numeri delle porte e le quattro cifre dell'indirizzo IP dell'*IDiESeLNode*. Tramite i metodi *getComPort()* e *getPort()* si ottengono le porte del nodo; tramite invece l'invocazione di *getInetAddress).toString()* si ottiene l'*InetAddress* convertito in stringa di caratteri, dal quale si estrae l'indirizzo IP. Da questo si estraggono a loro volta le quattro cifre dell'indirizzo, che vengono convertite in valore intero e sommate. Infine si somma il risultato ai numeri delle porte e si applica al valore ottenuto la funzione di compressione.

```
private int hashCode(IDiESeLNode n){

    int hash = n.getComPort() + n.getPort();
    String temp1 = n.getInetAddress().toString();
    int i = temp1.indexOf("/");
    String address = temp1.substring(i+1);
    for(int j=0;j<3;j++){
        int index = address.indexOf(".");
        hash += Integer.parseInt( address.substring
            (0,index) );
        address = address.substring(index+1);
    }
    hash += Integer.parseInt(address);
    return ((a*hash + b) % bas);
}
```

Per mantenere il fattore di carico λ sotto un certo valore, si è visto in precedenza che ad ogni inserimento di un nuovo elemento nella tabella, viene invocato il metodo *needsRehash()*; tale metodo verifica che, dopo un inserimento, il fattore di carico si mantenga sotto il valore 1 e restituisce il valore *true* in caso contrario.

```
private boolean needsRehash(){

    if ( (size / bas) >= 1 )
        return true;
    else
        return false;
}
```

Se necessario quindi si procede al rehash; in prima fase si crea una nuova tabella temporanea di dimensione doppia rispetto a quella originale; successivamente si scandiscono le posizioni non vuote del bucket di array e si copia ciascun contenuto in una lista concatenata chiamata *oldlist*. Questa operazione, apparentemente laboriosa, è stata ideata per poter poi copiare gli elementi dalla lista concatenata alla tabella hash temporanea creata a inizio metodo; l'operazione è agevole perché è sufficiente utilizzare un iteratore che scandisca la lista concatenata *oldlist* per procedere alla copia. Infine vengono aggiornati le variabili globali, in maniera che puntino alla variabili della nuova tabella hash creata.

```
public void rehash(){

    DiESeLHashTable tmp = new DiESeLHashTable(bas*2);
    LinkedList<DiESeLHashTableEntry> oldlist = new
       LinkedList<DiESeLHashTableEntry>();
    for(int i=0;i<bucketArr.length;i++){
        LinkedList<DiESeLHashTableEntry> list =
           bucketArr[i];
        if (list != null){
            ListIterator<DiESeLHashTableEntry> iter =
                list.listIterator();
            while(iter.hasNext())
                oldlist.add(iter.next());
        }
```

```
        }
        ListIterator<DiESeLHashTableEntry> olditer =
            oldlist.listIterator();
        while(olditer.hasNext() )
            tmp.put(olditer.next());
        size = tmp.size;
        a = tmp.a;
        b =tmp.b;
        bas = tmp.bas;
        bucketArr = tmp.bucketArr;
}
```

Per concludere l'implementazione della tabella hash seguono dei metodi ausiliari. Tali metodi, in realtà, sono utilizzati dalla classe *NeighborhoodV2*, di conseguenza la loro utilità sarà chiarita quando verrà esaminata tale classe nel paragrafo successivo.

Il metodo *keys* scandisce l'intero l'array (esaminando in successione le liste concatenate non vuote) per creare un oggetto di tipo *Enumeration* che contenga tutti gli elementi della tabella; per fare ciò utilizza un vettore come contenitore intermedio che gli permetto di copiare gli elementi *DiESeLHashTableEntry* dall'array di bucket all'*Enumeration keys*.

```
public Enumeration<IDiESeLNode> keys(){

        Vector<IDiESeLNode> vect = new Vector<IDiESeLNode
            >();
        for(int i=0;i<bucketArr.length;i++){
            LinkedList<DiESeLHashTableEntry> list =
                bucketArr[i];
            if (list != null){
                ListIterator<DiESeLHashTableEntry> iter =
                    list.listIterator();
                while (iter.hasNext()){
```

```
                    vect.add(iter.next().getNode());
                }
            }
        }
        return vect.elements();
    }
```

