Università di Pisa

Facoltà di Scienze Matematiche Fisiche e Naturali
Corso di Laurea Specialistica in Informatica

Tesi di laurea

Localizzazione di comunità per similarità su reti peer to peer basate su DHT

Fabio Baglini

Relatori
Dott.ssa Laura Ricci
Dott. Patrizio Dazzi

Controrelatore
Prof. Gualtiero Leoni

Anno Accademico 2009/2010

Indice

Capitolo 1

Introduzione

Una delle aree di ricerca e sviluppo più attive negli ultimi anni riguarda i sistemi basati sulla teoria delle reti sociali, pensati per permettere l'aggregazione degli utenti in comunità sulla base di interessi condivisi, allo scopo di favorire una diffusione più mirata dell'enorme mole di informazioni disponibile in internet.

Negli ultimi anni, a partire dalla prima proposta di un sistema di raccomandazione di artisti e album musicali basato sulla tecnica del "*collaborative filtering*" [2], sono stati realizzati molti strumenti di questo tipo, sia in ambito accademico che commerciale. Tali soluzioni hanno ottenuto un notevolissimo successo per la loro capacità di modellare il comportamento umano di scambiarsi informazioni per "passaparola", tipico tra persone che condividono uno o più interessi.

La maggior parte di questi sistemi è basata su architetture centralizzate, caratteristica che comporta una serie di problemi rilevanti, come una limitata scalabilità, la presenza di pochi, rilevanti punti di fallimento, che rende difficilmente ammortizzabili eventuali malfunzionamenti, e la concentrazione dei dati degli utenti presso un'unica "autorità".

Una delle alternative più promettenti per la soluzione di questi problemi è l'uso di architetture di tipo peer-to-peer (P2P) [1], che, distribuendo la gestione delle risorse e l'implementazione delle funzionalità richieste su un grande numero di entità equivalenti, supportano la realizzazione di sistemi

dinamici scalabili e resistenti ai fallimenti di insiemi di nodi di dimensioni anche rilevanti, evitando la concentrazione delle informazioni.

La realizzazione di un sistema basato su un approccio P2P per l'aggregazione degli utenti in gruppi individuti in base alla condivisione di interessi, detti *comunità*, è stata l'oggetto di due tesi di laurea, tra cui questa.

L'idea di partenza è quella di associare a ciascun utente un *profilo* che ne modelli gli interessi, quindi, basandosi sulle informazioni in esso contenute, mettere in contatto automaticamente e in modo trasparente gli utenti caratterizzati da profili sufficientemente simili, generando, per ognuna delle comunità individuate, un descrittore, confrontabile con quelli degli utenti, che sia rappresentativo di quelli di tutti i partecipanti.

Il sistema proposto è organizzato in due livelli , il primo dei quali è adibito alla creazione e all'aggiornamento delle comunità in modo self-emerging attraverso lo scambio continuo di informazioni tra gli utenti, mentre il secondo ha il compito di mantenere un indice aggiornato delle comunità esistenti, in modo da consentire ai nuovi utenti l'individuazione immediata delle comunità più adeguate ai loro interessi, e di notificare agli utenti esistenti delle stesse la creazione di nuove comunità rappresentative di interessi simili ai loro.

L'oggetto di questa tesi è la realizzazione delle funzionalità fornite dal secondo livello, a partire da un modello P2P strutturato di tipo *Distributed Hash Table* (DHT) [11], che fornisce le funzionalità delle tabelle hash distribuendo i dati sulla rete, offrendo un buon bilanciamento del carico, una gestione accurata della disconnessione dei nodi e una complessità logaritmica del routing sia per quanto riguarda il numero di hop necessari alla risoluzione di un'interrogazione che dal punto di vista delle dimensioni delle tabelle di instradamento.

Questo tipo di struttura dati non è però pensato per supportare il servizio fondamentale che l'indice delle comunità deve offrire, ossia la localizzazione dei profili per *similarità*. Come nel caso delle tabelle hash centralizzate, infatti, a ciascun valore è associato un identificatore calcolato tramite una funzione che distribuisce uniformemente i risultati, senza preservare la località dei dati originari. In questo modo è estremamente probabile che a valori

simili, ma non identici, ad esempio due stringhe composte da un milione di caratteri ciascuna che differiscono per uno solo di essi, vengano associati identificatori completamente diversi, che non possono in alcun modo essere messi in correlazione, rendendo impossibile la localizzazione di uno dei due dati a partire dall'altro.

Per realizzare le funzionalità dell'indice distribuito è pertanto necessario *estendere* la DHT con meccanismi che consentano ricerche non esatte dei contenuti.

La risoluzione di questo problema è l'argomento principale di questa tesi.

I modelli da impiegare per rappresentare gli interessi degli utenti devono essere applicabili a più domini e tipi di dato distinti ed essere facilmente costruibili e confrontabili con strumenti automatici.

I tre tipi di profilo proposti in questa tesi sono basati su insiemi di *attributi* di dimensioni arbitrarie confrontabili secondo semplici metriche di similarità, che possono essere costruiti in modo semplice a partire da oggetti di qualunque natura (testi memorizzati dagli utenti, log di navigazione sul web, metadati relativi a file multimediali, insiemi di coordinate geografiche relative a luoghi visitati, ecc.).

Il primo tipo di profilo è un *vettore di oggetti pesati*, definito associando a ciascun attributo un *peso*, che ne stima l'importanza, il secondo può essere costruito arricchendo un vettore di oggetti pesati con valori che stimino il grado di *correlazione* tra ciascuna coppia di attributi, definendo così una *matrice di adiacenza simmetrica*. Infine il terzo è costruito a partire da una matrice di adiacenza, suddividendola in più parti, ciascuna delle quali definita da un gruppo di attributi particolarmente correlati. Questa operazione viene effettuata applicando alla matrici un algoritmo esteso di estrazione delle componenti connesse, simile a quello proposto in [12]. Le relazioni più significative vengono evidenziate imponendo di una politica di potatura degli archi il cui peso non raggiunge una determinata soglia, mentre, eliminando le componenti costituite da un numero di oggetti inferiore ad un minimo prefissato si evita di considerare interessi irrilevanti o insufficientemente descritti.

Il profilo di ciascun utente è costituito dai vettori distinti di attributi pesati che definiscono le varie componenti della matrice. Ogni utente può quin-

di partecipare ad una comunità per ciascun gruppo di oggetti strettamente correlati individuato all'interno della sua matrice di adiacenza.

Ciò permette di ottenere un modello più realistico degli interessi degli utenti e di definire comunità maggiormente focalizzate su argomenti specifici, inoltre poiché ciascun oggetto può far parte al più di una componente della matrice di adiacenza, la dimensione dei profili è lineare nel numero degli oggetti pur sfruttando la disponibilità dei valori di correlazione tra gli attributi.

In questa tesi sono definite delle metriche per la valutazione della similarità tra vettori di oggetti pesati e tra matrici di adiacenza in base al numero di attributi condivisi tra i profili, ai loro pesi e ai loro valori di correlazione.

Nel caso di valori multi attributo, come i profili impiegati in questo sistema, il metodo impiegato tradizionalmente per implementare la ricerca per similarità prevede la generazione di un *identificatore* per ciascun attributo e la memorizzazione sulla DHT di una copia del valore per ciascuno di tali identificatori. A ciascun identificatore sulla DHT corrisponde quindi l'*insieme* dei valori che contengono l'attributo che viene mappato su tale identificatore. La ricerca per similarità viene quindi eseguita effettundo una query per ciascuna chiave e, una volta ottenuti tutti i risultati, intersecandoli, e calcolando la similarità di ciascun risultato con il valore iniziale secondo un'opportuna matrica e selezionando i risultati più significativi.

Questo meccanismo, pur essendo un metodo *esatto* per la risoluzione del problema, comporta, se applicato ad oggetti composti da un numero rilevante di attributi, elevati consumi di banda per le interrogazioni e di spazio per la memorizzazione degli oggetti sulla DHT.

In questa tesi è proposto un approccio *probabilistico* basato sull'impiego di *Locality Sensitive Hash Functions* (*funzioni LSH*) *[13]* per l'indicizzazione dei profili sulla DHT, che con un impiego di risorse drasticamente ridotto rispetto a quello imposto dal metodo di base, approssima la soluzione esatta del problema mantenendo molto elevata la qualità dei risultati.

Ciascuna famiglia di funzioni LSH è definita rispetto a una metrica di similarità tra insiemi s.

Come esposto in sezione 4.2.1, dati insiemi di oggetti confrontabili in

base ad s, impiegando un numero limitato di funzioni di questo tipo scelte casualmente è possibile associare ciascuno di essi un insieme di identificatori di *dimensioni limitate* (10-20 elementi), *costanti* e *indipendenti dalla sua cardinalità* tale che la probabilità di ottenere almeno un identificatore in comune per due insiemi *simili* secondo s sia estremamente elevata.

Il numero di identificatori associati per ciascun insieme di oggetti varia *solamente* in base alla probabilità desiderata di ottenere almeno un identificatore identico per oggetti simili e a quella di evitare *falsi positivi*.

Queste proprietà delle funzioni LSH vengono impiegate per la localizzazione sulla DHT di profili definiti a partire da *insiemi di attributi simili*.

Fissato un gruppo di funzioni LSH scelte casualmente, a ciascun profilo di comunità viene associato un insieme di n identificatori calcolati applicando tali funzioni all'*intero insieme* dei suoi attributi, quindi ne viene memorizzata sulla DHT una copia per ciascun identificatore.

Ciascuna entry della DHT contiene quindi tutti i profili che condividono l'identificatore ad essa associato.

Dato il profilo di un utente o di una comunità la ricerca dei profili simili memorizzati sull'indice avviene calcolando, tramite *lo stesso* gruppo di funzioni LSH, n identificatori, per ciascuno dei quali viene effettuata una query sulla DHT. Con elevata probabilità, per ciascuno dei profili di comunità il cui insieme di attributi è simile a quello del profilo dato, almeno una di queste query sarà relativa ad un identificatore condiviso tra i due profili.

Gli insiemi di profili ottenuti da ciasuna query vengono intersecati e filtrati per selezionare i profili delle comunità caratterizzati dalla più alta similarità con quello dato in base alle metriche opportunamente definite.

Le funzioni LSH utilizzate in questa tesi, descritte in sezione 4.2.1.2, sono definite a partire dalle *min-wise independent permutations* [28] e relative alla misura di similarità di Jaccard [40] definita, per ogni coppia di insiemi A e B, come $\frac{|A \cap B|}{|A \cup B|}$. Impiegando questo tipo di funzioni è quindi altamente probabile individuare insiemi di oggetti che condividono un numero di attributi significativo rispetto alle loro dimensioni. Questa nozione di similarità è affine a quelle espresse dalle metriche definite in questa tesi per i profili.

Come dimostrato in [28] un'implementazione esatta delle min-wise in-

dependent permutations avrebbe una complessità in spazio esponenziale a causa della necessità di selezionare in modo casuale con probabilità uniforme di una *permutazione* dello spazio dei possibili input tra tutte quelle possibili.

Per questo motivo nella pratica si usano delle implementazioni approssimate di queste permutazioni, in particolare in questa tesi è stato impiegato un algoritmo descritto in [14], che si è dimostrato valido sperimentalmente.

Nel caso di entrambi i metodi di indicizzazione proposti gli identificatori dei profili vengono generati senza considerare i pesi e i valori di correlazione associati agli oggetti. Questo è dovuto al fatto che è estremamente improbabile che due o più profili condividano *esattamente* le stesse coppie $\langle attributo, peso \rangle$, quindi considerare *simili* esclusivamente profili con questa caratteristica già in fase di indicizzazione sarebbe restrittivo al punto da rendere impossibile il funzionamento del sistema.

La ricerca dei profili simili ad uno dato avviene quindi in due fasi.

La prima fase consiste nell'individuare, tramite il meccanismo di indicizzazione della DHT, i gruppi di profili che *condividono* attributi o, usando l'indicizzazione tramite funzioni LSH, insiemi di attributi con quello dato.

Una volta individuati tali profili, la loro similarità con il profilo di riferimento viene *quantificata* in base alla metrica, tra quelle definite in questa tesi, adatta per il tipo di profili in questione.

Questo *filtraggio* dei profili avviene, in un primo momento, a livello locale presso ciascuno dei nodi della DHT interrogati, quindi, nuovamente, nel momento in cui il peer richiedente interseca e valuta i risultati ottenuti dalle interrogazioni.

Per consentire ai peer interrogati di selezionare i profili da inviare al richiedente in base alla loro similarità con il profilo di riferimento, una copia dello stesso viene inclusa in ciascuno dei messaggi di interrogazione.

Per permettere agli utenti di contattare i membri delle comunità individuate tramite il servizio di ricerca, e ai peer che implementano le funzionalità di mantenimento dell'indice di inviare notifiche di eventi ai membri delle opportune comunità, è necessario conoscere per ciascuna di esse l'indirizzo di uno o più membri, detti *peer di contatto*.

Poiché è possibile che una comunità continui ad esistere anche in caso di modifiche (presumibilmente limitate) al proprio profilo, ad ogni comunità è associato inoltre un *identificatore* univoco.

Poiché entrambi i metodi individuati per l'indicizzazione dei profili prevedono, sia pure in misura molto diversa, la replicazione dei dati per consentire la ricerca per similarità dei profili, è stato scelto di non memorizzare congiuntamente i descrittori delle comunità e le liste dei relativi peer di accesso, dal momento che tali liste non sono significative ai fini della valutazione della similarità dei profili e che la loro replicazione comporterebbe un maggior consumo di spazio e renderebbe più complicato il loro aggiornamento.

L'indice è stato quindi realizzato impiegando due DHT, la prima delle quali supporta la ricerca per similarità e memorizza i descrittori delle comunità, mentre la seconda associa a ciascun identificatore di comunità l'insieme degli indirizzi dei relativi peer di contatto.

Ciascun peer che contribuisce alla realizzazione dell'indice distribuito partecipa contemporaneamente ad entrambe le DHT, in modo da rendere il funzionamento del sistema trasparente agli utenti.

L'indice distribuito dei profili di comunità è stato implementato estendendo il servizio di DHT offerto dal framework Open Source per la costruzione di overlay network strutturati Overlay Weaver[3].

Il supporto alla ricerca dei profili per similarità è stato realizzato per entrambi i metodi proposti per l'indicizzazione dei profili.

L'impiego dell'implementazione approssimata delle min-wise indepentent permutations in questo sistema è stato validato associando gruppi di identificatori a insiemi di oggetti di similarità nota tramite le funzioni LSH definite a partire da tale approssimazione, confrontando le probabilità di generare identificatori coincidenti per oggetti simili ottenute con quelle ottenibili teoricamente da un'implementazione esatta e rilevando una sostanziale identità tra tali valori.

Le tecniche proposte sono state valutate sperimentalmente sulla base di un *dataset reale*, costituito da una lista di articoli di letteratura medica consultati da utenti di PubMed [4], dove ogni pubblicazione è individuata da titolo, *keywords* e *abstract*. Da questo dataset sono stati estratti i profili

relativi a oltre 2700 utenti.

Per valutarne la qualità i risultati ottenuti effettuando ricerche per similarità sulla DHT impiegando l'indicizzazione dei profili tramite funzioni LSH sono stati confrontati con quelli ottenuti, per le stesse interrogazioni, indicizzando i singoli attributi dei profili.

Questa valutazione è stata effettuata, utilizzando varie configurazioni dell'indicizzazione con funzioni LSH, sia per i *vettori di oggetti pesati* che per le *matrici di adiacenza.*

I profili appartenenti a un sottoinsieme selezionato casualmente di quelli a disposizione sono stati memorizzati sulla DHT, mentre i restanti sono stati utilizzati come argomenti per le ricerche. Inizialmente queste operazioni sono state effettuate impiegando l'indicizzazione dei singoli attributi e i risultati ottenuti dalle ricerche per similarità sono stati presi come riferimento in quanto ottenuti tramite un metodo *esatto.*

Successivamente la memorizzazione dei profili e le ricerche sono state ripetute, impiegando gli stessi dati, al variare della configurazione dell'indicizzazione con funzioni LSH.

Il confronto tra i risultati ottenuti con i due metodi di indicizzazione ha dimostrato che, utilizzando il metodo basato su funzioni LSH, è possibile risolvere le interrogazioni in modo assolutamente soddisfacente anche in una situazione in cui i dati risultano particolarmente sparsi.

Per i vari tipi di profilo sono stati stimati, al variare del metodo di indicizzazione, il consumo di banda per la costruzione dell'indice distribuito, per l'invio dei messaggi di interrogazione e delle risposte alle query da parte dei nodi interrogati e lo spazio occupato per la memorizzazione dell'indice distribuito, notando una riduzione dei costi di un ordine di grandezza già per profili di medie dimensioni nel caso in cui i profili vengono indicizzati con funzioni LSH.

Infine è stato dimostrato che, anche indicizzando i dati con funzioni LSH, è possibile mantenere il bilanciamento del carico sui nodi della DHT a livelli ottimali.

La tesi è strutturata come segue. Nel capitolo 2 vengono illustrati alcuni sistemi esistenti per la ricerca di oggetti in sistemi peer to peer e per la mo-

dellazione degli interessi degli utenti. Nel capitolo 3 è descritta l'architettura generale del sistema. Nel capitolo 4 sono illustrati i modelli individuati per la definizione dei profili, il metodo tradizionale di risoluzione delle interrogazioni per similarità e la proposta basata sull'impiego di funzioni LSH, con un confronto dei costi teorici indotti dall'applicazione di entrambi ai tipi di profilo descritti in precedenza. Nel capitolo 5 viene descritta la sperimentazione condotta, a partire da un insieme di dati reali, su un'implementazione di entrambi i metodi di indicizzazione basata sul framework Overlay Weaver [3].

Capitolo 2

Stato dell'arte

2.1 HAPS

L'architettura HAPS (Hybrid Architecture for P2P Search) viene proposta in [15] come risposta ai problemi derivanti dalla frequente connessione e disconessione di nodi (*node churn*) e dal cambiamento dei contenuti (*document change*) nell'ambito della ricerca di testi in sistemi peer to peer.

Questi inconvenienti si ripercuotono particolarmente sulle prestazioni dei sistemi basati su overlay network strutturati, come le *distributed hash table* [11], a causa dell'elevato numero di messaggi necessario a mantenere la coerenza dell'indice distribuito e della stessa DHT, provocando notevoli problemi di scalabilità. In questi casi, i rilevanti vantaggi derivanti dall'uso di questo tipo di strutture, come la disponibilità di un meccanismo di ricerca dei dati poco costoso e accurato anche per reti di grandi dimensioni e una gestione della disconnessione dei nodi che assicura nella maggior parte dei casi il mantenimento della disponibilità dei documenti, possono passare in secondo piano.

I sistemi non strutturati come, ad esempio, Gnutella, sono meno sensibili a tali problemi a causa dell'assenza di strutture dati distribuite, ma questa è anche la causa del loro limite principale, ovvero la necessità di risolvere le query tramite *flooding.* Impiegando questo meccanismo ciascun peer, nel caso in cui non sia in grado di risolvere localmente un'interrogazione, la inoltra

indiscriminatamente ai suoi immediati vicini i quali, se saranno in grado di fornire i dati richiesti, invieranno una risposta, altrimenti provvederanno a inoltrare nuovamente la query ai loro vicini, fino al raggiungimento di tutti i nodi della rete o di un numero prestabilito di inoltri. Se effettuato ad ampio raggio, questo procedimento può comportare un consumo di banda eccessivamente alto, mentre in caso contrario c'è il rischio di non riuscire ad individuare molti dei risultati significativi.

HAPS è un sistema gerarchico che rappresenta una soluzione ibrida tra questi due approcci: i peer, detti *provider*, vengono suddivisi in cluster individuati per similarità dei documenti che condividono, quindi i nodi appartenenti a ciascun cluster si organizzano in un overlay network a stella il cui centro è un superpeer scelto in base all'affidabilità stimata e alla capacità di gestione del traffico, detto *hub*.

Gli hub sono interconnessi da una DHT che non indicizza singoli documenti, ma i cluster individuati al passo precedente.

Il fatto che la DHT sia costituita da un gruppo ristretto di peer particolarmente affidabili permette di limitare il verificarsi del node churn nella componente del sistema che ne risulterebbe più danneggiata, mentre l'indicizzando cluster di documenti simili non è necessario aggiornare l'indice distribuito mantenuto dagli hub ogni volta che un documento viene aggiunto, eliminato o modificato; inoltre la dimensione dell'indice stesso è inferiore rispetto al caso in cui vengono indicizzati i singoli file.

2.1.1 Costruzione della directory di ricerca a due livelli

Ogni provider produce una struttura dati, chiamata *provider description*, costituita da una lista di tuple del tipo $< term_i, ProviderTF, ProviderDF >$ che indicano, per ciascun termine contenuto nei documenti che condivide, il numero di documenti in cui occorre ($ProviderDF$) e il numero totale di occorrenze ($ProviderTF$).

Questi dati vengono inviati all'hub del cluster di appartenenza insieme al numero di documenti, termini unici e parole presenti nella collezione di documenti del provider.

Unendo le provider description ricevute ciascun hub costruisce la propria directory di ricerca locale come un array del tipo

$$< term_i, < ProviderID, ProviderTF, ProviderDF > -list >$$

quindi, aggregando le stesse provider description crea una descrizione dello stesso tipo per se stesso (*hub description*), ciascuna tupla della quale è detta *post*, e calcola i valori relativi a numero di documenti, parole e termini unici presenti nel cluster.

I post vengono quindi memorizzati sulla DHT usando come chiave il termine corrispondente in modo da costruire la directory di ricerca globale, ogni entry della quale sarà del tipo $< term_i, < HubID, HubTF, HubDF > -list >$. I dati relativi al numero di termini, parole e documenti di ciascun hub vengono associate a una *stopword* (ossia un termine frequente ma non significativo come ad esempio una congiunzione o un articolo) definita a priori (ad esempio "a") e memorizzate in un'unica entry della DHT detta *overall statistics entry* (*OSE*).

2.1.2 Ricerca

Un peer che sottopone una query al sistema la invia, per prima cosa, all'hub del cluster a cui appartiene, che provvede a ricercare sulla DHT le statistiche globali associate a ciascun termine della query e l'OSE.

Basandosi su questi dati l'hub richiedente seleziona i cluster di documenti più rilevanti rispetto alla query, che inoltra quindi agli hub corrispondenti.

Questi a loro volta, usando il loro indice locale, scelgono alcuni provider all'interno della loro sottorete ai quali sottoporre l'interrogazione perché procedano alla ricerca locale dei documenti.

Ogni provider invia quindi i propri k risultati più significativi al proprio hub, che, una volta ottenuta una risposta da tutti i peer contattati, provvede ad effettuare il *ranking* di tutti i documenti ricevuti, selezionare a sua volta i k che tra tutti soddisfano meglio la query ed inviarli all'hub richiedente.

Infine quest'ultimo, una volta ricevuti i risultati della ricerca dai vari cluster, effettua un'ulteriore cernita e restituisce i documenti globalmente più significativi al peer che aveva effettuato l'interrogazione.

2.1.3 Sperimentazione

HAPS è stato confrontato con Minerva [16], un sistema strutturato basato su Chord [8]e con Pepper [17], un sistema a due livelli in cui gli hub sono connessi da un overlay network non strutturato, simulando sia il verificarsi del churn sia la variazione dei documenti.

In assenza di churn la precisione dei risultati ottenuti da HAPS si avvicina a quella di Minerva quando viene interrogato almeno il 4% dei provider, ad un costo per query leggermente più alto, mentre quando si verifica questo fenomeno è HAPS a fornire i risultati più accurati, con un vantaggio che aumenta all'aggravarsi del problema.

In ciascun caso con Pepper si ottengono risultati peggiori che con gli altri sistemi.

Il document change si ripercuote in maniera analoga sulla qualità dei risultati, mentre per quanto riguarda il costo di aggiornamento degli indici, quello imposto da Minerva è superiore di quasi un ordine di grandezza rispetto agli altri due.

2.2 P-Ring

P-Ring [19] è un sistema per la risoluzione di *range query* in ambiente P2P strutturato, studiato con l'obiettivo di garantire un buon bilanciamento del carico in maniera non probabilistica e risolvere le query in tempo $O(\lg_d n)$ con $d > 2$.

Il sistema è strutturato in 5 moduli:

1. *Fault tolerant ring:* overlay network ad anello, usato come struttura di base per garantire l'affidabilità. In questa implementazione è stato usato un anello Chord [8].

2. *Data store*: si occupa dell'allocazione dei dati sui peer in modo da renderla il più possibile uniforme. È una delle proposte originali di P-Ring.

3. *Content router*: effettua il routing delle query. È una delle proposte originali di P-Ring.

4. *Replication manager:* gestisce la replicazione dei dati. In questa implementazione è stato usato quello proposto per CFS. [20]

5. *P2PIndex:* indice distribuito, sfrutta il data store per gli aggiornamenti e il content router per la risoluzione delle query.

2.2.1 Il livello *data store*

In P-Ring l'assegnamento dei dati ai peer non viene effettuato in base al risultato di una funzione hash applicata alle chiavi di ricerca, ma direttamente a seconda del loro valore, per non perderne l'ordinamento.

Inizalmente lo spazio delle chiavi di ricerca viene suddiviso in un numero di range minore di quello dei peer, in questo modo ad alcuni di essi non verranno assegnati oggetti. In ciascun istante i peer sono quindi divisi in due insiemi, quello dei peer che memorizzano dati (*owner peer*s) e quello dei peer "liberi" (*helper peers*). Della rete ad anello fanno parte solamente gli owner peer.

Questa suddivisione viene sfruttata per bilanciare come segue il carico in modo da ovviare al probabile sbilanciamento della distrubuzione dei dati dovuta al fatto di non usare una funzione hash per la distribuzione degli oggetti sui nodi della rete.

Definita *storage factor* (sf) la quantità $\lceil \frac{N}{P} \rceil$ dove N è il numero stimato di oggetti da gestire e P il numero stimato di peer, l'obiettivo dell'algoritmo di bilanciamento del carico è di mantenere il numero di oggetti assegnato a ciascun peer all'interno dell'intervallo $[l, u]$, con $l=sf$ e $u=2l$.

A tal fine ciascun owner peer, nel caso in cui dovesse trovarsi a seguito dell'inserimento di nuovi dati sulla rete, ad essere responsabile di più di u

oggetti (situazione di *overflow*), provvede a contattare un helper peer a cui inviare la metà dei suoi dati caratterizzata dai valori più alti delle chiavi di ricerca.

L'helper peer (diventato adesso un *owner*) provvederà quindi ad inserirsi nel *fault tolerant ring* come suo successore richiedendo l'aggiornamento delle strutture dati utilizzate per il routing.

Nel caso in cui invece un ad owner peer rimanesse la responsabilità di un numero di oggetti minore di l (situazione di *underflow*), esso contatta il suo successore sull'anello inviandogli una richiesta di ridistribuzione dei dati o fusione dei nodi, comunicandogli il numero di oggetti che gestisce.

Il nodo successore sceglie quale delle due azioni intraprendere in base al suo carico attuale, in particolare, se la somma del numero dei suoi oggetti e di quelli del peer che è andato in *underflow* non oltrepassa la soglia u, procede alla fusione, altrimenti invia al predecessore una parte dei suoi dati, a partire dall'estremo inferiore dell'intervallo di sua competenza.

In caso di fusione il peer che ha subito l'*underflow* dopo aver ceduto i suoi dati al successore, che estende il range di sua competenza, diventa un *helper peer* ed esce dall'anello.

Tutto ciò permette di limitare superiormente a $2 + \varepsilon$ per ogni $\varepsilon > 0$ il fattore di sbilanciamento del carico, definito come

$$\frac{max_{p \in O}|p.own|}{min_{p \in O}|p.own|}$$

dove O è l'insieme degli owner peer e $p.own$ indica l'insieme degli oggetti assegnati al peer p.

L'uso degli *helper peer* risulta più conveniente delle tradizionali tecniche di bilanciamento del carico che prevedono di trasferire dati da nodi sovraccarichi ad altri che fanno già parte della rete perché non è richiesto lo spostamento lungo l'anello di questi ultimi, con conseguente ridistribuzione di parte dell'indice.

La forte asimmetria nell'assegnamento delle funzioni tra i peer, che potrebbe sembrare contraddittoria, viene in larga parte compensata dal fatto che gli helper peer sono comunque una piccola parte del totale e che, a cau-

sa degli inserimenti e delle cancellazioni di dati, si verifica con sufficiente frequenza la rotazione tra owner e helper peer.

Il meccanismo degli helper peer può essere esteso per assicurare un migliore bilanciamento del carico (in questo caso anche per quanto riguarda le query) facendo in modo che gli helper peer non rimangano inattivi in attesa del verificarsi di un overflow o underflow, ma partecipino attivamente alla gestione dei dati.

A ciascun owner peer p possono essere assegnati uno o più helper peer q_j con i quali spartire i dati.

L'intervallo $p.range$=$(lb,\ ub]$ dello spazio delle chiavi che individua i dati assegnati ad un owner p con k helper a disposizione viene suddiviso nei sottointervalli $(lb = b_0, b_1], (b_1, b_2], ..., (b_k, ub]$ di ampiezza in generale diversa, creati in modo che ciascuno contenga lo stesso numero di oggetti.

I primi k intervalli, con i relativi oggetti, passeranno sotto la responsabilità degli helper peer q_j, mentre l'ultimo rimarrà di competenza esclusiva di p.

Il sottointervallo di cui è responsabile il peer q (che sia helper o owner) è indicato con $q.resp$.

Le query per chiavi che rientrano nel range di un helper peer verranno inoltrate da p verso quest'ultimo, mentre p mantiene il controllo sulle operazioni di ridistribuzione dei dati e si occupa di gestire gli inserimenti e le cancellazioni di dati in tutto l'intervallo $p.range$.

Gli algoritmi per la gestione di overflow e underflow vengono modificati: in caso di overflow il peer interessato, se ha a disposizione degli helper, invia parte dei suoi dati ad uno di essi, cedendogli anche alcuni dei suoi helper residui, se invece si deve gestire un underflow, effettuando la fusione di due nodi gli helper peer del nodo che lascia l'anello vengono riassegnati ad altri owner scelti casualmente.

È previsto anche un algoritmo, detto *usurp*, per la ridistribuzione degli helper peer che prevede che un owner p, una volta individuato l'owner meno carico presente sulla rete, possa sottrargli un helper peer q a condizione che $|p.resp| \geq 2 * \sqrt{1+\delta} * |q.resp|$.

È dimostrato che con questa tecnica si ottiene un fattore di sbilanciamento, ridefinito come

$$\frac{max_{p \in P}|p.resp|}{min_{p \in P}|p.resp|}$$

limitato superiormente da $2+\varepsilon$ per ogni $\varepsilon > 0$, mantenendo costante il costo ammortizzato delle singole operazioni.

2.2.2 Il livello *content router*

Il content router ideato dagli autori, denominato *Hierarchical Ring* (*HR*) definisce un modello di routing gerarchico a partire dalla tradizionale rete ad anello.

Preso come parametro un intero d tale che $1 \leq i \leq d$ detto *ordine* dell'HR, ogni peer mantiene ed utilizza come indice per il routing delle query l'array bidimensionale di riferimenti a nodi *node[level][position]* così definito:

- $1 \leq level \leq numLevels 1 \leq position \leq d$
- $p.node[1[1] = succ(p)$
- $p.node[1[j+1] = succ(p.node[i][j]), 1 \leq j < d$
- $p.node[l+1[1] = p.node[l][d]$
- $p.node[l+1[j+1] = p.node[l][j].node[l+1][1], 1 \leq l < numLevels, 1 \leq j < d$
- $p.node[numLevels.lastPeer.node[numLevels][1] \in [p, p.node[numLevels].lastPeer)$

La procedura di costruzione dell'array inizia al livello 1, viene iterata incrementando il livello ogni d riferimenti memorizzati e termina una volta verificata l'ultima condizione, ossia all'ultimo passo prima di iniziare un "nuovo giro" lungo l'anello.

Per costruzione *numLevels* è pari al più a $\lceil \log_d P \rceil$, lo spazio necessario a ciascun peer per mantenere la struttura dati *node* è quindi $O(d \cdot \lceil \log_d P \rceil)$.

Anziché i valori delle chiavi di ricerca l'indice riguarda le posizioni dei peer sull'anello (a ciascun livello l di $p.node$ sono memorizzati riferimenti a

peer che si trovano a una distanza compresa tra d^{l-1} e d^l), cosa che rende il comportamento dell'algoritmo di routing indipendente dalla distribuzione dei dati.

Un peer p che riceve una query di range $(lb, ub]$ alla quale non può rispondere, scorre a ritroso $p.node$ a partire dall'ultima posizione del livello più alto fino a trovare il successore più distante il cui range di competenza non oltrepassa quello della query e inoltra la richiesta a quest'ultimo.

Una volta raggiunto il primo livello della tabella di routing la query viene inoltrata lungo l'anello ai vari successori diretti finché non viene raggiunto quello responsabile del valore *ub*.

Se ciascuna tabella di routing è completa e consistente l'insieme di tutte le tabelle copre l'intero spazio di indirizzamento ed è garantito che, ogni volta che un peer inoltra una query ad un suo successore che si trova al livello l della sua tabella, quest'ultimo, a meno che non sia già stato raggiunto il primo livello, la invierà a sua volta ad un successore di livello al più *l-1*.

Ciò comporta un costo in termini di numero di hop che è $O(\lceil \log_d P \rceil)$ per il raggiungimento del nodo che memorizza il valore più vicino a *lb*, con un costo totale per la risoluzione della query che è $O(\lceil \log_d P \rceil) + m$ dove m è il numero di peer che memorizzano dati compresi nel range della query.

La consistenza delle tabelle di routing è però compromessa dall'inserimento o dalla dipartita di nodi a seguito di fallimenti o di situazioni di overflow/underflow al livello data store, quindi è necessario provvedere periodicamente all'esecuzione di un'opportuna procedura di stabilizzazione che ciascun peer esegue periodicamente indipendentemente dagli altri, iterandola per tutti i livelli della tabella.

È dimostrato che, in assenza di fallimenti di nodi sull'anello e se ciascun peer può determinare esattamente il suo successore immediato, la procedura di stabilizzazione rende le tabelle di routing complete e consistenti in tempo $O((d-1)\lceil \log_d P \rceil) \cdot su$ dove *su* (*stabilization unit*) è il tempo necessario a rendere consistente un livello della tabella per tutti i peer.

Ciò permette di limitare superiormente il costo di una range query in presenza di inserimenti e cancellazioni di nodi sulla rete, se questi avvengono a un ritmo pari a r/su e inizialmente le tabelle di routing sono consistenti, a

$\lceil \log_d P \rceil + 2(d-1)\lceil \log_d P \rceil + m$ hop.

2.3 MAAN

MAAN (Multi-Attribute Addressable Network) [21] è un'estensione di Chord [8] pensata per supportare la risoluzione di *range query multi attributo* su DHT relativamente alla *localizzazione di risorse computazionali* nell'ambito del Grid computing [22].

Ciascuna *risorsa* in MAAN è definita da un insieme di coppie $\langle attributo, valore \rangle$ che ne descrivono le caratteristiche, ad esempio il tipo e il numero di processori, la quantità di memoria RAM libera, il livello di carico attuale del processore, ecc.

MAAN è pensato per risolvere interrogazioni del tipo: "Trova le risorse caratterizzate da $NumeroCPU \geq 4 \wedge 2GB \leq RAMLibera \leq 6GB \wedge Carico \leq 30\%$".

A questo scopo, mentre gli attributi di tipo testuale sono mappati sullo spazio degli identificatori della DHT impiegando le tradizionali funzioni hash, ai valori degli attributi di tipo numerico sono applicate delle *funzioni hash che preservano la località.*

Una funzione hash H gode di tale proprietà se verifica le condizioni:

- $H(a) < H(b) \Leftrightarrow a < b$
- Se un intervallo di valori $[a, c]$ è suddiviso nei sottointervalli $[a, b]$ e $[b, c]$, allora l'intervallo $[H(a), H(c)]$ è suddiviso in $[H(a), H(b)]$ e $[H(b), H(c)]$.

Utilizzando questo tipo di funzioni, per ciascun intervallo di valori $[a, b]$, gli attributi i cui valori ricadono in tale intervallo verranno mappati sui nodi i cui identificatori cadono nel sottospazio $[H(a), H(b)]$.

Quando una risorsa viene registrata in MAAN, per ciascuna delle coppie $\langle attributo, valore \rangle$ che la descrivono vengono memorizzati sulla DHT i dati che la identificano sulla griglia (tipicamente indirizzo IP e porta) e una copia dell'intero insieme di coppie che la descrive.

La risoluzione di una range query riguardante n attributi avviene eseguendo una range query unidimensionale per ciascuno degli attributi specificati e intersecando i risultati ottenuti per selezionare quelli che soddisfano tutte le condizioni.

Questo procedimento può essere effettuato eseguendo le n sotto-query separatamente oppure, come proposto in MAAN, facendo uso di un'ottimizzazione , che consente di risolvere l'interrogazione n-dimensionale effettuando una sola sotto-query sulla DHT.

Tale ottimizzazione è basata sui concetti di *selettività* di un attributo relativamente a una range query e di *attributo dominante* di una interrogazione.

Data una range query in cui si richiede che il valore dell'attributo a sia compreso tra v_1 e v_2, la *selettività* di a rispetto a tale query è il rapporto tra la dimensione del sottospazio degli identificatori $[H(v_1), H(v_2)]$ e quella dell'intero spazio degli identificatori. L'*attributo dominante* di una range query è quello che presenta la selettività più alta.

L'ottimizzazione del procedimento di risoluzione delle range query multi attributo proposta da MAAN consiste nell'effettuare unicamente la sotto-query relativa all'attributo dominante dell'interrogazione, interrogando quindi solamente i nodi che memorizzano risorse che soddisfano i requisiti richiesti per tale attributo e sfruttando il fatto che tali nodi memorizzano le copie integrali delle descrizioni di tali risorse per selezionare localmente quelle che soddisfano l'intero insieme delle condizioni poste nell'interrogazione.

2.4 Web Retrieval P2P con chiavi altamente discriminatorie

Questa proposta [18] nasce per risolvere il problema della quantità di traffico generato dalla ricerca di testi in collezioni di grandi dimensioni per mezzo di parole chiave, in sistemi peer to peer strutturati basati su DHT.

Tradizionalmente l'indicizzazione dei testi viene effettuata assegnando a ciascun peer, per mezzo di una funzione hash, un sottoinsieme dei termini

presenti nella collezione e costruendo un indice distribuito composto da una entry per ciascun termine, in cui è memorizzata la lista dei documenti in cui il termine occorre e degli indirizzi dei peer che li condividono, detta *posting list*, nella quale, a ciascun documento, è associato un punteggio di *ranking* che ne stima la rilevanza.

In questo caso ogni query Q viene risolta effettuando una ricerca separata per ciascun termine $q_i \in Q$, quindi effettuando l'intersezione delle posting list ottenute in risposta e scegliendo i documenti da presentare all'utente in base al numero di termini q_i contenuti e al valore di ranking.

Questo modello di indicizzazione, per quanto teoricamente efficace dal punto di vista dei risultati, soffre di notevoli problemi di scalabilità legati alla quantità di traffico generato per la risoluzione delle query a causa dell'invio, in risposta a ciascuna sotto-query, di posting list molto lunghe.

Gli autori propongono, in alternativa, di indicizzare i testi tramite delle *chiavi altamente discriminatorie* (HDK) che individuino in modo molto preciso degli insiemi relativamente piccoli di documenti.

2.4.1 Chiavi discriminatorie, non discriminatorie e altamente discriminatorie

Con *chiave*, in generale, viene inteso un qualunque insieme di termini appartenenti alla collezione di documenti, le chiavi generabili a partire da una collezione contenente $|T|$ termini distinti sarebbe quindi $2^{|T|}$, quantità che, per valori realistici di $|T|$ sarebbe sicuramente ingestibile.

L'insieme delle chiavi associabili ai documenti viene quindi ristretto a quelle che rispettano opportuni vincoli di dimensione massima, fissando un valore s_{max} prossimo alla dimensione media delle query, e vicinanza dei termini all'interno dei documenti.

Tra queste vengono distinte come *chiavi discriminatorie* (DK) quelle che occorrono in al più un numero fissato DF_{max} di documenti.

Sia le chiavi discriminatorie che quelle *non discriminatorie* (NDK) godono specifiche proprietà di *sussunzione*, in particolare, qualunque sovrainsie-

me di una chiave discriminatoria è a sua volta una DK, mentre qualunque sottoinsieme di una chiave non discriminatoria è a sua volta una NDK.

Queste proprietà permettono di imporre sulle chiavi un ulteriore vincolo di ridondanza, che definisce le *chiavi altamente discriminatorie* come le DK i cui sottoinsiemi sono tutti NDK.

2.4.2 Costruzione dell'indice

L'indice invertito è costituito dall'associazione, alle chiavi altamente discriminatorie e non discriminatorie calcolate sull'intera collezione di documenti, delle relative posting list che, per costruzione, sono relative ad al più DF_{max} documenti per le HDK, mentre vengono ristrette ai DF_{max} testi con valore di ranking più alto nel caso delle NDK.

La costruzione dell'indice avviene in modo distribuito, seguendo un procedimento incrementale in s_{max} fasi.

Al generico passo s, $1 \leq s \leq s_{max}$, ciascun peer calcola le HDK e le NDK di dimensione s relative alla propria collezione di documenti locale, quindi le memorizza, insieme alle relative posting list, sulla DHT. A livello globale, a questo passo fa seguito l'aggiornamento delle posting list troncate relative alle NDK e il controllo che ciascuna HDK calcolata localmente mantenga la sua proprietà di occorrere in al più DF_{max} testi anche relativamente all'intera collezione.

Se una chiave considerata altamente discriminatoria a livello locale non gode effettivamente questa proprietà, ai peer che l'avevano proposta viene inviata una notifica in modo che, al passo $s+1$, possano tentare di espanderla con le NDK di dimensione 1 per generare una nuova HDK candidata di $s+1$ termini.

Poichè ad ogni passo s i peer generano le loro HDK locali basandosi unicamente sulle NDK di dimensione 1 e $s-1$, la frequenza di tali chiavi nell'intera collezione di documenti è l'unica conoscenza globale richiesta.

Le chiavi calcolate per la collezione e le informazioni sulla loro frequenza sono memorizzate a livello globale.