clear() è utilizzato per svuotare la tabella, rimuovendo tutti gli elementi dalle liste concatenate.

```
    public void clear() {

        for (int i=0;i<bucketArr.length;i++){
            LinkedList<DiESeLHashTableEntry> tmp =
                bucketArr[i];
            if (tmp != null)
                tmp.clear();
        }
    }
```

contains() permette di sapere se un certo nodo è presente nella tabella. Infatti, una volta generato il codice hash corrispondente, invoca il metodo *find*; se questo restiruisce un valore diverso da *null*, allora il nodo cercato è presente nella tabela, e il metodo *contains* restituisce il valore *true*. Naturalmente, nel caso il metodo *find* dovesse restituire il valore *null*, *contains* restituirebbe il valore *false*.

```
    public boolean contains(IDiESeLNode node){

        IDiESeLNodeExtended nodeExt;
        if (node instanceof DiESeLNode)
            nodeExt = new DiESeLNodeExtended(node);
```

```
        else
            nodeExt = (DiESeLNodeExtended) node;
        int index = hashCode(nodeExt);
        DiESeLHashTableEntry e = find(nodeExt, bucketArr[
            index]);
        if (e != null)
                return true;
        return false;
}
```

Il metodo *size()* restituisce il valore della variabile privata *size*, che fornisce il numero di elementi presenti nella tabella (non la dimensione dell'array di bucket, come precisato a inizio paragrafo). Il metodo *isEmpty()* invece serve per determinare se la tabella hash è vuota o presenta almeno un elemento memorizzato al suo interno.

```
public int size(){
    return size;
}

public boolean isEmpty(){
    if (size==0)
        return true;
    else
        return false;
}
```

5.3 NeighborhoodV2

La classe *NeighborhoodV2* prende spunto dalla classe *Neighborhood*. Entrambe infatti implementano la stessa interfaccia *INeighborhood*. La sua

dichiarazione è quindi:

```
public class NeighborhoodV2 implements INeighborhood
```

Di seguito si riporta il codice dei metodi che sono stati modificati rispetto alla "vecchia" versione che, ricordiamo, astraeva il concetto di tabella hash sfruttando la classe *Hashtable* di Java.

Un aspetto comune ai metodi pubblici che seguono sono le prime due righe di codice: queste controllano che ai metodi non vengano passati come parametri dei valori *null*; in tal caso lanciano una *IllegalArgumentException()*, che poi dovrà essere cattturata nel codice di DiESeL che farà uso di tali metodi.

L'insieme delle variabili globali in questo caso è costituito da una sola variabile, che chiameremo *nodes*, textitDiESeLHashTable, classe descritta nel paragrafo precedente. Il costruttore della classe textitNeighborhoodV2 inizializza poi un'istanza di tale classe.

```
private DiESeLHashTable nodes;

public NeighborhoodV2(){
    nodes = new DiESeLHashTable();
}
```

Seguono i metodi *add* e *addOne*. Il primo è utilizzato per l'inserimento di un nuovo elemento in tabella, formato da un nodo *IDiESeL* e da una lista di *IDiESeLNode* conosciuti; il secondo invece permette di inserire nella tabella un'elemento costituito dalla coppia di due *IDiESelNode*. Nel primo caso, i valori ricevuti dal metodo come parametri vengono semplicemente passati al metodo *put* che viene invocato sulla tabella *nodes*.