2.4.3 Risoluzione delle interrogazioni

Al momento in cui viene sottoposta una query vengono individuate al suo interno le eventuali chiavi (HDK e NDK) corrispondenti a quelle calcolate per i documenti della collezione, quindi vengono cercate sulla DHT le posting list corrispondenti, che vengono quindi intersecate per ricavare i documenti globalmente più significativi.

Tali posting list, consistendo al massimo di DF_{max} record ciascuna, sono notevolmente più compatte di quelle utilizzate quando l'indice è costruito a partire da termini singoli, in questo modo il traffico dovuto alle ricerche viene ampiamente ridotto.

2.4.4 Risultati teorici

È stato dimostrato che la dimensione dell'indice cresce linearmente con quella della collezione indicizzata, mentre il traffico necessario alla risoluzione di ciascuna query q è limitato superiormente da $n_k \cdot DF_{max}$ dove $n_k = 2^{|q|} - 1$ se $|q| \leq s_{max}$ oppure $\sum_{i=1}^{s_{max}} \binom{|q|}{i}$ altrimenti.

Poiché per il tipo di applicazione previsto, ovvero la risoluzione di query web basate su parole chiave, il valore di s_{max} è stimato tra 2 e 3 e quello massimo di $|q|$ nell'ordine della decina, l'operazione risulta scalabile.

2.4.5 Risultati sperimentali

Per quanto riguarda la dimensione dell'indice e il traffico necessario alla sua costruzione sono stati notati degli incrementi significativi (fino a 14 volte per la dimensione e fino a 40,7 volte per il traffico) rispetto a quanto necessario per la costruzione di un indice tradizionale per la stessa collezione di documenti.

Tali incrementi sono stati giudicati sopportabili, tenendo conto del fatto che la disponibilità di spazio per la memorizzazione non dovrebbe essere un problema in questo tipo di rete, che l'indicizzazione viene eseguita di rado e che questi valori tendono a stabilizzarsi al crescere della dimensione della collezione.

Il traffico generato per la risoluzione delle query ha invece subito, rispetto al caso base una riduzione drastica, al punto che il traffico totale stimato, ipotizzando un carico di lavoro di circa 1,5 milioni di query al mese (dimensione reale dei query log utilizzati) e una frequenza mensile delle indicizzazioni è inferiore di 20 volte rispetto a quello generato con un indice su termini singoli nel caso della collezione utilizzata nell'esperimento (653,546 documenti) e di 42 volte per una collezione di 1 milione di documenti.

Il confornto con i risultati ottenuti per le stesse query dal motore di ricerca centralizzato Terrier ha mostrato una sovrapposizione dei risultati che si è mantenuta superiore al 90% al crescere del numero di documenti nella collezione e per due valori di DF_{max}.

2.5 Efficient semantic search on DHT overlays

In [24] gli autori affrontano il problema della ricerca di testi su DHT basata sulla similarità allo scopo di gestire efficientemente query del tipo "*trovare tutti i documenti simili a D*", dove D, anziché una breve lista di parole chiave è a sua volta un documento, possibilmente in linguaggio naturale.

Il metodo individuato consiste nell'impiegare tecniche di information retrieval come il *Vector Space Model* (VSM) per estrarre da ciascun documento un vettore di termini distinti e pesati che lo rappresenti (tipicamente tra i 150 e i 300), dopodiché impiegare per l'indicizzazione e la ricerca degli stessi un modello probabilistico basato su funzioni hash sensibili alla località [13] (LSH) che consente di interrogare un numero di nodi della DHT (10-20) piccolo rispetto alla dimensione media dei *term vector* riuscendo comunque a risolvere in modo soddisfacente le interrogazioni.

L'estrazione di un vettore di termini da un documento secondo il VSM consiste nell'individuare innanzitutto le radici dei termini presenti (ad esempio, alle parole *waited, waits* e *waiting* corrisponderà la sola radice *wait*) con un procedimento detto *stemming*, quindi nell'eliminare dalla lista ottenuta le *stopword,* ovvero i termini, come ad esempio gli articoli e le congiunzioni, molto frequenti, ma poco significativi.

A ciascuno dei termini rimasti viene quindi assegnato un peso che ne stima la rilevanza rispetto al documento in questione e, se sono disponibili infomazioni globali, rispetto all'intera collezione.

L'architettura del sistema prevede che, tra l'utente e la DHT siano interposti un modulo che si occupa di estrarre i *term vector* (TV) dai documenti sottoposti ed uno che, utilizzando più funzioni LSH, assegna a ciascun TV un numero relativamente contenuto (10-20) di *identificatori semantici* (*semID*) utilizzati per l'indicizzazione del documento e del vettore dei termini sulla DHT.

L'algoritmo di definizione dei *semID* è definito in modo che a documenti simili secondo la metrica di Jaccard corrisponda, con elevata probabilità, almeno un *semID* uguale.

Ciascuna entry dell'indice distribuito è quindi costituita da una tripla $< semID, < fileID, TV > -list >$, dove *fileID* è l'identificatore, calcolato tramite una funzione hash tradizionale, del documento al quale si accede, quindi, con un livello di indirezione.

Nel momento in cui un utente sottopone un'interrogazione per similarità il documento sottoposto viene anch'esso sottoposto all'estrazione del TV corrispondente e alla successiva assegnazione dei *semID* utilizzando le stesse funzioni LSH impiegate per l'indicizzazione, quindi verrà inviata una richiesta ai nodi della DHT responsabili di ciascun identificatore semantico, i quali effettueranno in locale un confronto tra i vettori di termini corrispondenti al *semID* in questione con quello dato, utilizzando un'opportuna funzione di similarità (Jaccard, coseni etc.) e restituiranno i *fileID* corrispondenti ai TV sufficientemente simili. Una volta ottenuti i risultati il nodo che aveva inviato la query provvederà quindi ad effettuare un'ulteriore passo di integrazione, cernita ed eventuale caching degli stessi.

A livello locale ciascun peer che riceve un'interrogazione confronta il *term vector* contenuto nella query con tutti quelli associati all'identificatore richiesto usando la metrica $REL(D,Q) = \sum_{t \in D,Q} d_t \cdot q_t$, dove Q è il TV contenuto nella query, D il documento con cui confrontarlo, d_t e q_t i pesi assegnati a ciascun termine.

Una possibile ottimizzazione per la ricerca locale è costituita dal clustering

dei term vector, in modo da poter confrontare il vettore in ingresso con un numero ridotto di oggetti.

Un'altra ottimizzazione, più importante, è quella derivante dalla possibilità di individuare all'interno di ciascun *term vector* il sottoinsieme dei termini di peso più alto, detti *top terms,* ed utilizzare solamente questo per le operazioni di indicizzazione e ricerca.

In questo modo è possibile ridurre la probabilità di generare lo stesso *semID* per documenti che condividono molti termini poco rilevanti aumentando al tempo stesso quella di mettere in relazione documenti che hanno in comune un numero non molto alto di termini molto significativi e diminuendo il numero di documenti potenzialmente superflui ottenuti in risposta a una query.

I risultati sperimentali dimostrano che già utilizzando 50 *top terms* si hanno prestazioni migliori senza un decadimento apprezzabile dei risultati.

2.6 SUGGEST

SUGGEST [12] è uno strumento di raccomandazione di pagine web basato su tecniche di *Web Usage Mining* pensato per consigliare all'utente di un sito web dei link a pagine non ancora visitate e reputate interessanti in base alle informazioni ricavate dalla sua sessione di navigazione e ad una stima della correlazione tra le pagine che compongono il sito, lavorando completamente online.

Il sito web viene modellato con un grafo non diretto, del quale le pagine rappresentano i nodi, mentre i pesi assciati agli archi ne stimano il grado di correlazione secondo la formula $W_{ij} = \frac{N_{ij}}{max(N_i, N_j)}$, dove N_{ij} è il numero di sessioni in cui sono state richieste sia la pagina i che la pagina j, mentre N_i e N_j indicano il numero di sessioni in cui compaiono unicamente l'una o l'altra pagina.

I valori N_{ij} definiscono la matrice di adiacenza M utilizzata per memorizzare il grafo.

Per ciascuna sessione utente, identificata tramite un cookie, viene memorizzata la lista delle ultime pagine visitate (*PageWindow*).

I link da suggerire all'utente vengono scelti all'interno di gruppi di pagine fortemente correlate, detti *cluster*, costituiti da sottoinsiemi delle componenti connesse del grafo, che vengono individuati utilizzando un algoritmo incrementale derivato da quello descritto in [25], modificato con l'introduzione di una strategia di potatura degli archi.

A ciascuna pagina è associato un punteggio, detto *UsageRank*, che ne stima la popolarità in modo simile, in linea di principio, al *PageRank* [26], utilizzando i valori M_{ij} e basandosi sul fatto che una pagina è tanto più popolare quanto più è correlata ad altre pagine popolari.

Al valore di *UsageRank* della i-esima pagina contribuiscono quindi tutte quelle presenti nell'insieme $B(p_i) = \{j \mid M_{ji} > 0\}$, ciascuna in proporzione alla propria popolarità e a quanto la sua correlazione con la pagina i è significativa rispetto a quella con le altre pagine.

Per ogni pagina i si ha quindi $UR(p_i) = \sum_{j \in B(p_i)} \alpha(j \to i) \cdot UR(p_j)$, dove $\alpha(j \to i) = \frac{M_{ji}}{\sum_{k=0, k \neq j}^{k=(|M|-1)} M_{jk}}$.

Quando un utente richiede la pagina j il sistema ne identifica la sessione e la pagina i di provenienza e, se questa coppia di pagine non occorre nella *PageWindow*, aggiorna i corrispondenti valori M_{ij} e M_{jj}, quindi aggiorna i cluster in base alle modifice introdotte nella matrice.

A partire dal nodo j viene effettuata una visita DFS del grafo, percorrendo però solamente gli archi con un peso W_{ij} superiore a una determinata soglia *Minfreq* e fondendo tutti i cluster ai quali appartengono i nodi attraversati.

I nodi appartenenti allo stesso cluster di j che non vengono visitati sono quelli il cui valore di correlazione con gli altri elementi del gruppo è sceso al di sotto di *Minfreq* in seguito all'aggiornamento della matrice e che devono quindi esserene esclusi.

Per ciascuno di essi viene quindi ripetuta la visita del grafo, assegnando i nodi attraversati ad un cluster creato ex novo.

In questo modo vengono generati oggetti di dimensioni gestibili anche nel caso di grafi molto ampi eliminando contemporaneamente le relazioni meno

significative.

Nel caso pessimo, in cui la matrice è piena e non viene effettuata la potatura degli archi, la complessità dell'algoritmo è lineare nel numero di archi del grafo.

I cluster aggiornati vengono quindi intersecati con la *PageWindow* dell'utente e le pagine non ancora visitate da suggerire vengono scelte, in base al loro valore di *UsageRank,* all'interno del cluster per cui l'intersezione è più ampia.

2.7 Considerazioni

L'esempio portato da HAPS dimostra che l'uso di una struttura gerarchica, basata sulla clusterizzazione di peer che mettono in condivisione oggetti simili, per la gestione di collezioni di dati altamente dinamiche e di rilevanti dimensioni può rivelarsi notevolmente vantaggiosa rispetto alle architetture peer to peer tradizionali, rendendo promettente l'adozione di questo approccio per la realizzazione del sistema.

Poiché, come verrà esposto in sezione 4.1, i profili degli utenti del sistema sono definiti a partire da insiemi di generici oggetti, senza che vengano imposte restrizioni sui loro tipi,e sulla loro semantica, la similarità dei profili non può essere valutata direttemente in base al *valore* degli oggetti stessi. Anche nel caso particolare in cui venissero impiegati unicamente attributi appartenenti a insiemi ordinabili e l'insieme dei possibili attributi fosse noto e di dimensioni limitate, la definizione di *precisi intervalli di valori* per i quali due profili vengono considerati simili sarebbe, oltre che difficilmente automatizzabile, poco significativa, a causa dal numero elevato (nell'ordine delle centinaia) di attributi previsto per i profili, e, comunque, troppo restrittiva per il contesto in cui si pone questa tesi.

Per questi motivi, la ricerca per similarità non può essere effettuata, in questo contesto, adottando un sistema per la risoluzione di *range query multi attributo* come, ad esempio, MAAN [21].

Dal momento che la dimensione prevista per i profili è dell'ordine delle centinaia di elementi, anche una strategia di indicizzazione basata su chiavi come in [18], per quanto interessante per il campo di applicazione previsto

dagli autori, non è applicabile alla ricerca per similarità di oggetti di questo tipo, poiché prevede la generazione di un numero di chiavi di ricerca esponenziale rispetto alla dimensione media delle query.

L'indicizzazione e la ricerca per similarità dei profili tramite l'impiego di funzioni LSH sono invece possibili, dal momento che non è necessario definire la similarità per mezzo di intervalli di valori e che il numero di chiavi associate a ciascun profilo è ridotto (10-30), costante e indipendente dal numero di attributi che lo compongono.

SUGGEST, infine, presenta un interessante spunto per la modellazione degli interessi degli utenti, come esposto in sezione 4.1.

Capitolo 3

Il sistema: architettura generale

3.1 Introduzione

Il sistema proposto in questa tesi ha come compito il raggruppamento automatico degli utenti in comunità in base ai loro interessi comuni, in modo da facilitare una diffusione maggiormente mirata delle informazioni disponibili in internet.

A questo scopo è necessario associare ad ogni utente un *profilo* che modelli i suoi interessi e che, confrontato con quelli degli altri partecipanti al sistema, possa dare un'indicazione del loro grado di similarità.

Analogamente ciascuna comunità dovrà essere caratterizzata da un profilo che descriva gli interessi comuni ai suoi membri e che possa essere confrontato con quello di ciascun utente per verificare se questi possa o meno essere interessato a far parte della stessa.

Poiché tanto gli interessi degli utenti quanto la loro partecipazione al sistema sono soggetti ad un'elevata variabilità, le comunità devono necessariamente essere pensate come entità dinamiche la cui composizione e descrizione, oltre alla stessa esistenza, devono essere frequentemente riconsiderate.

Il sistema è organizzato in due livelli, il primo dei quali è adibito alla creazione delle comunità in modo self-emerging e all'aggiornamento dei loro descrittori attraverso lo scambio continuo di informazioni tra gli utenti, mentre il secondo ha il compito di mantenere un indice aggiornato dei descrittori

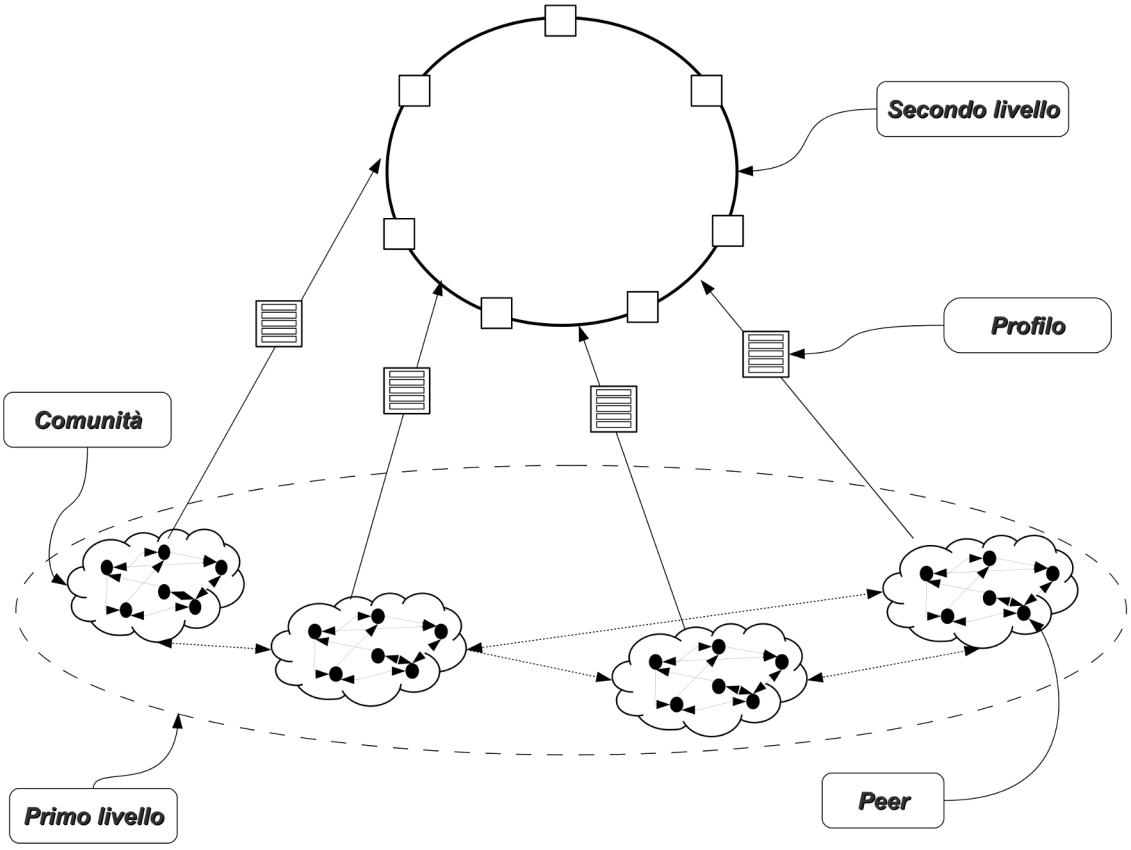


Figura 3.1: Struttura a livelli del sistema

delle comunità esistenti, in modo da consentire ai nuovi utenti l'individuazione immediata delle comunità più adeguate ai loro interessi, e di notificare ai membri delle stesse la creazione di nuove comunità rappresentative di interessi simili ai loro.

Un esempio della struttura a due livelli è mostrato in figura 3.1.

L'intero sistema è concepito per essere realizzato in modo *completamente decentralizzato* secondo un approccio peer-to-peer (P2P), in modo da evitare problemi legati alla scalabilità e alla presenza di punti centralizzati di fallimento e di concentrazione dei dati personali degli utenti.

Nelle sezioni successive di questo capitolo verranno illustrate le caratteristiche generali dei due livelli.

3.2 Il primo livello

Poiché gli interessi degli utenti e la loro stessa partecipazione al sistema di comunità sono soggetti ad un'elevata variabilità, per la realizzazione delle funzionalità del primo livello è stato scelto di basarsi su un modello P2P non strutturato.

Le esigenze di base sono la creazione di un overlay network la cui struttura rifletta la suddivisione dei peer in comunità e la definizione di un algoritmo totalmente decentralizzato per la creazione del profilo di ciascuna comunità.

Le soluzioni individuate devono, oltre a produrre risultati apprezzabili, dimostrarsi resistenti alla variabilità dei dati e al fatto che nelle reti non strutturate, la disconnessione dei nodi avviene frequentemente ed è difficilmente gestibile (fenomeno del *node churn*), inoltre devono essere adattabili a vari tipi di profilo e richiedere un basso costo sia in termini computazionali che, soprattutto, di uso di banda.

Questi problemi sono stati affrontati in [35], individuando una soluzione basata su protocolli di tipo Gossip [9] (nel caso specifico Cyclon e Vicinity [10]) per l'individuazione dei gruppi di utenti simili e su un protocollo di *leader election* chiamato *LEADER* (Leader: Elezione Asincrona, Dinamica, Efficiente, Rapida), realizzato tramite un meccanismo di votazione distribuita in due fasi che sfrutta le informazioni ottenute tramite i protocolli sottostanti.

La prima parte del protocollo *LEADER* consiste nella scelta, all'interno di ciascun gruppo di vicini individuati dai protocolli Cyclon e Vicinity, di uno o più *candidati leader*, ovvero peer i cui profili risultano rappresentativi degli interessi un numero sufficientemente elevato di utenti. A tale scopo ciascun peer ha a disposizione un numero limitato di *voti* da assegnare ai vicini caratterizzati dai profili maggiormente simili al suo, i peer che ricevono in questo modo un numero di voti superiore a una soglia prefissata vengono considerati come candidati leader.

Nella seconda fase ogni peer sceglie, tra i candidati leader a lui noti, quello a lui più simile come leader potenziale, assegnandogli un secondo voto. Tra i candidati leader viene quindi effettuata una seconda selezione in base al numero di voti ricevuti, che dà come risultato l'insieme dei leader effettivi

delle comunità. Ciascun peer si aggrega quindi alla comunità rappresentata dal candidato leader da lui votato, se questo è risultato essere un leader effettivo, oppure a quella del leader effettivo a lui più simile.

I profili di ciascuna comunità coincidono con quelli dei relativi leader.

Un esempio del procedimento di elezione dei leader è illustrato in figura 3.2.

Il basso consumo di banda e di risorse computazionali indotto dall'uso di protocolli di tipo Gossip permette un frequente scambio di informazioni tra tutti i nodi della rete, consentendo alla struttura delle comunità di adattarsi efficacemente ai cambiamenti tramite la reiterazione della procedura di elezione dei leader.

3.3 Il secondo livello

La realizzazione delle funzionalità di indicizzazione delle comunità è l'oggetto di questa tesi.

A questo livello i dati, costituiti dai profili delle comunità individuate dall'algoritmo di primo livello, sono soggetti ad una variabilità relativamente contenuta, perciò è possibile, coinvolgendo un insieme di peer selezionati e sufficientemente stabili, impiegare un modello P2P strutturato, come una *Distributed Hash Table* (DHT) [11], sfruttandone vantaggi quali la disponibilità di algoritmi di routing affidabili e di complessità logaritmica rispetto al numero dei nodi e la gestione affidabile delle disconnessioni dei peer.

Per permettere agli utenti di contattare i membri delle comunità individuate tramite il servizio di ricerca, e ai peer che implementano le funzionalità di mantenimento dell'indice di inviare notifiche di eventi ai membri delle opportune comunità, è necessario conoscere per ciascuna di esse l'indirizzo di uno o più membri, detti *peer di contatto*, che si suppongono designati tramite opportune funzionalità implementate al primo livello del sistema.

Poiché è possibile che una comunità continui ad esistere anche in caso di modifiche (presumibilmente limitate) al proprio profilo, ad ogni comunità è

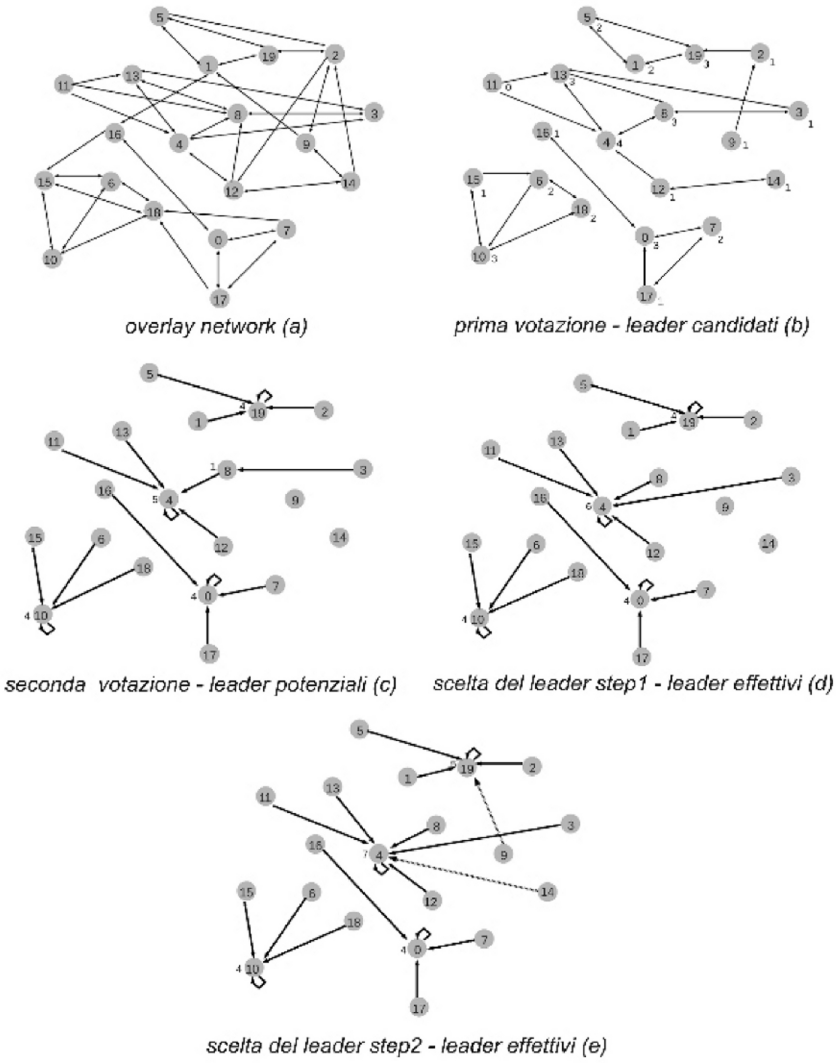


Figura 3.2: Protocollo LEADER: Elezione dei leader.

associato un *identificatore* univoco che, insieme al profilo, ne costituisce il *descrittore*, che si assume generato dai servizi di primo livello.

La realizzazione dei servizi di questo livello è basata sulla possibilità di localizzare, su una DHT, i descrittori delle comunità effettuando, a partire sia dai profili degli utenti che da quelli delle comunità, delle ricerche *per similarità* anziché le ricerche esatte *per chiave* per le quali le DHT sono progettate, mantenendo contenuti i costi relativi al consumo di banda e di spazio di memorizzazione.

Questo aspetto non banale, che rappresenta il cardine di questa tesi, verrà approfondito nel capitolo 4, così come quello riguardante la definizione dei profili di utenti e comunità.

Come viene esposto e motivato nel capitolo 4, entrambi i metodi individuati per implementare la ricerca per similarità sulla DHT comportano, anche se in misura diversa, l'associazione di più identificatori a ciascun profilo, con la conseguente replicazione dei dati sia nel caso della memorizzazione dei dati sull'indice che in quello della loro ricerca. Per questo motivo è stato scelto di non memorizzare congiuntamente i descrittori delle comunità e le liste dei relativi peer di accesso, dal momento che tali liste non sono significative ai fini della valutazione della similarità dei profili e che la loro replicazione comporterebbe un maggior consumo di spazio e renderebbe più complicato il loro aggiornamento.

L'indice è stato quindi realizzato impiegando due DHT, la prima delle quali, detta *ProfilesDHT,* supporta la ricerca per similarità e memorizza i descrittori delle comunità, mentre la seconda, detta *AddressesDHT*, associa a ciascun identificatore di comunità l'insieme degli indirizzi dei relativi peer di contatto.

Ciascun peer che contribuisce alla realizzazione dei servizi di questo livello partecipa contemporaneamente ad entrambe le DHT, in modo da rendere il funzionamento del sistema trasparente agli utenti.

Nelle sezioni seguenti verranno descritte le funzionalità messe a disposizione e le operazioni svolte al momento della loro invocazione.

3.3.1 Accesso al sistema e localizzazione della comunità iniziale

Al momento dell'ingresso nel sistema, ciascun utente invia ad uno dei peer che mantengono l'indice distribuito il proprio profilo. Il peer interrogato effettua sulla DHT che memorizza i descrittori delle comunità (ProfilesDHT) la ricerca di profili simili a quello dell'utente, scegliendo quindi, tra i risultati ricevuti, il descrittore della comunità con il profilo più simile a quello dato. A partire dall'identificatore di tale comunità, il nodo interrogato ricerca quindi la corrispondente lista degli indirizzi dei peer di accesso sull'apposita DHT (AddressesDHT), restituendola quindi all'utente assieme al descrittore della comunità selezionata.

L'utente potrà quindi aggregarsi a tale comunità se dovesse ritenerne il profilo sufficientemente simile al proprio, utilizzando in ogni caso gli indirizzi dei peer ricevuti come insieme iniziale di vicini per l'esecuzione degli algoritmi del primo livello.

Algoritmo 3.1 Ingresso nel sistema

Input: il profilo P dell'utente

Output: il descrittore della comunità raccomandata e gli indirizzi dei relativi peer di accesso.

1: $similarCommunities \leftarrow$ ProfilesDHT.GetSimilarCommunities(P); // Ricerca i profili di comunità più simili a P.
2: Sort($similarCommunities, P$); // Ordina i risultati in base alla similarità con P.
3: $selectedCommunity \leftarrow$ first($similarCommunities$); // Sceglie il descrittore della comunità con profilo più simile a P.
4: $addresses \leftarrow$ AddressesDHT.Get($selectedCommunity.ID$); // Ricerca gli indirizzi dei peer di accesso per la comunità selezionata.
5: $retval \leftarrow \langle selectedCommunity, addresses \rangle$;
6: **return** $retval$;

3.3.2 Creazione di una nuova comunità

Al momento della creazione di una nuova comunità, uno dei membri ne invia il descrittore e la lista degli indirizzi dei peer di accesso ad uno dei peer che mantengono l'indice, che provvede a memorizzarli sulle apposite DHT.

La presenza della nuova comunità deve quindi essere notificata ai membri dei gruppi di interesse più simili perché possano confrontarne il profilo con il proprio e decidere se aggregarvisi o meno.

A tale scopo, il peer memorizza il descrittore della nuova comunità su ProfilesDHT ricevendo in risposta quelli delle comunità affini, seleziona tra i risultati gli n più significativi, ricerca su AddressesDHT gli indirizzi dei rispettivi peer di accesso ed invia loro il nuovo descrittore.

Algoritmo 3.2 Creazione di una nuova comunità

Input: il descrittore D della nuova comunità, la lista L degli indirizzi dei peer di accesso, il numero n di comunità a cui inviare la notifica.

1: $similarCommunities \leftarrow$ ProfilesDHT.StoreDescriptor(D); // Memorizza D e riceve i descrittori delle comunità simili.
2: AddressesDHT.Put($D.ID$,L); // Associa L all'identificatore della nuova comunità sull'oppurtuna DHT.
3: Sort($similarCommunities, D$); // Ordina i descrittori ottenuti in base alla similarità con D.
4: $selectedCommunities \leftarrow$ sublist($similarCommunities, 0, n$); // Seleziona le n comunità più simili a D
5: **for all** $cS \in selectedCommunities$ **do**
6: $addresses \leftarrow$ AddressesDHT.Get($cS.ID$); // Ricerca gli indirizzi dei peer di accesso per le comunità selezionata.
7: **for all** $addr \in addresses$ **do**
8: NotifiyNewCommunity($addr, D$); // Invia il nuovo descrittore ai peer di accesso della comunità
9: **end for**
10: **end for**

3.3.3 Variazione del profilo di una comunità

In questo caso, come per la creazione di una nuova comunità, l'evento deve essere notificato agli utenti appartenenti ai gruppi di interesse più affini.

Il peer incaricato provvede a rimuovere il descrittore precedente dall'apposita DHT, quindi memorizza il nuovo descrittore ricevendo in risposta quelli delle comunità più simili. Scelti gli n risultati più significativi, il peer ricerca su AddressesDHT gli indirizzi dei peer di accesso delle rispettive comunità ed esegue la notifica dell'esistenza del nuovo profili.

Algoritmo 3.3 Variazione del profilo di una comunità

Input: il vecchio descrittore della comunità oD, il nuovo descrittore della comunità nD, il numero n di comunità a cui inviare la notifica.

```
1: ProfilesDHT.Remove(oD);
2: similarCommunities ← ProfilesDHT.StoreDescriptor(nD);
   // Memorizza nD e riceve i descrittori delle comunità simili.
3: Sort(similarCommunities, nD); // Ordina i descrittori ottenuti in base
   alla similarità con nD.
4: selectedCommunities ← sublist(similarCommunities, 0, n); // Seleziona le n comunità più simili a nD
5: for all cS ∈ selectedCommunities do
6:    addresses ← AddressesDHT.Get(cS.ID); // Ricerca gli indirizzi dei
      peer di accesso per le comunità selezionata.
7:    for all addr ∈ addresses do
8:       NotifiyNewDescriptor(addr, nD); // Invia il nuovo descrittore ai
         peer di accesso della comunità
9:    end for
10: end for
```

.

3.3.4 Scioglimento di una comunità

Lo scioglimento di una comunità comporta solamente l'eliminazione, da entrambe le DHT, di tutti i dati per essa memorizzati.

Algoritmo 3.4 Scioglimento di una comunità

Input: il descrittore della comunità da eliminare D

1: ProfilesDHT.Remove(D); // Rimuove D dalla DHT dei profili.
2: AddressesDHT.Remove($D.ID$); // Rimuove la lista degli indirizzi dei peer di accesso.

Capitolo 4

Definizione e ricerca per similarità dei profili

In questo capitolo vengono esposti e motivati i modelli individuati per la definizione dei profili degli utenti e delle comunità e le soluzioni proposte per permetterne la localizzazione per similarità a partire da strutture di memorizzazione, le DHT, concepite per supportare unicamente ricerche esatte per chiave.

In particolare, dopo aver illustrato il modello di base per la definizione dei profili, ne verranno descritti tre derivati, progressivamente arricchiti.

Successivamente verranno descritti due approcci, uno di base ed uno basato sull'uso di *funzioni LSH (Locality Sensitive Hash)* [13] per l'indicizzazione e la localizzazione per similarità su DHT di tali tipi di profilo.

Per ambedue i metodi verranno illustrati gli algoritmi per le operazioni di ricerca di un profilo, memorizzazione e notifica alle comunità simili di un nuovo descrittore di comunità e variazione e notifica alle comunità simili del profilo di una comunità per ciascun tipo di profilo, corredati da una stima dei costi in termini di consumo di banda e di spazio di memorizzazione.

4.1 Modellazione degli interessi degli utenti

Il modello scelto per la definizione dei profili degli utenti e delle comunità deve, oltre a mettere in evidenza efficacemente gli interessi degli utenti, rispondere alla necessità, tipica degli ambienti totalmente decentralizzati, di contenere le dimensioni dell'indice distribuito e la quantità di traffico sull'overlay network necessaria al funzionamento efficiente del sistema.

Non sono inoltre da trascurare fattori come la possibilità di applicare il modello a più domini e tipi di dato distinti, la facilità di modificarne le istanze in base alle variazioni degli interessi degli utenti ed il costo computazionale di creazione, modifica e confronto dei profili. Per queste ragioni non sono stati presi in considerazione modelli di di categorizzazione e confronto tipici degli studi sul web semantico e sull'elaborazione dei linguaggi naturali come, ad esempio, WordNet [37] o ontologie come OWL [36], in favore di modelli più semplici, basati su insiemi di *oggetti* di dimensione arbitraria confrontabili secondo semplici metriche di similarità.

Un modello di questo tipo può essere costruito a partire da dati di qualunque tipo (testi memorizzati dagli utenti, log di navigazione sul web, metadati relativi a file multimediali, insiemi di coordinate geografiche relative a luoghi visitati ecc.), inoltre può essere facilmente arricchito nel caso in cui si possano associare agli oggetti dei *pesi*, che ne stimino l'importanza, definendo un *vettore di oggetti pesati*.

Un esempio di profilo di questo tipo, definito a partire da termini, è mostrato in figura 4.1.

canto	Verdi	Rigoletto	chitarra
1,4	0,9	0,62	0,8

Figura 4.1: Esempio di vettore di oggetti pesati.

Avendo a disposizione anche informazioni relative alla correlazione tra gli oggetti (ad esempio il numero o la frequenza delle loro co-occorrenze nell'insieme di dati da cui viene estratto il profilo), è possibile definire i profili come *matrici di adiacenza simmetriche*, sfruttando queste informazioni per arricchire il modello degli interessi degli utenti, a discapito della compattezza

della rappresentazione, dal momento che la dimensione di un profilo basato su n oggetti passa da $O(n)$ a $O(\frac{n^2}{2})$.

Due esempi di matrice di adiacenza sono riportati in figura 4.2 e in figura 4.3, dove si nota che sulla diagonale principale viene memorizzato il *peso* di ciascun oggetto. in figura 4.3con W_i è indicato il *peso* dell'i-esimo oggetto, con $Rel_{i,j}$ il valore che quantifica la correlazione tra l'i-esimo e il j-esimo oggetto.

	canto	Verdi	Rigoletto	chitarra
canto	1,4	0,6	0,7	0,4
Verdi	0,6	0,9	0,5	0
Rigoletto	0,7	0,5	0,62	0
chitarra	0,4	0	0	0,8

Figura 4.2: Un esempio di matrice di adiacenza.

	obj_1	obj_2	obj_3	obj_4	$\cdots\cdots$	obj_{n-1}	obj_n
obj_1	W_1	$Rel_{1,2}$	$Rel_{1,3}$	$Rel_{1,4}$	$\cdots\cdots$	$Rel_{1,n-1}$	$Rel_{1,n}$
obj_2	$Rel_{1,2}$	W_2	$Rel_{2,3}$	$Rel_{2,4}$	$\cdots\cdots$	$Rel_{2,n-1}$	$Rel_{2,n}$
obj_3	$Rel_{1,3}$	$Rel_{2,3}$	W_3	$Rel_{3,4}$	$\cdots\cdots$	$Rel_{1,2}$	$Rel_{3,n}$
obj_4	$Rel_{1,4}$	$Rel_{2,4}$	$Rel_{3,4}$	W_4	$\cdots\cdots$	$Rel_{4,n-1}$	$Rel_{4,n}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\cdots\cdots$	$\vdots$	$\vdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\cdots\cdots$	$\vdots$	$\vdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\cdots\cdots$	$\vdots$	$\vdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\cdots\cdots$	$\vdots$	$\vdots$
obj_{n-1}	$Rel_{1,n-1}$	$Rel_{2,n-1}$	$Rel_{3,n-1}$	$Rel_{4,n-1}$	$\cdots\cdots$	W_{n-1}	$Rel_{n-1,n}$
obj_n	$Rel_{1,n}$	$Rel_{2,n}$	$Rel_{3,n}$	$Rel_{4,n}$	$\cdots\cdots$	$Rel_{n-1,n}$	W_n

Figura 4.3: Esempio di matrice di adiacenza

Questo tipo di profilo è ulteriormente raffinabile utilizzando il contributo informativo dato dai valori di correlazione tra gli attributi della matrice di adiacenza per individuare le singole aree di interesse dell'utente, suddividendo il profilo originario in più parti, ciascuna delle quali definita da un gruppo di attributi particolarmente correlati.

Questa operazione può essere effettuata applicando alle matrici di adiacenza un algoritmo esteso di estrazione delle componenti connesse, come ad

esempio avviene in [12]. Le relazioni più significative vengono evidenziate imponendo di una politica di potatura degli archi il cui peso non raggiunge una determinata soglia, mentre, eliminando le componenti costituite da un numero di oggetti inferiore ad un minimo prefissato si evita di considerare interessi irrilevanti o insufficientemente descritti.

In questo caso, il profilo di ciascun utente non sarà più costituito da un singolo gruppo di oggetti pesati o dall'intera matrice di correlazione, ma dai gruppi distinti di attributi pesati che definiscono le varie componenti connesse della matrice. Ogni utente potrà quindi partecipare ad una comunità per ciascun gruppo di oggetti strettamente correlati individuato all'interno della sua matrice di adiacenza.

Ciò permette di ottenere un modello più realistico degli interessi degli utenti e di definire comunità maggiormente focalizzate su argomenti specifici, inoltre poiché ciascun oggetto può far parte al più di una componente connessa della matrice di adiacenza, la dimensione dei profili torna ad essere lineare nel numero degli oggetti pur sfruttando la disponibilità valori di correlazione tra gli attributi.

Un esempio di estrazione delle componenti connesse da una matrice di adiacenza con potatura degli archi è illustrato in figura 4.4, in tal caso il profilo risultante è composto, supponendo di scartare le componenti connesse composte da meno di due oggetti, dai vettori di oggetti pesati in figura 4.5. L'utente potrebbe quindi partecipare a due comunità.

	0	1	2	3	4	5	6	7
0	–	0	0	0	0	2/3	0	0
1	0	–	1	0	0	0	1	0
2	0	1	–	0	0	0	1	0
3	0	0	0	–	0	0	0	0
4	0	0	0	0	–	0	0	1/2
5	2/3	0	0	0	0	–	0	2/3
6	0	1	1	0	0	0	–	0
7	0	0	0	0	1/2	2/3	0	–

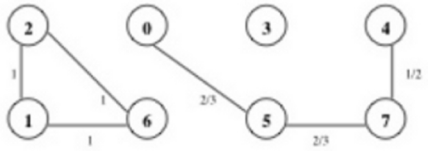


Figura 4.4: Estrazione delle componenti connesse con potatura degli archi di peso inferiore a 1/3

1	2	6
W_1	W_2	W_6

0	4	5	7
W_0	W_4	W_5	W_7

Figura 4.5: Profilo derivato dall'estrazione delle componenti connesse

4.1.1 Metriche di similarità

Una semplica metrica di similarità tra insiemi di natura arbitraria è quella di Jaccard [40], definita, per due generici insiemi A e B, come

$$\frac{|A \cap B|}{|A \cup B|}$$

utile per misurare la similarità di due insiemi relativamente a contenuto e dimensioni.

La metrica di similarità impiegata per confrontare due profili costituiti da vettori n-dimensionali di oggetti pesati, caso che comprende anche gli interessi modellati come componenti connesse estratte da una matrice di adiacenza, prende in considerazione i pesi degli oggetti e le dimensioni dei due profili.

Dati due profili P_1 e P_2, sia *obj* un oggetto contenuto in entrambi. Il contributo dato da *obj* alla similarità tra P_1e P_2 è pari a $min\,[W_1(obj), W_2(obj)]$, dove $W_1(obj)$ e $W_2(obj)$ sono i *pesi* associati all'oggetto *obj* nei due profili e *min* indica il minimo dei due valori, scelto per evitare le sovrastime che potrebbero derivare dall'impiego del massimo o della media dei due valori.

La metrica di similarità è quindi definita come

$$SIM_V(P_1, P_2) = \frac{\sum_{obj \in P_1 \cap P_2} min\,[W_1(obj), W_2(obj)]}{max(|P_1|, |P_2|)} \tag{4.1}$$

dove $|P_1|$ e $|P_2|$ sono le dimensioni dei due profili e *max* il massimo dei due valori. Questa pesatura è introdotta per evitare sovrastime nel caso in cui uno dei profili sia incluso totalmente o in larga parte nell'altro.

In figura 4.6 è mostrato un esempio di calcolo della similarità tra due vettori di termini pesati in base alla metrica 4.1.