Il metodo successivo è utilizzato quando si vuole inserire all'interno della tabella un nodo che annovera un solo altro nodo tra quelli da esso conosciuti. E' necessario quindi creare una lista contenente un solo ele-

mento (l'unico nodo conosciuto), prima di poter invocare il metodo *add* precedentemente descritto.

```
public void add(IDiESeLNode node, ArrayList<
    IDiESeLNode> friends){

    }
    nodes.put(node,friends);
}

public void addOne(IDiESeLNode node, IDiESeLNode
    friend){

    if( (node==null)||(friend==null) ){
        throw new IllegalArgumentException();
    }
    ArrayList<IDiESeLNode> friends = new ArrayList<
        IDiESeLNode>();
    friends.add(friend);
    add(node,friends);
}
```

Il metodo *contains* verifica che la tabella contenga un certo nodo; delega semplicemente l'operazione al metodo *contains* della classe *DiESeLHash-Table*

```
public boolean contains(IDiESeLNode node){

    if( node==null ){
        throw new IllegalArgumentException();
    }
    return nodes.contains(node);
}
```

Il metodo successivo, *containsNeghbor*, verifica che un certo nodo, chiamato *ip*, contenga, nell'elenco dei nodi "conosciuti", il nodo *friend*. Ovviamente la prima verifica da eseguire è che all'interno della tabella ci sia il nodo *ip*(se non c'è il nodo non c'è neanche una lista in cui cercare il nodo *friend*). Successivamente si salva in una lista i nodi conosciuti dal nodo *ip*, e all'interno di essa si cerca il nodo *friend*. Se questo viene trovato allora il metodo restituisce il valore *true*. In caso contrario *false*.

```
public boolean containsNeighbor(IDiESeLNode ip,
    IDiESeLNode friend){

      if( (ip==null)||(friend==null) ){
          throw new IllegalArgumentException();
      }
      if(!this.contains(ip))
          return false;
      ArrayList<IDiESeLNode> tmp = nodes.get(ip);
      return tmp.contains(friend);
}
```

getNeighbors è un metodo di accesso usato per ottenere un *ArrayList* di *IDiESeLNode* che contiene la lista dei nodi conosciuti da un certo nodo *node* fornito in ingresso come parametro. Prima di delegare il compito al metodo *get* della classe *DiESeLHashTable*, il metodo verifica che *node* sia realmente presente nella tabella; in caso contrario restituisce una lista vuota.

```
public ArrayList<IDiESeLNode> getNeighbors(
    IDiESeLNode node){

      if( node==null ){
          throw new IllegalArgumentException();
```

```
    }
    if (this.contains(node)){
        return nodes.get(node);
    }
    else{
        return new ArrayList<IDiESeLNode>();
    }
}
```

Il metodo successivo fornisce il numero di nodi conosciuti da un certo nodo *node*. Il valore viene ottenuto invocando il metodo *size* sulla lista dei nodi, a sua volta ottenuta tramite il metodo *get* di *DiESeLHashTable*. Se l'elemento *node* non è presente in tabella, restituisce il valore 0.

```
public int neighborsCount(IDiESeLNode node){

    if( node==null ){
        throw new IllegalArgumentException();
    }
    if(this.contains(node)){
        return nodes.get(node).size();
    }
    else{
        return 0;
    }
}
```

I due metodi successivi sono utilizzati per rimuovere dei nodi dalla tabella. Nel primo si cerca un dato nodo *node* fornito come parametro e, se presente all'interno della tabella, lo si rimuove delegando l'operazione al metodo *remove* della classe *DiESeLHashTable*.

Il secondo riceve una coppia di *IDiESeLNode* come parametro, *node* e *friendTodel*. La sua invocazione consente di rimuovere il nodo *friendTodel* dalla lista dei nodi conosciuti da *node*. Dopo aver salvato tale lista in una variabile temporanea, ricerca al suo interno il nodo *friendTodel* e, se lo trova, lo rimuove dalla lista; successivamente invoca il metodo *remove* sulla *DiESeLHashTable* che, come visto in precedenza, rimuove il corrispondente elemento *DiESeLHashTableEntry*. Infine inserisce, tramite il comando *put*, la coppia formata dal nodo *node* e dalla lista aggiornata (cioè senza il nodo *friendTodel*).

```
public void delNode(IDiESeLNode node){

    if( node==null ){
        throw new IllegalArgumentException();
    }
    if(this.contains(node)){
        nodes.remove(node);
    }
}

public void delNeighbor(IDiESeLNode node, IDiESeLNode
    friendTodel){

    if( (node==null)||(friendTodel==null) ){
        throw new IllegalArgumentException();
    }
    if (this.contains(node)){
        ArrayList<IDiESeLNode> tmp = nodes.get(node);
        if (tmp.contains(friendTodel)){
            tmp.remove(friendTodel);
            nodes.remove(node);
            nodes.put(node,tmp);
        }
```

```
    }
}
```