P_1

canto	Verdi	Rigoletto	chitarra
1,4	0,9	0,62	0,8

P_2

Verdi	violino	Rigoletto
0,81	1,3	0,7

$$SIM_V(P_1, P_2) = \frac{0,81 + 0,62}{4} = 0,3575$$

Figura 4.6: Calcolo della similarità tra due vettori di oggetti pesati.

Per quanto riguarda le matrici di adiacenza, ciascun attributo comune tra due profili contribuisce al loro valore di similarità con il proprio peso e con la media dei suoi valori di correlazione con gli altri attributi comuni.

Anche in questo caso, per evitare sovrastime, vengono scelti i pesi e i valori di correlazione più bassi e il valore ottenuto viene pesato rispetto alla dimensione del profilo più grande.

La metrica di similarità è quindi definita come:

$$SIM_M(P_1, P_2) = \frac{\sum_{obj \in P_1 \cap P_2} \left\{ RW + RW \times \frac{\sum_{obj' \in P_1 \cap P_2} min(Rel_1(obj,obj'), Rel_2(obj,obj'))}{|\{obj' \in P_1 \cap P_2 | \exists Rel(obj,obj')\}|} \right\}}{max(|P_1|, |P_2|)} \tag{4.2}$$

dove $RW = min\left[W_1(obj), W_2(obj)\right]$, $Rel_1(obj, obj')$ e $Rel_2(obj, obj')$ sono i valori di correlazione tra gli oggetti obj e obj' nei due profili e $|P_1|$ e $|P_2|$ il numero di attributi degli stessi.

La figura mostra un esempio del calcolo della similarità tra due matrici di adiacenza secondo la metrica 4.2.

P_1

	canto	Verdi	Rigoletto	chitarra
canto	1,4	0,6	0,7	0,4
Verdi	0,6	0,9	0,5	0
Rigoletto	0,7	0,5	0,62	0
chitarra	0,4	0	0	0,8

P_2

	Verdi	violino	Rigoletto
Verdi	0,81	0,75	0,6
violino	0,75	1,3	0,5
Rigoletto	0,6	0,5	0,7

$$SIM_M(P_1, P_2) = \frac{(0,81 + 0,81 \cdot 0,5) + (0,62 + 0,62 \cdot 0,5)}{4} = 0,53625$$

Figura 4.7: Calcolo della similarità tra due matrici di adiacenza.

4.2 Ricerca per similarità e memorizzazione dei profili

Le implementazioni di base delle DHT, pur offrendo valide caratteristiche a livello di affidabilità ed efficienza del routing, sono strutture poco adatte alla ricerca dei contenuti per similarità.

Il loro meccanismo di memorizzazione, infatti, si basa, come per le tabelle hash centralizzate, sulla mappatura dei valori su un insieme di *chiavi* tramite l'applicazione di una funzione hash.

La caratteristica principale delle funzioni hash è quella di generare chiavi con una distribuzione il più possibile uniforme, non rispecchiando in alcun modo la località dei dati di partenza.

Le strutture dati basate sull'uso di funzioni hash risultano quindi adatte alla ricerca di valori unidimensionali esatti.

Per consentire la ricerca per similarità di valori multidimensionali è quindi necessario estendere il funzionamento di base delle DHT.

Una volta definita una metrica di similarità per il tipo di dato in questione, un metodo banale per realizzare la ricerca per similarità è quello di calcolare una chiave per ciascuno degli attributi che lo compongono, associando una

copia del valore ad ogni chiave, memorizzando quindi sulla DHT le singole coppie $\langle chiave, valore \rangle$.

A ciascuna chiave sulla DHT corrisponderà quindi, anziché un unico valore, l'insieme di tutti i dati che contengono un attributo il cui valore viene mappato sulla chiave.

La ricerca per similarità viene quindi eseguita effettundo una query per ciascuna chiave e, una volta ottenuti tutti i risultati, intersecandoli, e calcolando la similarità di ciascun risultato con il valore iniziale.

In generale, ciascuno dei nodi della DHT interrogati potrà inviare più di un valore in risposta alla query. Per ridurre il numero di risposte è possibile inviare ai nodi interrogati il dato ricercato in modo da permettere un primo filtraggio locale dei risultati.

La figura 4.8 mostra la sottomissione di un'interrogazione secondo queste modalità, mentre in figura 4.9 è mostrata la risposta da parte dei peer interrogati.

Pur essendo un metodo esatto per la risoluzione del problema, questo tipo di approccio comporta un notevole impiego di banda e di spazio di memorizzazione a causa dell'elevata replicazione dei messaggi e delle informazioni memorizzate, rendendo opportuna la ricerca di metodi alternativi più economici.

In questa tesi, viste le considerazioni espresse in sezione 2.7 viene sperimentato un approccio basato sull'impiego di funzioni LSH [13] per la generazione degli identificatori associati ai profili. Tale metodo, pur consentendo un notevole risparmio di banda e di spazio di memorizzazione rispetto a quello basato sull'associazione delle chiavi ai singoli attributi, è però *probabilistico*.

Per quanto i margini di errore teorici siano notevolmente ridotti, è comunque necessario valutare l'accettabilità della perdita di qualità dei risultati rispetto al contentimento dei costi ottenuto.

Nelle sezioni successive, dopo aver illustrato le proprietà di questo tipo di funzioni, ne viene descritta una famiglia, con un relativo algoritmo di calcolo.

Successivamente, per ciascuno dei tipi di profilo descritti nella sezione precedente, viene descritta l'applicazione dei due metodi di indicizzazione e

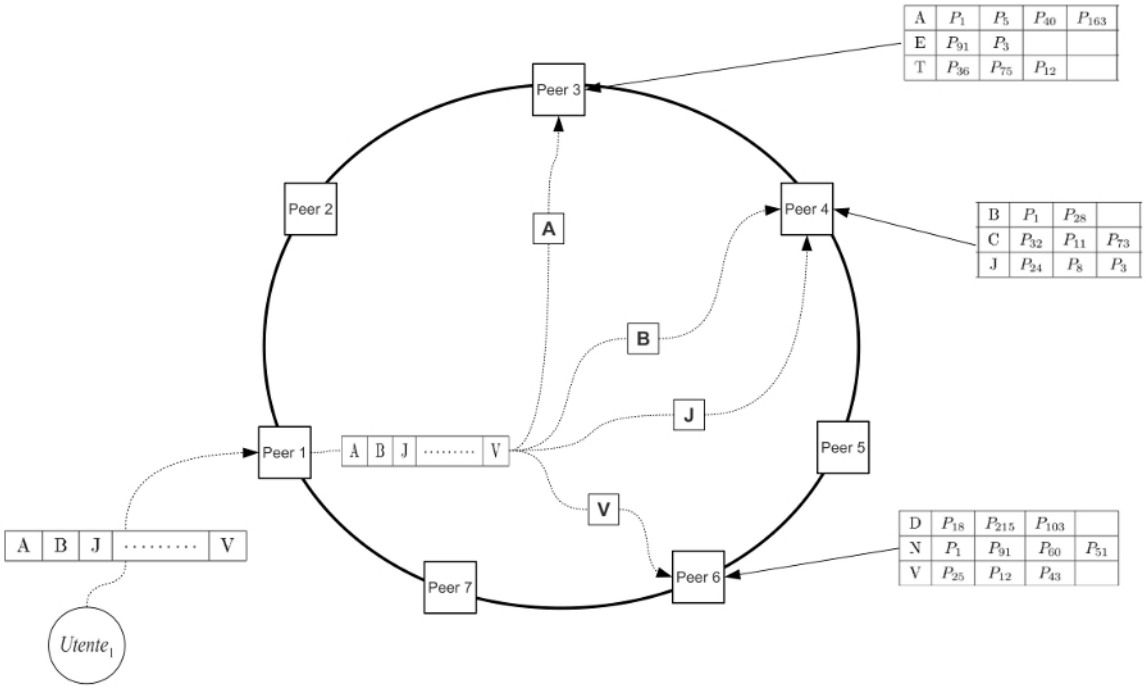


Figura 4.8: Ricerca per similarità, indicizzazione dei singoli attributi: interrogazione.

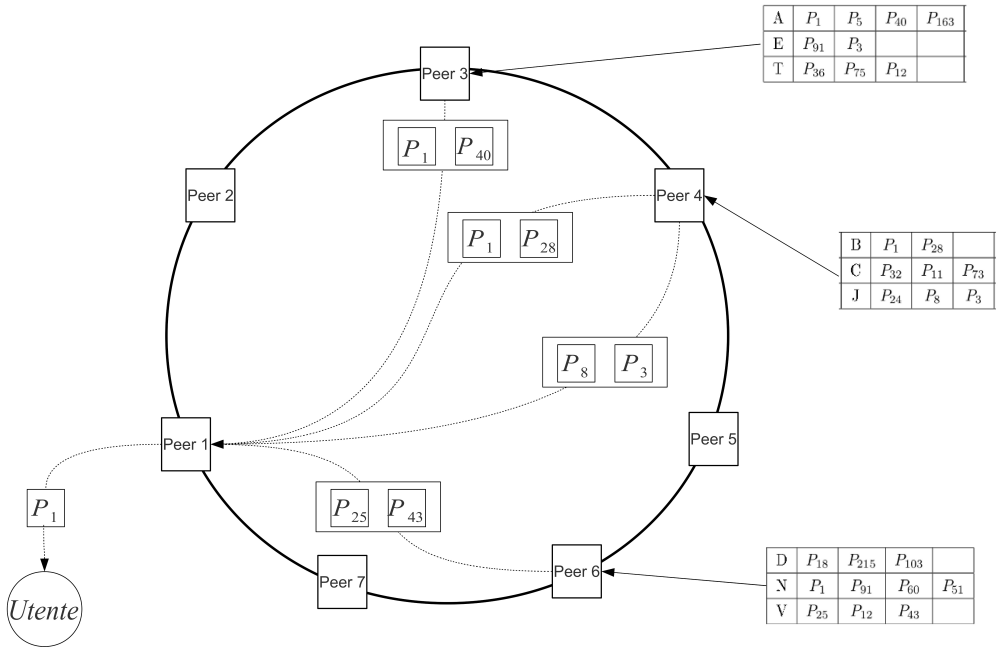


Figura 4.9: Ricerca per similarità, indicizzazione dei singoli attributi: risposte.

ricerca per similarità alle operazioni di ricerca dei profili e di mantenimento dell'indice delle comunità.

Per ciascuna combinazione di tipo di profilo e metodo di indicizzazione viene indicata una stima dei costi in termini di traffico generato e di spazio impiegato per la memorizzazione dei profili delle comunità, supponendo che i profili degli utenti e delle comunità siano definiti rispettivamente da $|U|$ e $|C|$ oggetti ciascuno e che ogni comunità abbia $|A|$ punti di accesso, su una DHT composta da X nodi, che memorizza Com descrittori di comunità .

4.2.1 Locality Sensitive Hash Functions

Date una famiglia di funzioni hash F e una funzione di similarità s operanti sul dominio D, F è detta *locality sensitive* [13] rispetto a s se, per ogni funzione h appartenente alla famiglia F, la probabilità che l'applicazione di h agli elementi di un qualunque insieme $A \subseteq D$ produca lo stesso risultato dell'applicazione a quelli di un qualunque insieme $B \subseteq D$ è pari alla similarità tra i due insiemi secondo s.

Più formalmente, F deve godere della seguente proprietà: $\forall h \in F. \forall A \subseteq D, B \subseteq D. Pr[h(A) = h(B)] = s(A, B)$, dove Pr è una funzione di probabilità e $s(A, B) \in [0, 1]$.

Impiegando funzioni di questo tipo è possibile costruire tabelle hash in cui valori simili vengono mappati nel medesimo *bucket* con una probabilità elevata.

Applicando ai sottoinsiemi di un certo dominio più funzioni LSH scelte uniformemente a caso è possibile, con un costo contenuto, incrementare la probabilità di ottenere almeno una collisione riducendo al contempo quella di ottenere falsi positivi.

Questo risultato viene ottenuto calcolando, per ciascun insieme di dati, un numero $n > 1$ di identificatori hash, ciascuno dei quali ottenuto effettuando un'operazione di OR esclusivo sui risultati dell'applicazione di m funzioni LSH (impiegando quindi un totale di $n \times m$ funzioni LSH).

Dato un insieme $g = \{h_1, h_2, ..., h_m\}$ di funzioni hash scelte casualmente con distribuzione uniforme da una famiglia di funzioni LSH rispetto alla

metrica s ed i dati A e B, la probabilità che tutte le funzioni appartenenti a g diano lo stesso risultato per i due dati è $Pr[g(A) = g(B)] = p^m$, dove $p = s(A, B)$, e la probabilità che ciò non accada $1 - p^m$. Considerando n gruppi di funzioni LSH $g_1, g_2, ..., g_n$ e definito la probabilità che in nessun caso si ottenga $g(A) = g(B)$ è quindi $(1-p^m)^n$, di conseguenza la probabilità che venga generato almeno un hash in comune per A e B è $1 - (1 - p^m)^n$.

Per opportune scelte di m ed n, questo valore si rivela molto elevato, ad esempio per $p = 0,7$, $n = 20$ ed $m = 5$ la probabilità è pari a 0,975.

La probabilità di ottenere almeno una collisione cresce all'aumentare di n e al decrescere di m, ma è necessario effettuare una scelta di compromesso poiché per valori troppo bassi di m c'è il rischio di generare molti falsi positivi, mentre un l'uso di un numero di identificatori hash troppo alto farebbe perdere buona parte dei vantaggi computazionali in alcuni tipi di applicazioni.

Queste proprietà vengono sfruttate per la risoluzione approssimata di problemi che richiedono il confronto di dati caratterizzati da alte dimensionalità allo scopo di individuare elementi simili, come ad esempio la ricerca di duplicati nell'ambito dell'indicizzazione e della memorizzazione di testi, la ricerca di immagini o file audio basata sulla similarità.

In questo modo è possibile ottenere risultati sufficientemente precisi evitando di confrontare ciascun dato con tutti gli altri elementi del dominio, abbassando così i costi computazionali.

L'uso di questa tecnica permette inoltre di evitare i problemi derivanti dalla cosiddetta *curse of dimensionality* [38], che comporta la perdita di significatività delle funzioni di distanza nel caso in cui la dimensionalità dei dati sia molto elevata.

4.2.1.1 Min-wise Independent Permutations

Le *min-wise independent permutations*, definite in [28] sono una famiglia di permutazioni definite sull'insieme dei numeri interi.

Sia S_nl'insieme di tutte le possibili permutazioni degli elementi dell'insieme di numeri interi $[n] = \{0, 1, \ldots, n\}$ e sia $F \subseteq S_n$una famiglia di permuta-

zioni dell'insieme $[n]$. F è detta *min-wise independent* se, per ciascun $X = \{x_1, x_2, \ldots x_n\} \subseteq [n]$ e per ogni $x \in X$, scelta casualmente con probabilità uniforme una permutazione $\pi \in F$ e definito $\pi(X) = \{\pi(x_1), \pi(x_2), \ldots \pi(x_n)\}$ si ha

$$P(min\{\pi(X)\} = \pi(x)) = \frac{1}{|X|}.$$

In altri termini, purché la permutazione π sia scelta casualmente tra quelle appartenenti alla famiglia F, ciascun elemento di X ha la stessa probabilità di generare il minimo elemento dell'immagine di $\pi(X)$.

Una famiglia di min-wise independent permutations è definita in [28] con la seguente costruzione.

Posto per semplicità $n = 2^r$ per un qualche r, la costruzione di una permutazione appartenente a questa famiglia avviene in r passi.

Inizialmente gli elementi dell'insieme $[n]$ vengono suddivisi in due bucket, detti "superiore" e "inferiore", procedimento che può essere effettuato in $\binom{n}{n/2}$ modi. Ciascuno dei possibili partizionamenti può essere rappresentato da una stringa binaria di n bit, di valore equamente suddiviso tra 0 e 1. Scelta casualmente con distribuzione di probabilità uniforme una di queste stringhe tra tutte quelle possibili, la ripartizione avviene considerando gli elementi di $[n]$ nel loro ordine naturale e assegnando l'i-esimo elemento al bucket superiore se l'i-esimo bit della stringa ha valore 1, al bucket inferiore altrimenti.

Al passo successivo un procedimento analogo viene effettuato separatamente per entrambi i bucket, utilizzando, per definire il partizionamento di ciascuno di essi, la stessa stringa di $\frac{n}{2}$ bit scelta casualmente tra le $\binom{n/2}{n/4}$ che presentano esattamente $\frac{n}{4}$ bit di valore 1.

Al generico passo i, ciascuno dei 2^{i-2} bucket individuati al passo $i-1$ viene suddiviso in due metà impiegando una stringa di $\frac{n}{2^{i-1}}$ bit con caratteristiche analoghe alle precedenti, questo procedimento viene ripetuto fino a costruire n bucket di dimensione 1, a ciascuno dei quali è assegnato un elemento di $[n]$.

Ripartendo in questo modo gli elementi dell'insieme $[n]$ è possibile stabili-

re un semplice ordinamento considerando ad ogni passo gli elementi assegnati alla metà "superiore" di ciascun bucket minori di quelli assegnati alla metà "inferiore", ottenendo dopo l'ultimo partizionamento un nuovo ordinamento totale dell'insieme.

Numerando gli n bucket ottenuti dal partizionamento dell'insieme di partenza in accordo a tale ordinamento e mettendo l'indice di ciascun bucket in corrispondenza con l'intero ad esso assegnato si ottiene il risultato della permutazione dell'insieme $[n]$.

Ciascuna delle $\prod_{i=1}^{log_2 n} \binom{n/2^{i-1}}{n/2^i}$ possibili successioni di r stringhe binarie composte da un numero uguale di 0 e 1, ciascuna di lunghezza pari alla metà della precedente, individua una permutazione dell'insieme $[n]$ appartenente alla famiglia descritta da questa procedura.

Poiché le stringhe di bit che definiscono il partizionamento dei vari bucket sono sempre scelte casualmente con probabilità uniforme e dal momento che i bit di ciascuna di esse sono equamente suddivisi in 0 e 1, ad ogni passo della procedura di permutazione ciascun elemento di $[n]$ ha esattamente la stessa probabilità di essere assegnato alla metà superiore o a quella inferiore del suo bucket di appartenenza.

Dopo il generico passo i, quindi, la probabilità che un dato elemento dell'insieme sia stato assegnato ad uno specifico bucket è pari a $1/2^i$, di conseguenza, alla fine del procedimento di permutazione, ogni elemento dell'insieme ha una probabilità pari a $1/2^r = 1/2^{|[n]|}$ di essere associato ad un dato bucket.

Questa proprietà vale anche per ogni sottoinsieme di $[n]$, di conseguenza, dati una permutazione π appartenente a questa famiglia e un qualunque insieme $X \subseteq [n]$, ogni intero $x \in X$ ha la medesima probabilità $1/|X|$ di essere associato all'elemento minimo dell'insieme $\pi(X) \subseteq \pi([n])$. Le permutazioni appartenenti a questa famiglia sono quindi min-wise independent permutations.

In [28] è descritta anche una generalizzazione di questa definizione per insiemi di interi di dimensione arbitraria.

4.2.1.2 Una famiglia di funzioni LSH basata su min-wise independent permutations

A partire da ciascuna famiglia di min-wise independent permutations è possibile costruire una famiglia di funzioni hash che godono della proprietà LSH rispetto alla metrica di Jaccard $j(A,B) = \frac{|A \cap B|}{|A \cup B|}$.

Data una min-wise independent permutation π definita sugli elementi dell'insieme di interi U ed $A = \{a_1, a_2, ..., a_n\} \subseteq U$, la funzione hash h_π è definita come $h_\pi = min\{\pi(A)\} = min\{\pi(a_1), \pi(a_2), ..., \pi(a_n)\}$, ossia applicando la permutazione π a ciascun elemento dell'insieme A e scegliendo quindi il minimo dei risultati.

Dati due insiemi A e B risulta $x = h_\pi(A) = h_\pi(B) \Leftrightarrow \pi^{-1}(x) \in A \cap B$, in tal caso è ovviamente verificata anche la condizione $x = h_\pi(A \cup B)$.

Poiché π è una min-wise independent permutation, ciascun elemento di $A \cup B$ ha la stessa probabilità $1/|A \cup B|$ di generare il minimo di $\pi(A \cup B)$, quindi la probabilità di avere $h_\pi(A) = h_\pi(B)$ è pari a $|A \cap B| \times \frac{1}{|A \cup B|} = j(A,B)$.

Questa famiglia di funzioni, benché definita su insiemi di numeri interi, può essere impiegata anche a partire da dati di tipo diverso, dopo averne codificato tutti gli elementi tramite una funzione hash.

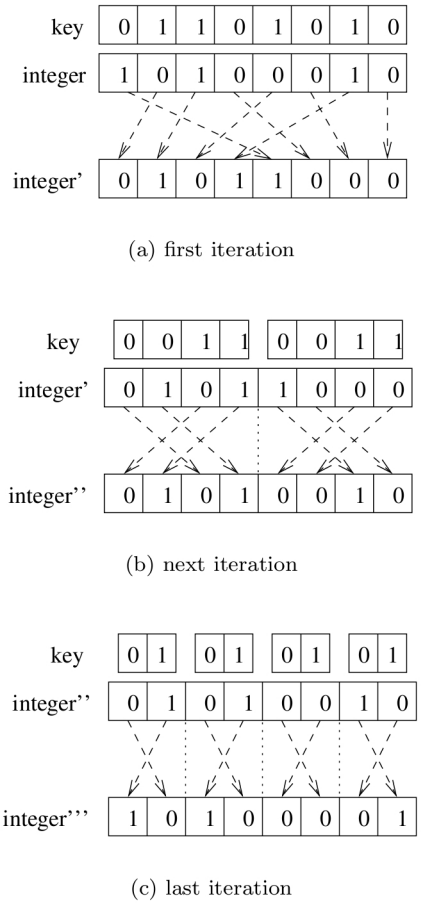
Poiché per funzioni di questo tipo la probabilità di collisione è trascurabile, è lecito assumere che i numeri interi generati in questo modo siano in corrispondenza biunivoca con i dati di partenza e che, dati gli insiemi A e B e detti rispettivamente A' e B' gli insiemi ottenuti codificandone gli elementi, sia verificata la condizione $j(A,B) = j(A',B')$.

Dal momento che, come dimostrato in [28], il numero di bit necessario a rappresentare una min-wise independent permutation definita su insiemi di n elementi è $\Omega(n)$ e $O(4^n)$, nella pratica si ricorre a delle approssimazioni di tali permutazioni, che possano essere implementate per insiemi di dimensioni ragionevoli.

In [14] è descritto un algoritmo per il calcolo di un'approssimazione delle min-wise independent permutations, basato sull'applicazione ai singoli bit della rappresentazione binaria degli elementi dell'insieme da permutare della

procedura esposta nella sottosezione 4.2.1.1, che ha prodotto sperimentalmente risultati validi.

In figura 4.10 è riportato un esempio di permutazione di un intero di 8 bit.



(a) first iteration

(b) next iteration

(c) last iteration

Figura 4.10: Un esempio di schema di permutazione

Un'altra approssimazione delle min-wise independent permutations è data, come esposto in [28], da una famiglia di trasformazioni lineari definite come $\pi(x) = ax + b(mod\ P)$, dove $a \neq 0$ è un valore scelto casualmente e P è un numero primo molto grande, anch'esso selezionato in modo casuale.

In [24] queste due approssimazioni vengono confrontate paragonando, al variare della similarità tra gli argomenti delle query e gli elementi dell'indice,

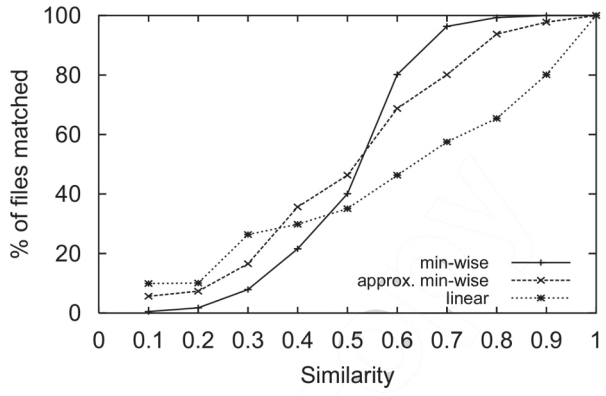


Figura 4.11: Min-wise independent permutations: confronto tra approssimazioni.

la percentuale di interrogazioni per le quali viene ottenuta almeno una corrispondenza tra gli identificatori calcolati per l'argomento e quelli associati a file simili.

I risultati del confronto, per valori dei parametri n e m pari rispettivamente a 20 e a 5, sono mostrati in figura 4.11, dove, con "min-wise" è indicato l'algoritmo di permutazione dei bit e con "linear" le trasformazioni lineari.

È possibile notare come, per valori di similarità pari o superiori a 0,5, i risultati ottenuti impiegando l'algoritmo di permutazione dei bit migliorino sensibilmente rispetto a quelli garantiti dalle trasformazioni lineari, con un divario che si amplia rapidamente per i valori di similarità più "interessanti" dal punto di vista della ricerca di profili simili.

Le probabilità di collisione tra identificatori ottenute con il primo algoritmo sono inoltre sensibilmente più vicine a quelle teoriche rispetto a quanto ottenuto utilizzando le trasformazioni lineari.

Nel caso in cui dovesse essere utile considerare come risultati accetabili degli oggetti caratterizzati da una similarità piuttosto bassa con quello cercato, per esempio se i dati dovessero essere per loro natura particolarmente sparsi, sarebbe comunque preferibile e più coerente utilizzare un valore più basso del parametro m piuttosto che un'approssimazione peggiore delle min-wise independent permutations.

Per questi motivi è stato scelto di utilizzare, per l'implementazione del-

l'indicizzazione con funzioni LSH, l'approssimazione proposta in [14].

L'algoritmo 4.1 illustra la generazione di un insieme di n schemi di permutazione, ciascuno composto da m funzioni, mentre gli algoritmi 4.2, 4.3 e 4.4 mostrano le varie fasi del calcolo, per un insieme di oggetti *values*, di di n identificatori lunghi b bit a partire da $n \times m$ schemi di permutazione.

In particolare, l'algoritmo 4.2 mostra la codifica degli elementi di *values* in numeri interi attraverso una funzione hash robusta, l'applicazione a tali interi degli schemi di permutazione e la combinazione dei risultati ottenuti tramite l'OR esclusivo per generare gli identificatori LSH, l'algoritmo 4.3 è relativo alla generazione, in base ad uno schema di permutazione definito dall'insieme di chiavi *keys*, del valore $\pi(val)$ per un oggetto $val \in values$, mentre l'algoritmo 4.4 è relativo all'applicazione a un intero di un singolo passo dello schema di permutazione.

Algoritmo 4.1 Generazione di un insieme di schemi di permutazione

Input: il numero n di schemi da generare, il numero m di permutazioni per ogni schema, il numero b di bit degli interi da permutare.

Output: l'insieme di $n \times m \times \log_2(b)$ array di bit *keys* che costituiscono gli schemi di permutazione.

```
1: for i = 0 to n − 1 do
2:    // Genera n insiemi di schemi di permutazione
3:    for j = 0 to m − 1 do
4:       // Genera il j-esimo schema dell'i-esimo insieme
5:       s ← b; // La prima chiave dello schema è di b bit
6:       k ← 0;
7:       while s > 1 do
8:          // Finché la nuova chiave da generare è di almeno 2 bit
9:          key ← bitArray(s, 0); // Inizializza un array di s bit uguali a 0
10:         setCounter ← 0 // Inizializza il contatore dei bit impostati a 1
11:         while setCounter < s/2 do
12:            bitToSet ← randomInteger(s); // Genera un numero intero
               casuale tra 0 e s-1
13:            if key[bitToSet] = 0 then
14:               key[bitToSet] ← 1;
15:               setCounter ← setCounter + 1;
16:            end if
17:         end while
18:         keys[i][j][k] ← key; // Memorizza la chiave di s bit appena
            generata
19:         k ← k + 1;
20:         s ← s/2; // La prossima chiave sarà di s/2 bit
21:      end while
22:   end for
23: end for
24: return keys;
```

Algoritmo 4.2 Calcolo degli identificatori

Input: l'insieme di oggetti per i quali generare gli identificatori *values* , gli schemi di permutazione *schemes*, la dimensione in bit b degli identificatori da generare.

Output: l'insieme di identificatori *IDs*.

```
1: for t = 0 to values.Size − 1 do
2:    integersToProcess[t] ← Hash(values[t]); // Usando una funzione
      hash robusta calcola il numero intero associato a ciascun oggetto.
3: end for
4: for all permutationSet in schemes do
5:    currentID ← null // Dichiara un identificatore per lo schema di
      permutazione corrente.
6:    for all permutation in permutationSet do
7:       min ← null;
8:       for all valTP in integersToProcess do
9:          pVal ← ApplyPermutation(valTP, permutation, b); // Applica
            a valTP lo schema di permutazione corrente.
10:         if min = null or pVal < min then
11:            min ← pVal; // Se necessario aggiorna il valore attuale del
               minimo.
12:         end if
13:      end for
14:      if currentID = null then
15:         currentID ← min;
16:      else
17:         currentID ← currentID ⊕ min; // Combina con un'operazione
            di or esclusivo i risultati delle m permutazioni.
18:      end if
19:   end for
20:   IDs.Add(currentID);
21: end for
22: return IDs;
```

Algoritmo 4.3 Calcolo degli identificatori: ApplyPermutation

Input: il numero intero *val* da permutare, l'insieme di chiavi *keys*, la dimensione in bit b di *val*.

Output: l'intero pVal ottenuto permutando i bit di *val* tramite le chiavi contenute in *keys*.

1: $toShuffle \leftarrow val$;
2: **for** $i = 0$ to $keys$.Size() **do**
3: $keySize \leftarrow b \div 2^h$; // Ogni chiave è lunga la metà della precedente
4: $pVal \leftarrow$ ApplyKey($toShuffle$, $keys[i]$, b, $keySize$); // Permuta i bit di toShuffle in base all'i-esima chiave
5: $toShuffle \leftarrow pVal$; // Salva il risultato parziale per la prossima iterazione
6: **end for**
7: **return** $pVal$;

Algoritmo 4.4 Calcolo degli identificatori: ApplyKey

Input: l'array di bit *val* che rappresenta il numero intero da permutare, la chiave *key* in base alla quale effettuare la permutazione, la dimensione in bit b di *val*, la dimensione in bit *keySize* di *key*.

Output: l'intero pVal ottenuto permutando i bit di *val* tramite la chiave *key*.

```
1:  lowerExtreme ← 0;
2:  upperExtreme ← keySize − 1;
3:  while upperExtreme < b do
4:    // Vengono effettuate b ÷ keySize iterazioni, durante ciascuna delle
      quali vengono permutati keySize bit.
5:    middle ← (lowerExtreme+upperExtreme)/2; // indice di inizio della seconda
      metà del blocco di bit corrente.
6:    nSet ← 0; // Contatore dei bit di key pari a 1 letti.
7:    nClear ← 0; // Contatore dei bit di key pari a 0 letti.
8:    for i = 0 to keySize − 1 do
9:      if key[i] = 1 then
10:       if val[lowerExtreme + i] = 1 then
11:         pVal[lowerExtreme + nSet] ← 1;
12:       else
13:         pVal[lowerExtreme + nSet] ← 0;
14:       end if
15:       nSet ← nSet + 1;
16:     else
17:       if val[lowerExtreme + i] = 1 then
18:         pVal[middle + nSet] ← 1;
19:       else
20:         pVal[middle + nSet] ← 0;
21:       end if
22:       nClear ← nClear + 1;
23:     end if
24:   end for
25:   lowerExtreme ← lowerExtreme + keySize;
26:   upperExtreme ← upperExtreme + keySize;
27:   // Calcola gli estremi del prossimo blocco di bit.
28: end while
29: return pVal;
```

4.2.2 Applicazione dei metodi di indicizzazione a vettori n-dimensionali di oggetti

Nel caso di entrambi i metodi di indicizzazione proposti le chiavi vengono calcolate senza considerare i pesi associati agli oggetti. Questo è dovuto al fatto che è estremamente improbabile che due o più profili condividano *esattamente* le stesse coppie $\langle attributo, peso \rangle$, quindi considerare *simili* esclusivamente profili con questa caratteristica già in fase di indicizzazione sarebbe restrittivo al punto da rendere impossibile il funzionamento del sistema.

I pesi associati agli attributi vengono quindi valutati, successivamente all'indicizzazione, nel momento in cui i nodi interrogati selezionano i profili da inviare in risposta a un'interrogazione per similarità e, nuovamente, nel momento in cui il peer richiedente effettua l'intersezione e il filtraggio dei risultati ottenuti. In entrambi i casi viene impiegata la metrica 4.1.

Nel caso in cui un descrittore di comunità debba essere rimosso dall'indice o da alcuni nodi, è sufficiente, poiché il valore da eliminare è determinato esattamente, inviare assieme alla richiesta di cancellazione un identificatore del descrittore calcolato con una funzione hash robusta o l'identificatore della comunità, anziché l'intero profilo, perché i peer che devono eseguire questa operazione possano individuarlo tra quelli associati alla stessa chiave.

In questo modo i messaggi di richiesta di cancellazione di un descrittore hanno una dimensione limitata, che si può considerare costante.

4.2.2.1 Indicizzazione dei singoli attributi

In figura 4.12 è illustrata la generazione delle chiavi a partire dai singoli attributi, in cui F è una funzione hash robusta.

Impiegando il metodo di indicizzazione basato sui singoli attributi del profilo, per ciascun oggetto che occorre in almeno un profilo di comunità viene creata una entry sulla DHT, contenente i descrittori di tali comunità.

La ricerca dei profili di comunità più simili a quello di un utente viene effettuata, secondo l'algoritmo 4.5, eseguendo una ricerca sulla DHT che memorizza i descrittori di comunità per ogni attributo del profilo. Ciascuno dei peer interrogati invia i propri k risultati più significativi secondo la metrica

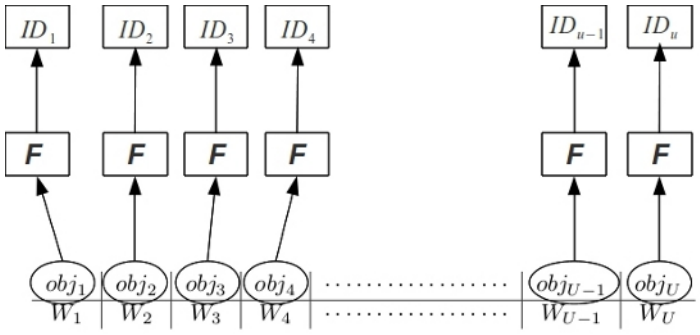


Figura 4.12: Vettore di oggetti pesati: indicizzazione per singoli attrbuti

di similarità 4.1, quindi il nodo richiedente effettua l'intersezione dei risultati, seleziona il più significativo, ottiene gli indirizzi dei peer di contatto delle comunità selezionata da AddressesDHT e la invia al richiedente insieme al descrittore della comunità.

In seguito alla variazione del profilo di una comunità, in base all'algoritmo 4.9, il nuovo vettore viene inviato ai peer responsabili di ciascun termine in esso contenuto, mentre a quelli responsabili degli eventuali attributi eliminati viene inviata la richiesta di rimuovere il riferimento alla comunità dalla relativa entry della DHT.

La memorizzazione di un nuovo descrittore comporta il calcolo di una chiave per ogni attributo del profilo e la memorizzazione delle copie sulla DHT, operazione illustrata dall'algoritmo 4.7.

In entrambi i casi precedenti, i nodi destinatari di ciascuna richiesta di memorizzazione di un descrittore inviano in risposta i k descrittori caratterizzati dal profilo più simile a quello da memorizzare secondo la metrica 4.1. Il nodo richiedente seleziona i descrittori più significativi in assoluto ed invia ai peer di contatto delle rispettive comunità la notifica della presenza di un nuovo profilo. Queste operazioni sono mostrate nell'algoritmo 4.8.

La cancellazione di un descrittore di comunità avviene calcolando una chiave per ciascun attributo del profilo ed inviando una richiesta di rimozione per ciascuna chiave secondo l'algoritmo 4.11.

La risposta di un peer interrogato a una ricerca per similarità è mostrata nell'algoritmo 4.12, mentre gli algoritmi 4.13 4.14 sono relativi rispettiva-

mente alla memorizzazione del descrittore di una nuova comunità e all'aggiornamento del profilo di una comunità esistente, in entrambi i casi il peer provvede all'invio al richiedente dei descrittori delle comunità simili.

La selezione dei profili più simili ad uno dato è mostrata nell'algoritmo 4.15, mentre il confronto tra profili in base alla metrica 4.1 è illustrato dell'algoritmo 4.16.

Algoritmo 4.5 Ingresso di un utente nel sistema: indicizzazione per singoli attributi, interesse unico

---

Input: il profilo P dell'utente, il numero massimo k di profili simili richiesti a ciascun peer interrogato.
Output: il descrittore della comunità raccomandata e gli indirizzi dei relativi peer di accesso.

1: $IDs \leftarrow \emptyset$;
2: **for all** *obj* in $P.Attributes$ **do**
3: IDs.Add(Hash(*obj*)); // Associa al profilo un identificatore per ciascun attributo
4: **end for**
5: *retval* $\leftarrow$ GetSimilarCommunitiesData(P,IDs,k); // Vedi algoritmo 4.6
6: **return** *retval*;

Algoritmo 4.7 Creazione di una nuova comunità: indicizzazione per singoli attributi

---

Input: il descrittore D della nuova comunità, la lista A degli indirizzi dei peer di accesso, il numero l di comunità a cui inviare la notifica, il numero massimo k di profili simili richiesti a ciascun peer interrogato.

1: $IDs \leftarrow \emptyset$;
2: **for all** *obj* in $D.Profile.Attributes$ **do**
3: IDs.Add(Hash(*obj*)); // Associa al profilo un identificatore per ciascun attributo
4: **end for**
5: StoreAndNotify(IDs, D, A, l, k); // Memorizza il nuovo descrittore e invia la notifica alle comunità più affini. Vedi algoritmo 4.8

Algoritmo 4.6 Ingresso di un utente nel sistema: GetSimilarCommunitiesData

Input: il profilo P dell'utente, l'insieme di identificatori *IDs* calcolati per P, il numero massimo k di profili simili richiesti a ciascun peer interrogato.
Output: il descrittore della comunità raccomandata e gli indirizzi dei relativi peer di accesso.

1: $similarCommunities \leftarrow \emptyset$;
2: **for all** id in IDs **do**
3: $similarCommunities$.Add(ProfilesDHT.GetSimilarCommunities(id, P, k)); // Per ogni identificatore ricerca i profili di comunità più simili a P.
4: **end for**
5: Sort($similarCommunities, P$); // Ordina i risultati in base alla similarità con P.
6: $selectedCommunity \leftarrow$ first($similarCommunities$);
7: $addresses \leftarrow$ AddressesDHT.Get($selectedCommunity.ID$); // Ricerca gli indirizzi dei peer di accesso per la comunità selezionata.
8: $retval \leftarrow \langle selectedCommunity, addresses \rangle$;
9: **return** $retval$;

Algoritmo 4.8 Creazione di una nuova comunità / variazione di profilo: StoreAndNotify.

Input: l'insieme *IDs* degli identificatori calcolati per il descrittore da memorizzare, il descrittore D della nuova comunità, la lista A degli indirizzi dei peer di accesso, il numero l di comunità a cui inviare la notifica, il numero massimo k di profili simili richiesti a ciascun peer interrogato.

1: $similarCommunities \leftarrow \emptyset$;
2: **for all** id in IDs **do**
3: $similarCommunities$.Add(ProfilesDHT.StoreDescriptor(id, D, k)); // Per ogni identificatore memorizza D e riceve i descrittori delle comunità simili.
4: **end for**
5: AddressesDHT.Put($D.ID$,A);
6: Sort($similarCommunities, D.Profile$); // Ordina i descrittori ottenuti in base alla similarità con il profilo di D.
7: $selectedCommunities \leftarrow$ sublist($similarCommunities, 0, l$); // Seleziona le l comunità più simili a D
8: **for all** $sC \in selectedCommunities$ **do**
9: $addresses \leftarrow$ AddressesDHT.Get($sC.ID$);
10: **for all** $addr \in addresses$ **do**
11: NotifiyNewCommunity($addr, D$); // Invia il nuovo descrittore ai peer di accesso della comunità
12: **end for**
13: **end for**

Algoritmo 4.9 Variazione del profilo di una comunità: indicizzazione per singoli attributi

Input: il vecchio descrittore della comunità oD, il nuovo descrittore della comunità nD, il numero l di comunità a cui inviare la notifica, il numero massimo k di profili simili richiesti a ciascun peer interrogato.

1: $oldIDs \leftarrow \emptyset$;
2: **for all** $oldObj$ in $oD.Profile.Attributes \setminus nD.Profile.Attributes$ **do**
3: $oldIDs$.Add(Hash($oldObj$)); // Calcola gli identificatori degli attributi non presenti nel nuovo profilo.
4: **end for**
5: **for all** $mantainedObj$ in $oD.Profile.Attributes \cap nD.Profile.Attributes$ **do**
6: $mantainedIDs$.Add(Hash($mantainedObj$)); // Calcola gli identificatori degli attributi presenti in entrambi.
7: **end for**
8: **for all** $newObj$ in $nD.Profile.Attributes \setminus oD.Profile.Attributes$ **do**
9: $newIDs$.Add(Hash($newObj$)); // Calcola gli identificatori degli attributi presenti solo nel nuovo profilo.
10: **end for**
11: PerformUpdate(oD, nD, $oldIDs$, $mantainedIDs$, $newIDs$, l, k); // Vedi algoritmo 4.10

Algoritmo 4.10 Variazione del profilo di una comunità: PerformUpdate

Input: il vecchio descrittore oD, il nuovo descrittore nD, la lista *oldIDs* degli identificatori associati a oD e non a nD, la lista *mantainedIDs* degli identificatori associati sia a oD che a nD, la lista *newIDs* degli identificatori associati a nD e non a oD, il numero l di comunità a cui inviare la notifica, il numero massimo k di profili simili richiesti a ciascun peer interrogato.