Infine il metodo *size()* restituisce semplicemente la dimensione della tabella, delegando il calcolo al metodo *size* della classe *DiESeLHashTable*.

```
public int size(){
    return nodes.size();
}
```

Capitolo 6

Conclusioni

In questa tesi è stato descritto il lavoro svolto per consentire a DiESeL di gestire opportunamente una tabella hash che conservi le informazioni sulla rete. Per raggiungere tale scopo è stato necessario un lavoro approfondito di documentazione sulla tale struttura dati. Nel corso di tale studio si è visto come siano possibili numerosi approcci per la sua implementazione, tutti validi; si è poi valutato quale poteva essere la più funzionale in relazione al nostro utilizzo. Bisogna inoltre considerare il contesto del progetto PariPari: il lavoro svolto ha sempre tenuto in considerazione la necessità di produrre codice utilizzabile per un lavoro di squadra. In un progetto che coinvolge un team come quello di PariPari è stato essenziale, quando si effettua una modifica sul codice, consentire agli altri sviluppatori di continuare a utilizzare DiESeL senza costringerli ad effettuare a loro volta troppe modifiche all'interno del loro codice; i cambiamenti sono dovuti quindi riultare pressochè invisibili agli sviluppatori degli altri plugin di PariPari. Da qui la scelta ad esempio di lavorare sulle 3 classi di cui si è parlato nel capitolo 5: queste hanno mantenuto la stratificazione preesistente nell'organizzazione del codice riguardante la tabella, sostituendo la tabella hash di Java con una nuova implementazione. Si sarebbe potuto sicuramente implementare una tabella hash in maniera differente ma questo era il modo più adatto affinchè quella nuova potesse funzionare senza la necessità di modificare altri classi all'interno della libreria o, cosa anco-

ra peggiore, di richiedere ad altri membri del team di riadattare tutto il codice usato all'interno delle loro classi.

Sviluppi futuri

Dopo una fase di testing si è osservato che il codice si comporta come ci si aspettava: il confronto tra i nodi è gestito in maniera corretta e le operazioni di inserimento, ricerca e rimozione avvengono senza generare errori. Per lo sviluppo di questa tesi però, principalmente per difficoltà logistiche non si è potuto effettuare dei test su ampia scala, ma al massimo su una decina di nodi connessi alla rete. Il passo successivo sarebbe quello di poter effettuare in maniera agevole dei test su un numero elevato di macchine, andando poi ad approfondire le valutazioni sull'ottimizzazione del codice e sulle sue prestazioni in fase di esecuzione. Le scelte effettuate infatti hanno portato a un'implementazione che consentisse un discreto livello prestazionale per il tipo di utilizzo richiesto ma fosse allo stesso tempo relativamente semplice da implementare. Un analisi più approfondita (e con un maggior numero di risorse disponibilli per i test) potrebbe consentire, anche se a scapito della facilità di implementazione, una valutazione migliore di alcuni dettagli realizzativi, in modo da permettere un notevole guadagno prestazionale.

Bibliografia

[1] Stefano D'Incà Levis (2010), *PariPari - DiESeL: implementazione supporto multiserver e features*, Tesi Magistrale Università degli studi di Padova.

[2] Michael T. Goodrich, Roberto Tamassia (2005), *Data Structures and Algorithms in Java* 4th ed., John Wiley & Sons.

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein (2001), *Introduction to Algorithms* 2nd ed., Mcgraw-Hill.

[4] D. E. Knuth (1998), *Sorting and Searching*, vol. 3 of *The Art of Computer Programming* 2nd ed., Addison-Wesley.