1: **for all** $oldId$ in $oldIDs$ **do**
2: ProfilesDHT.RemoveDescriptor($oldId$, Hash(oD))); // Rimuove le copie di oD corrispondenti agli attributi eliminati.
3: **end for**
4: $similarCommunities \leftarrow \emptyset$;
5: **for all** mId in $mantainedIDs$ **do**
6: $similarCommunities$.Add(ProfilesDHT.UpdateDescriptor(mId, nD, Hash(oD), k)); // Per ogni identificatore memorizza nD, rimuove oD e riceve i descrittori delle comunità simili escluso oD.
7: **end for**
8: **for all** nId in $newIDs$ **do**
9: $similarCommunities$.Add(ProfilesDHT.StoreDescriptor(nId, nD, k)); // Per ogni identificatore memorizza nD e riceve i descrittori delle comunità simili.
10: **end for**
11: Sort($similarCommunities, nD.Profile$); // Ordina i descrittori ottenuti in base alla similarità con nD.
12: $selectedCommunities \leftarrow$ sublist($similarCommunities, 0, l$); // Seleziona le l comunità più simili a nD
13: **for all** $cS \in selectedCommunities$ **do**
14: $addresses \leftarrow$ AddressesDHT.Get($cS.ID$);
15: **for all** $addr \in addresses$ **do**
16: NotifiyNewDescriptor($addr, D$); // Invia il nuovo descrittore ai peer di accesso della comunità
17: **end for**
18: **end for**

Algoritmo 4.11 Scioglimento di una comunità: indicizzazione per singoli attributi

Input: il descrittore della comunità da eliminare D

1: $IDs \leftarrow \emptyset$;
2: **for all** *obj* in $D.Attributes$ **do**
3: IDs.Add(Hash(*obj*)); // Associa al profilo un identificatore per ciascun attributo
4: **end for**
5: **for all** *id* in IDs **do**
6: ProfilesDHT.Remove(*id*, Hash(D); // Rimuove D dalla DHT dei profili.
7: **end for**
8: AddressesDHT.Remove($D.ID$); // Rimuove la lista degli indirizzi dei peer di accesso.

Algoritmo 4.12 ProfilesDHT: GetSimilarCommunities.

Input: il profilo P, l'identificatore *id* associato a P, il numero massimo k di risultati richiesti

Output: la lista *mostSimilarCommunities* dei descrittori delle comunità più affini a P con i relativi valori di similarità

1: *mostSimilarCommunities* ← SelectSimilarCommunities(P,*id*,k); // Vedi algoritmo 4.15
2: **return** *mostSimilarCommunities*;

Algoritmo 4.13 ProfilesDHT: StoreDescriptor.

Input: il descrittore D della nuova comunità, l'identificatore *id* associato a D, il numero massimo k di risultati richiesti

Output: la lista *mostSimilarCommunities* dei descrittori delle comunità più affini a P con i relativi valori di similarità

1: *mostSimilarCommunities* ← SelectSimilarCommunities(D.GetProfile,*id*,k); // Seleziona localmente i descrittori delle al più k comunità più simili a quella descritta da D. Vedi algoritmo 4.15
2: *localTable*.Put(*id*,D); // Memorizza D sulla DHT
3: **return** *mostSimilarCommunities*;

Algoritmo 4.14 ProfilesDHT: UpdateDescriptor.

Input: il nuovo descrittore D della nuova comunità, l'identificatore id associato a D, un valore hash H calcolato a partire dal vecchio descrittore, il numero massimo k di risultati richiesti

Output: la lista *mostSimilarCommunities* dei descrittori delle comunità più affini a P con i relativi valori di similarità

```
1: candidates ← localTable.Get(id); // Uno dei descrittori associati a id
   deve essere rimosso dalla DHT
2: for all descr in candidates do
3:    // Cerca ed elimina il vecchio descrittore della comunità
4:    if H = Hash(descr) then
5:       localTab.Remove(id,descr);
6:    end if
7: end for
8: mostSimilarCommunities ← SelectSimilarCommunities(D.GetProfile,id,k);
   // Seleziona localmente i descrittori delle al più k comunità più simili a
   quella descritta da D. Vedi algoritmo 4.15
9: localTable.Put(id,D); // Memorizza D sulla DHT
10: return mostSimilarCommunities;
```

Algoritmo 4.15 SelectSimilarCommunities

Input: il profilo P, l'identificatore id associato a P, il numero massimo k di risultati richiesti

Output: la lista *selectedCommunities* dei descrittori delle comunità più affini a P con i relativi valori di similarità

```
1: candidates ← localTable.Get(id); // tutti i profili associati a id nella
   porzione della DHT gestita dal nodo sono potenzialmente simili a P
2: rankedDescriptors ← ∅
3: for all descr in candidates do
4:    rankedDescriptors.Add(<descr,CalculateSimilarity(P,descr.GetProfile());
      // Calcola la similarità tra il profilo della comunità individuata da
      descr e P, ne associa il valore a descr e salva il risultato
5: end for
6: Sort(rankedDescriptors, P); // Ordina i descrittori in base alla
   similarità con P
7: mostSimilarCommunities ← sublist(rankedDescriptors, 0, k); //
   Seleziona gli al più k descrittori da inviare al richiedente
8: return selectedCommunities;
```

Algoritmo 4.16 CalculateSimilarity, metrica 4.1.

Input: i vettori pesati A e B
Output: il valore sim della similarità tra A e B

```
1: sim ← 0;
2: for all a in A do
3:    if a ∈ B then
4:       sim ← sim + min(A.getWeight(a),B.getWeight(a));
5:    end if
6: end for
7: sim ← sim ÷ max(A.Size,B.Size);
8: return sim;
```

4.2.2.2 Indicizzazione tramite funzioni LSH

Utilizzando il metodo di indicizzazione basato sull'uso di funzioni LSH, gli identificatori di ciascun profilo vengono calcolati a partire dall'intero vettore di oggetti, impiegando un insieme fisso di funzioni, come mostrato in figura 4.13.

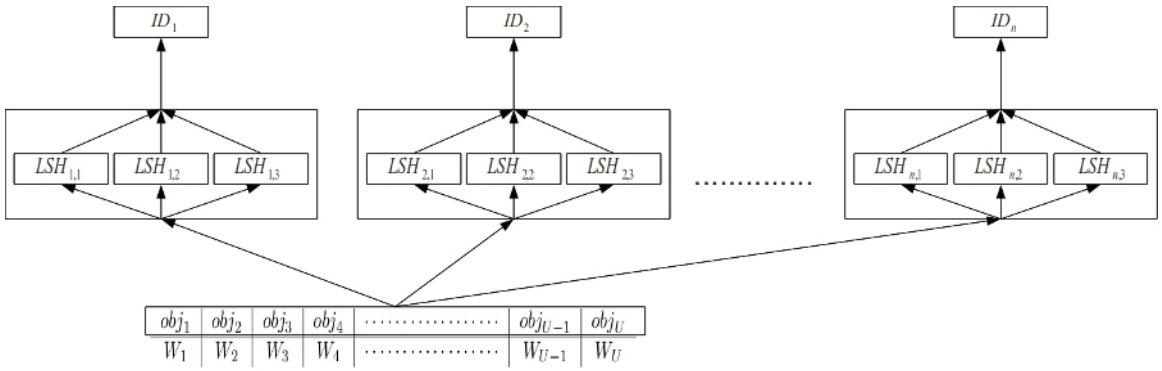


Figura 4.13: Vettore di oggetti pesati: indicizzazione con funzioni LSH

L'indice distribuito è costituito da una entry per ciascuno degli identificatori associati ad almeno un profilo. Ciascuna di tali entry contiene la lista dei descrittori di comunità che condividono il relativo identificatore.

La ricerca delle comunità più adatte agli interessi di un utente viene effettuata, seguendo l'algoritmo 4.17, calcolando un numero n costante e rela-

tivamente piccolo (10-20) di identificatori per il profilo dell'utente e inviando quindi una query sulla DHT per ciascuno di essi. Anche in questo caso la selezione dei risultati viene effettuata in due fasi.

L'aggiornamento del profilo di una comunità, in base all'algoritmo 4.19, comporta la generazione di n nuovi identificatori semantici per il nuovo vettore di oggetti, l'invio di una richiesta di rimozione del riferimento alla comunità ai peer responsabili dei vecchi identificatori e la memorizzazione del profilo aggiornato presso i peer responsabili di quelli appena generati.

La memorizzazione di un nuovo descrittore di comunità viene effettuata, impiegando l'algoritmo 4.18, calcolando gli n identificatori per il profilo e memorizzandone n copie anziché $|P|$ (dove $|P|$ è il numero di attributi del profilo) sulla DHT.

L'eliminazione di un descrittore, descritta dall'algoritmo 4.20, comporta il calcolo degli n identificatori corrispondenti e l'invio di una richiesta di cancellazione per ciascun identificatore.

Algoritmo 4.17 Ingresso di un utente nel sistema: indicizzazione con funzioni LSH, interesse unico

---

Input: il profilo P dell'utente, il numero massimo k di profili simili richiesti a ciascun peer interrogato, l'insieme S di funzioni LSH da impiegare per il calcolo degli identificatori.
Output: il descrittore della comunità raccomandata e gli indirizzi dei relativi peer di accesso.

1: $IDs \leftarrow \emptyset$;
2: **for all** $LSHFunction$ in S **do**
3: IDs.Add($LSHFunction(P.Attributes)$); // Associa al profilo un identificatore per ciascuna funzione LSH
4: **end for**
5: $retval \leftarrow$ GetSimilarCommunitiesData(P,IDs,k); // Vedi Algoritmo 4.6
6: **return** $retval$;

4.2.2.3 Complessità: indicizzazione per singoli attributi

La ricerca dei profili comporta l'invio sull'overlay network di una copia dello stesso ai peer responsabili di ciascuno degli agttributi che lo compon-

Algoritmo 4.18 Creazione di una nuova comunità: indicizzazione con funzioni LSH

Input: il descrittore D della nuova comunità, la lista A degli indirizzi dei peer di accesso, il numero l di comunità a cui inviare la notifica, il numero massimo k di profili simili richiesti a ciascun peer interrogato, l'insieme S di funzioni LSH da impiegare per il calcolo degli identificatori.

1: $IDs \leftarrow \emptyset$;
2: **for all** $LSHFunction$ in S **do**
3: IDs.Add($LSHFunction(D.Attributes)$); // Associa al profilo un identificatore per ciascuna funzione LSH
4: **end for**
5: StoreAndNotify(IDs, D, A, l, k); // Memorizza il nuovo descrittore e invia la notifica alle comunità più affini. Vedi Algoritmo 4.8

Algoritmo 4.19 Variazione del profilo di una comunità: indicizzazione con funzioni LSH

Input: il vecchio descrittore della comunità oD, il nuovo descrittore della comunità nD, il numero l di comunità a cui inviare la notifica, il numero massimo k di profili simili richiesti a ciascun peer interrogato, l'insieme S di funzioni LSH da impiegare per il calcolo degli identificatori.

1: $oldIDs \leftarrow \emptyset$;
2: **for all** $LSHFunction$ in S **do**
3: $oldIDs$.Add($LSHFunction(oD.Attributes)$); // Associa al profilo vD un identificatore per ciascuna funzione LSH
4: **end for**
5: **for all** $LSHFunction$ in S **do**
6: $newIDs$.Add($LSHFunction(nD.Attributes)$); // Associa al profilo nD un identificatore per ciascuna funzione LSH
7: **end for**
8: PerformUpdate(oD, nD, $oldIDs \setminus newIDs$, $oldIDs \cap newIDs$, $newIDs \setminus oldIDs$, l, k); // Vedi Algoritmo 4.10

Algoritmo 4.20 Scioglimento di una comunità: indicizzazione con funzioni LSH

Input: il descrittore della comunità da eliminare D, l'insieme S di funzioni LSH da impiegare per il calcolo degli identificatori.

1: $IDs \leftarrow \emptyset$;
2: **for all** $LSHFunction$ in S **do**
3: IDs.Add($LSHFunction(D.Attributes)$); // Associa al profilo un identificatore per ciascuna funzione LSH
4: **end for**
5: **for all** id in IDs **do**
6: ProfilesDHT.Remove(id, Hash(D); // Rimuove D dalla DHT dei profili.
7: **end for**
8: AddressesDHT.Remove($D.ID$); // Rimuove la lista degli indirizzi dei peer di accesso.

gono generando un volume di traffico $O(|U|^2 \cdot \log(X))$.

Ogni peer interrogato risponderà inviando i profili corrispondenti ai propri k risultati più significativi, con un messaggio di dimensione $O(k \cdot |C|)$, per un traffico totale $O(k \cdot |C| \cdot |U|)$.

La costruzione dell'indice distribuito comporta la memorizzazione, per ogni comunità, di una copia del profilo per ciascuno dei suoi elementi, quindi la quantità spazio occupata dall'indice è $O(|C|^2 \cdot Com)$.

In seguito all'aggiornamento del profilo di una comunità vengono inviate $|C|$ copie del nuovo vettore ai peer responsabili dei singoli attributi, oltre a R notifiche ai peer che memorizzano gli attributi rimossi.

Il traffico necessario all'aggiornamento dell'indice è quindi $O(|C|^2 \cdot \log(X) + R \cdot \log(X))$, mentre i messaggi di risposta hanno una dimensione totale $O(|C|^2 \cdot s)$, dove s è il numero di descrittori di comunità richiesti a ciascun peer.

La memorizzazione di un nuovo profilo di comunità comporta l'invio di $|C|$ copie del profilo ai peer responsabili dei relativi attributi, per un traffico totale $O(|C|^2 \cdot \log(X))$. Anche in questo caso il traffico generato dai messaggi di risposta è $O(|C|^2 \cdot s)$.

La notifica dell'esistenza del nuovo profilo alle l comunità più simili ha un costo $O(l \cdot |A| \cdot |C| \cdot \log(X))$.

La richiesta di eliminazione di un descrittore di comunità dall'indice comporta l'invio di $|C|$ messaggi, per un traffico complessivo $O(|C| \cdot \log(X))$;

4.2.2.4 Complessità: indicizzazione tramite funzioni LSH

Per la ricerca delle comunità viene eseguita una query per ognuno degli n identificatori calcolati per il profilo utente, il numero di messaggi inviati è quindi costante.

Ciascun messaggio di interrogazione contiene il profilo dell'utente, quindi la quantità di traffico generata è $O(n \cdot |U| \cdot \log(X))$.

Ogni query prevede un messaggio di risposta di dimensione $O(k \cdot |C|)$, il traffico totale è quindi $O(n \cdot k \cdot |C|)$.

Poiché viene memorizzata una copia del profilo per ciascun identificatore associato ad ogni comunità, la quantità di spazio occupata dall'indice distribuito è $O(n \cdot |C| \cdot Com)$.

In seguito all'aggiornamento di un profilo di comunità vengono memorizzate sulla DHT n copie del nuovo vettore e, al caso pessimo, vengono effettuate n richieste di eliminazione del vecchio profilo.

In realtà, per le proprietà delle funzioni LSH, il numero di identificatori diversi per le due versioni del profilo dovrebbe essere inferiore a n.

Il volume di traffico dovuto all'aggiornamento di ciascun profilo di comunità sarà quindi $O(n \cdot |C| \cdot \log(X) + n \cdot \log(X))$ per quanto riguarda le richieste di memorizzazione e $O(n \cdot |C| \cdot s)$ per i messaggi di risposta.

La cancellazione di un descrittore di comunità prevede l'invio di n richieste, che generano un traffico $O(n \cdot \log(X))$.

4.2.3 Applicazione dei metodi di indicizzazione a matrici di adiacenza tra oggetti

Anche in questo caso, per ragioni analoghe a quelle esposte in precedenza, le chiavi associate ai descrittori vengono calcolate in base ai valori degli attributi, senza considerare i pesi associati.

Questi, come le informazioni relative alle correlazioni tra gli oggetti, vengono impiegati successivamente per la *quantificazione* della similarità tra profili in base alla metrica 4.2.

In figura 4.14 è illustrata la generazione delle chiavi in base ai singoli attributi, mentre in figura 4.15 è illustrata quella basata sull'uso di $n \times 3$ funzioni LSH.

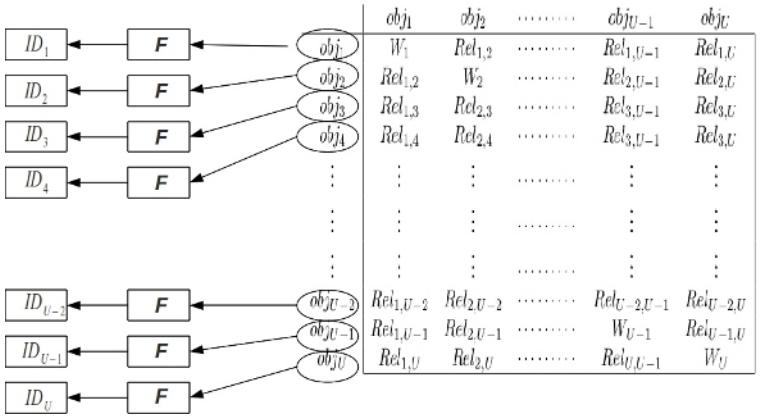


Figura 4.14: Matrice di adiacenza: indicizzazione per singoli attrbuti

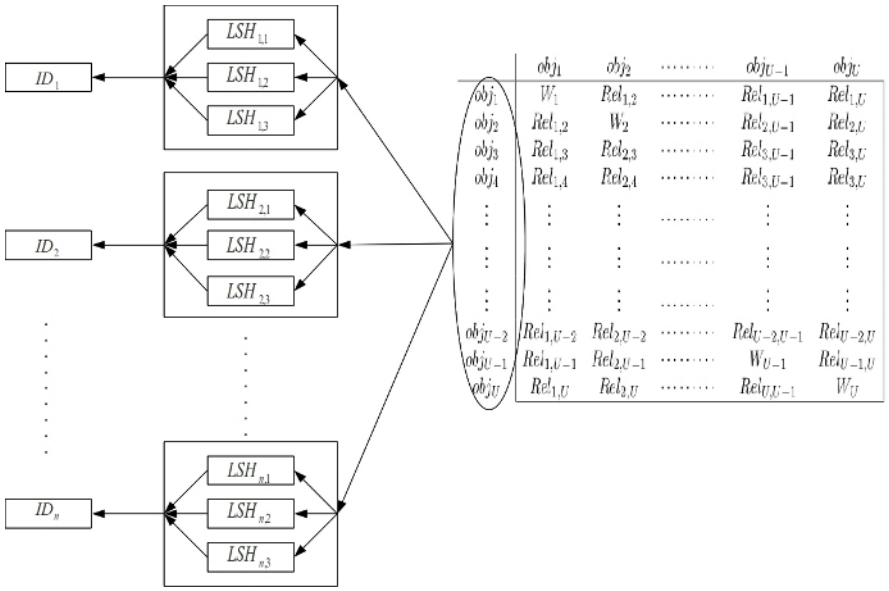


Figura 4.15: Matrice di adiacenza: indicizzazione con funzioni LSH

Le operazioni di aggiornamento dell'indice delle comunità e di ricerca nello stesso dei profili delle comunità affini agli interessi degli utenti si svolgono come nel caso in cui i profili sono costituiti da vettori n-dimensionali.

Riguardo all'eliminazione dei descrittori dall'indice, o solamente da alcuni nodi, valgono le considerazioni espresse in precedenza.

Il calcolo del valore di similarità tra due profili in base alla metrica 4.2 è illustrato dall'algoritmo 4.21.

Algoritmo 4.21 CalculateSimilarity, metrica 4.2

Input: le matrici di correlazione A e B
Output: il valore *sim* della similarità tra A e B

```
1:  globalSim ← 0;
2:  for all a in A do
3:    if a ∈ B then
4:      rowSim ← 0;
5:      rowWeight ← min(A.getWeight(a), B.GetWeight(a)); // Il peso
        associato ai valori di correlazione di a comuni ai due profili è il
        minimo dei pesi di a nei due profili
6:      globalSim ← globalSim + rowWeight;
7:      nRelations ← 0;
8:      for all obj in A do
9:        if obj ≠ a ∧ obj ∈ B then
10:         rowSim ← rowSim+min(A.GetRelation(a,obj), B.GetRelation(a,obj));
11:         nRelations ← nRelations + 1;
12:       end if
13:     end for
14:     if nRelations ≠ 0 then
15:       rowSim ← rowSim ÷ nRelations;
16:       globalSim ← globalSim + rowWeight × rowSim;
17:     end if
18:   end if
19: end for
20: globalSim ← globalSim ÷ max(A.Size,B.Size);
21: return globalSim;
```

4.2.3.1 Complessità: indicizzazione per singoli attributi

Poiché le matrici che definiscono i profili sono simmetriche la dimensione media dei profili sarà pari a $\frac{|U|^2}{2}$.

Per effettuare una ricerca per similarità per il profilo di un utente vengono inviati sulla DHT $|U|$ messaggi di interrogazione, ciascuno dei quali contiene i $\frac{|U|^2}{2}$ elementi del profilo, per un traffico totale di dimensione $O(\frac{|U^3|}{2} \cdot \log(X))$.

Ogni messaggio di risposta, che contiene k profili, è di dimensione $O(k \cdot \frac{|C|^2}{2})$, la dimensione totale dei risultati è quindi $O(k \cdot \frac{|C|^2}{2} \cdot |U|)$.

L'indice distribuito memorizza, per ciascuna comunità, una copia del profilo per ogni oggetto che lo compone, occupando quindi uno spazio $O(\frac{|C|^3}{2} \cdot Com)$.

L'aggiornamento dell'indice in seguito alla modifica di un profilo richiede l'invio di $|C|$ messaggi di notifica e R richieste di rimozione, con un traffico totale di dimensione $O(\frac{|C|^3}{2} \cdot \log(X) + R \cdot \log(X))$.

I messaggi di risposta alla notifica di un nuovo profilo hanno un peso complessivo $O(s \cdot \frac{|C|^3}{2})$, mentre l'invio dello stesso alle l comunità in assoluto più simili genera una quantità di traffico $O(l \cdot |A| \cdot \frac{|C|^2}{2} \cdot \log(X))$.

4.2.3.2 Complessità: indicizzazione tramite funzioni LSH

Impiegando l'approccio LSH il numero di interrogazioni effettuate per ciascuna ricerca da parte degli utenti si riduce a un valore n costante, per un traffico totale $O(n \cdot \frac{|U|^2}{2} \cdot \log(X))$.

Ognuno dei peer interrogati invia un massimo di k profili di comunità al richiedente, quindi la dimensione complessiva dei messaggi di risposta è $O(n \cdot k \cdot \frac{|C|^2}{2})$.

Lo spazio occupato dall'indice distribuito, che memorizza una copia del profilo di ogni comunità per ciascuno dei suoi identificatori, è $O(n \cdot \frac{|C|^2}{2} \cdot Com)$.

L'aggiornamento dell'indice comporta l'invio del nuovo profilo ad al più n nodi della DHT e di al più n richieste di rimozione del vecchio profilo, per un traffico complessivo $O(n \cdot \frac{|C|^2}{2} \cdot \log(X) + n \cdot \log(X))$.

I messaggi di risposta, al più s per ogni nodo che memorizza il nuovo profilo, hanno una dimensione totale $O(n \cdot s \cdot \frac{|C|^2}{2})$.

4.2.4 Applicazione dei metodi di indicizzazione alle componenti connesse di matrici di adiacenza tra oggetti

Indicizzando i profili delle comunità in base ai singoli attributi, la ricerca da parte dell'utente delle comunità in cui inserirsi si svolge, per ciascun *interesse*, come nel caso precedentemente esposto per un vettore n-dimensionale di oggetti.

L'indicizzazione tramite funzioni LSH viene effettuata associando un gruppo di identificatori semantici a ciascun cluster di oggetti contenuto nel profilo dell'utente, dato che il profilo di ogni comunita modella un singolo interesse.

La ricerca delle corrispondenze per ciascun interesse prosegue anche in questo caso come descritto in precedenza.

In entrambi i casi le operazioni di aggiornamento dell'indice si svolgono, per ciascun interesse dell'utente, come nel caso in cui i profili sono definiti da semplici vettori di oggetti.

Gli algoritmi eseguiti sono il 4.22 o il 4.23 a seconda del metodo di indicizzazione impiegato.

Algoritmo 4.22 Ingresso di un utente nel sistema: indicizzazione per singoli attributi, interessi multipli

---

Input: il profilo P dell'utente, il numero massimo k di profili simili richiesti a ciascun peer interrogato.
Output: il descrittore della comunità raccomandata e gli indirizzi dei relativi peer di accesso.

1: **for all** I in P **do**
2: // Per ogni *interesse* I dell'utente
3: $IDs \leftarrow \emptyset$;
4: **for all** *obj* in $I.Attributes$ **do**
5: IDs.Add(Hash(*obj*)); // Associa all'interesse un identificatore per ciascun attributo
6: **end for**
7: $retval[I] \leftarrow$ GetSimilarCommunitiesData(I,IDs,k); // Vedi algoritmo 4.6
8: **end for**
9: **return** *retval*;

Algoritmo 4.23 Ingresso di un utente nel sistema: indicizzazione con funzioni LSH, interessi multipli

Input: il profilo P dell'utente, il numero massimo k di profili simili richiesti a ciascun peer interrogato, l'insieme S di funzioni LSH da impiegare per il calcolo degli identificatori.
Output: il descrittore della comunità raccomandata e gli indirizzi dei relativi peer di accesso.

1: **for all** I in P **do**
2: // Per ogni *interesse* I dell'utente
3: $IDs \leftarrow \emptyset$;
4: **for all** $LSHFunction$ in S **do**
5: IDs.Add($LSHFunction(I.Attributes)$); // Associa al profilo un identificatore per ciascuna funzione LSH
6: **end for**
7: $retval[I] \leftarrow \langle selectedCommunity, addresses \rangle$;
8: **end for**
9: **return** $retval$;

4.2.4.1 Complessità

Impiegando il metodo di base, per la ricerca delle comunità corrispondenti ad un profilo utente è necessario effettuare $\sum_m |I_m|$ query, ciascuna delle quali riguarderà $|I_m|$ oggetti, dove $|I_m| \leq |U|$ è il numero di oggetti che descrivono l'm-esimo *interesse* dell'utente e l'uguaglianza vale solo nel caso in cui tutti gli attributi del profilo appartengano ad un'unica componente connessa, anziché l'intero profilo, generando una quantità di traffico $O((\sum_m |I_m|)^2 \cdot \log(X))$.

Se all'algoritmo di estrazione delle componenti connesse vengono applicate politiche di potatura degli archi e\o di eliminazione delle componenti connesse più piccole, si avrà probabilmente $\sum_m |I_m| < |U|$.

Ciascun peer interrogato restituisce, nel proprio messaggio di risposta, i k profili di comunità più simili all'interesse dato, per un traffico complessivo $O(k \cdot |C| \cdot \sum_m |I_m|)$.

La dimensione dell'indice in questo caso è $O(|C^2| \cdot Com)$.

Effettuando l'indicizzazione con una funzione LSH la ricerca delle comunità comporterà invece l'esecuzione di $n \cdot Int$ interrogazioni sulla DHT, dove n è il numero di identificatori associati a ciascun cluster di oggetti e Int il numero di interessi di un utente generando un traffico $O(n \cdot Int \cdot \sum_m |I_m| \cdot \log(X))$, anche in questo caso ogni messaggio di risposta avrà dimensione $O(|C|)$ per un traffico complessivo $O(k \cdot n \cdot Int \cdot |C|)$.

Ciascuna comunità memorizzerà nell'indice distribuito una copia del proprio profilo per ciascun identificatore semantico ad esso associato, con un'occupazione totale di spazio $O(n \cdot |C| \cdot Com)$.

Utilizzare il metodo basato su funzioni LSH può risultare sconveniente nel caso in cui i peer abbiano molti interessi, ciascuno identificato da un numero ristretto di oggetti, oppure, come negli altri casi, per profili di dimensioni molto ridotte.

4.3 Riepilogo

Per modellare gli interessi degli utenti vengono proposti tre tipi di profili, costruiti a partire da collezioni di oggetti detti *attributi*. I tre tipi di profilo definiti sono:

- *Vettori di oggetti pesati*: collezioni di attributi nelle quali a ciascuno di essi è associato un *peso* numerico che ne indica l'importanza.

- *Matrici di adiacenza tra attributi*, che arricchiscono le informazioni fornite dai vettori di oggetti pesati con dei valori che stimano la *correlazione* tra gli attributi. Le matrici di questo tipo sono quadrate e simmetriche.

- *Insiemi di componenti connesse estratte da matrici di adiacenza*, ottenuti dall'applicazione alle matrici di adiacenza di un algoritmo che, visitandole come grafi, effettua una potatura degli archi di peso inferiore a un minimo fissato per isolare insiemi di attributi particolarmente correlati. Da ciascuna matrice possono essere estratte più componenti, ciascuna delle quali individua un distinto *interesse* dell'utente.

Ciascuno di questi tipi di profilo può essere mappato sulla DHT utilizzando due diversi metodi di indicizzazione:

- *Indicizzazione dei singoli attributi*, che consente di effettuare ricerche *esatte*, ma comporta consumi elevati di banda e spazio di memorizzazione.

- *Indicizzazione tramite funzioni LSH*, che riduce drasticamente i costi, ma è un metodo *probabilistico*.

Nella tabella 4.1 sono messi a confronto i costi determinati dall'impiego dei due metodi di indicizzazione per profili definiti come vettori di oggetti pesati, nella tabella 4.2 quelli relativi ai profili definiti come matrici di adiacenza e nella tabella 4.3 quelli relativi ai profili definiti come insiemi di componenti connesse estratte dalle matrici.

Con $|U|$ e $|C|$ sono indicati relativamente il numero di attributi del profilo di ciascun utente e di ciascuna comunità, con $|A|$ il numero di punti di accesso di ciascuna comunità, con X il numero di nodi della DHT, con Com il numero di comunità esistenti, con k il numero massimo di profili richiesti a ciascun nodo in risposta a un interrogazione per similarità, con s il numero massimo di profili di comunità richiest in risposta a una richiesta di memorizzazione di un nuovo profilo, con R il numero di attributi rimossi da un profilo in seguito al suo aggiornamento.

Nel caso dell'indicizzazione con funzioni LSH n indica il numero, costante, degli identificatori associati a ciascun profilo.

Nel caso dei profili definiti come insiemi di componenti connesse $|I_m|$ indica il numero di attributi che descrivono l'm-esimo *interesse* dell'utente e si ha $\sum_m |I_m| \leq |U|$, mentre Int indica il numero di interessi di un utente.

	Singoli attributi	*LSH*
Interrogazione	$O(\|U\|^2 \cdot \log(X))$	$O(n \cdot \|U\| \cdot \log(X))$
Risposte interrogazione	$O(k \cdot \|C\| \cdot \|U\|)$	$O(k \cdot \|C\| \cdot n)$
Aggiunta comunità	$O(\|C\|^2 \cdot \log(X))$	$O(n \cdot \|C\| \cdot \log(X))$
Aggiornamento profilo	$O(\|C\|^2 \cdot \log(X) + R \cdot \log(X))$	$O(n \cdot \|C\| \cdot \log(X) + R \cdot \log(X))$
Risposte nuovo profilo	$O(\|C\|^2 \cdot s)$	$O(n \cdot \|C\| \cdot s)$
Cancellazione descrittore	$O(\|C\| \cdot \log(X))$	$O(n \cdot \log(X))$
Dimensione indice	$O(\|C\|^2 \cdot Com)$	$O(n \cdot \|C\| \cdot Com)$

Tabella 4.1: Costi dei metodi di indicizzazione: vettori pesati.

	Singoli attributi	*LSH*
Interrogazione	$O(\frac{\|U^3\|}{2} \cdot \log(X))$	$O(n \cdot \frac{\|U^2\|}{2} \cdot \log(X))$
Risposte interrogazione	$O(k \cdot \frac{\|C\|^2}{2} \cdot \|U\|)$	$O(k \cdot \frac{\|C\|^2}{2} \cdot n)$
Aggiunta comunità	$O(\frac{\|C\|^3}{2} \cdot \log(X))$	$O(n \cdot \frac{\|C\|^2}{2} \cdot \log(X))$
Aggiornamento profilo	$O(\frac{\|C\|^3}{2} \cdot \log(X) + R \cdot \log(X))$	$O(n \cdot \frac{\|C\|^2}{2} \cdot \log(X) + R \cdot \log(X))$
Risposte nuovo profilo	$O(\frac{\|C\|^3}{2} \cdot s)$	$O(n \cdot \frac{\|C\|^2}{2} \cdot s)$
Cancellazione descrittore	$O(\|C\| \cdot \log(X))$	$O(n \cdot \log(X))$
Dimensione indice	$O(\frac{\|C\|^3}{2} \cdot Com)$	$O(n \cdot \frac{\|C\|^2}{2} \cdot Com)$

Tabella 4.2: Costi dei metodi di indicizzazione: matrici di adiacenza.

	Singoli attributi	*LSH*
Interrogazione	$O((\sum_m \|I_m\|)^2 \cdot \log(X))$	$O(n \cdot Int \cdot \sum_m \|I_m\| \cdot \log(X))$
Risposte interrogazione	$O(k \cdot \|C\| \cdot (\sum_m \|I_m\|)$	$O(k \cdot n \cdot Int \cdot \|C\|)$
Aggiunta comunità	$O(\|C\|^2 \cdot \log(X))$	$O(n \cdot \|C\| \cdot \log(X))$
Aggiornamento profilo	$O(\|C\|^2 \cdot \log(X) + R \cdot \log(X))$	$O(n \cdot \|C\| \cdot \log(X) + R \cdot \log(X))$
Risposte nuovo profilo	$O(\|C\|^2 \cdot s)$	$O(n \cdot \|C\| \cdot s)$
Cancellazione descrittore	$O(\|C\| \cdot \log(X))$	$O(n \cdot \log(X))$
Dimensione indice	$O(\|C\|^2 \cdot Com)$	$O(n \cdot \|C\| \cdot Com)$

Tabella 4.3: Costi dei metodi di indicizzazione: componenti connesse.

Capitolo 5

Sperimentazione

Nella prima sezione di questo capitolo sezione viene descritto il framework Overlay Weaver [3], utilizzato per l'implementazione dell'indice distribuito e per la sperimentazione in ambiente distribuito.

Nella seconda sezione vengono descritti il dataset impiegato per la sperimentazione e le procedure impiegate per estrarne i profili degli utenti.

Nella terza sezione vengono illustrati i test effettuati in ambiente centralizzato per validare l'implementazione dell'algoritmo per il calcolo approssimato delle *min-wise independent permutations* e per confrontare la capacità delle funzioni LSH da esse derivate di individuare gruppi di oggetti simili con quella dell'algoritmo di clustering K-Means [7], utilizzato per simulare la creazione di comunità di utenti in base ai loro profili.

Nella quarta sezione vengono illustrati gli esperimenti effettuati per verificare la qualità dei risultati ottenibili utilizzando l'indicizzazione tramite funzioni LSH per la ricerca per similarità dei vari tipi di profili su DHT e viene analizzata la distribuzione del carico sulla DHT stessa.

Nella quinta sezione viene fornita una stima dei costi, in termini di consumo di banda e di spazio di memorizzazione, indotti dall'uso delle diverse modalità di indicizzazione dei profili.

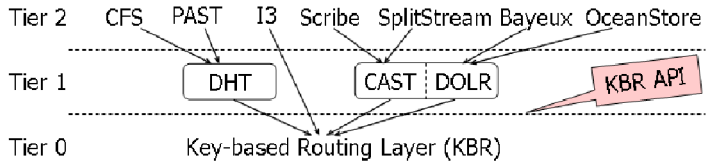


Figura 5.1: Key Based Routing: struttura a livelli.

5.1 Overlay Weaver

Overlay Weaver (OW) [3] è un framework Open Source per la costruzione di overlay network strutturati, scritto in Java e reperibile gratuitamente in rete, caratterizzato da una strutturazione a livelli e da un'implementazione fortemente modulare.

Oltre ad aderire al modello di key-based routing (KBR) proposto in [32], svincolando, come illustrato in figura 5.1 i servizi come la DHT o il multicast dal livello di routing, infatti, OW introduce un'ulteriore modularizzazione di tale livello, distinguendo le funzionalità di invio e ricezione dei messaggi dagli algoritmi di routing propriamente intesi. In questo modo vengono notevolmente facilitatate l'implementazione di nuovi meccanismi e algoritmi di routing, la modifica di quelli già esistenti, migliorando la flessibilità del framework senza complicare, grazie alla definizione di opportune interfacce, l'utilizzo delle funzionalità di routing da parte di applicazioni e servizi di livello superiore.

Queste caratteristiche, unite all'offerta di valide implementazioni di vari overlay strutturati e di un servizio di multicast, rendono OW un framework versatile e adatto tanto alla realizzazione di applicazioni basate su overlay network già esistenti quanto allo sviluppo di nuovi sistemi strutturati per il supporto di applicazioni distribuite.

Overlay Weaver mette inoltre a disposizione degli sviluppatori una serie di utili strumenti per effettuare test e debugging, tra i quali un emulatore in grado di simulare il funzionamento di un overlay network su una o più macchine fisiche ed un visualizzatore grafico della topologia della rete.

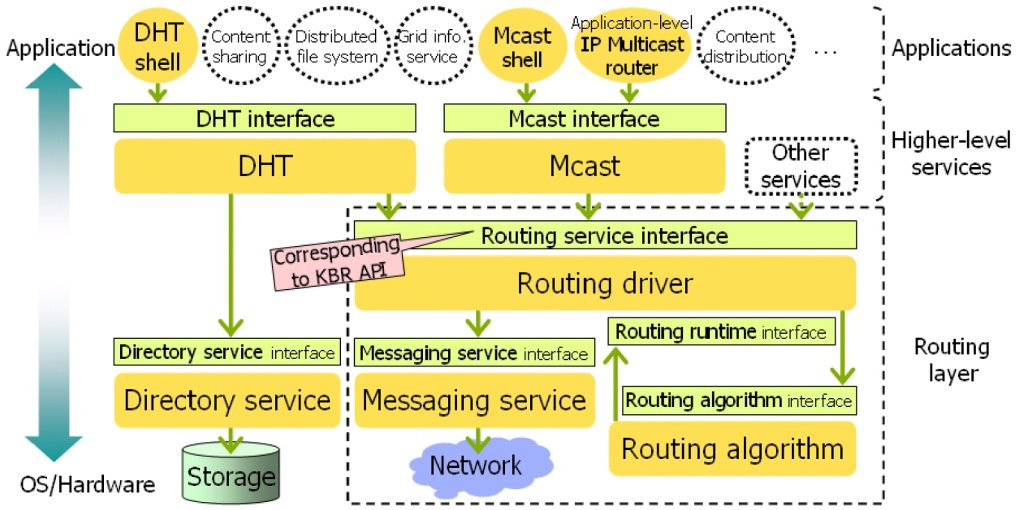


Figura 5.2: Architettura di Overlay Weaver.

5.1.1 Architettura del framework

Come già accennato, l'architettura di OverlayWeaver, illustrata in figura 5.2,è suddivisa nei livelli:

- Applicazioni.
- Servizi ad alto livello.
- Servizi di routing.
- Servizi di memorizzazione (directory).

In figura 5.2 appare evidente il sostanziale miglioramento introdotto da Overlay Weaver rispetto al modello KBR, ovvero la modularizzazione del livello di routing, suddiviso in:

- *Routing driver*: modulo che, in base alle informazioni ottenute dal modulo *routing algorithm*, gestisce il recapito delle richieste effettuate dai servizi di alto livello ad altri peer e dei relativi messaggi di risposta, avvalendosi dei servizi del modulo *messaging service* per l'invio e ricezione dei messaggi da e verso la rete.

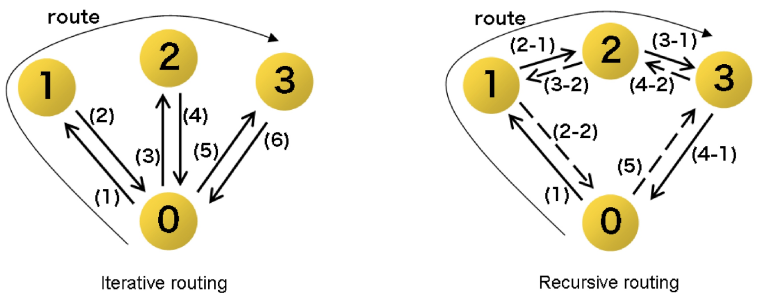


Figura 5.3: Routing iterativo e ricorsivo.

- *Routing algorithm*: modulo le cui istanze implementano gli algoritmi di routing che caratterizzano i diversi tipi di overlay network.

- *Messaging serivce*: modulo che si occupa dell'invio e la ricezione dei messaggi tra i nodi dell'overlay network, gestendo, in modo trasparente rispetto al routing driver, l'interazione con i protocolli TCP e UDP e le comunicazioni tra thread che simulano l'invio di messaggi in rete durante l'uso dell'emulatore. In Overlay Weaver sono implementate due versioni del routing driver, una iterativa ed una ricorsiva. La figura 5.3 illustra le modalità di comunicazione impiegate da ciascun tipo di routing.

Le linee che collegano i nodi e i numeri associati identificano rispettivamente i messaggi scambiati e il loro ordine temporale.

Il routing iterativo prevede che il nodo che sottomette una query o una richiesta di memorizzazione coordini l'intera operazione passo per passo, individuando in base all'algoritmo di routing il primo nodo a cui inoltrare la richiesta, ottenendo in risposta l'indicazione del successivo passo di routing e iterando il procedimento fino a raggiungere il nodo destinatario.

Nel caso del routing ricorsivo, invece, ciascun peer che riceve una richiesta la inoltra automaticamente al nodo successivo, senza coinvolgere il richiedente. In questo caso, le linee tratteggiate in figura indicano delle notifiche di avvenuta ricezione inviate in caso di utilizzo di protocolli non affidabili, come ad esempio UDP, che hanno comunque una priorità minore rispetto ai messaggi relativi alle interrogazioni.

Allo stato attuale in Overlay Weaver sono implementati diversi noti algoritmi di routing, tra cui Chord [8], Pastry [27], Tapestry [29], Kademlia [30]e Koorde [31].

Il modulo *Directory* di OW, utilizzato solamente dalla DHT, offre la possibilità di memorizzare i dati in un database relazionale (Berkeley DB [33] Java Edition) oppure in memoria principale attraverso le tabelle hash standard fornite da Java. In ciascun caso è possibile scegliere se permettere o meno l'associazione di più valori alla stessa chiave e se abilitare o meno un meccanismo di scadenza dei dati.

Il disaccoppiamento dei vari moduli tramite l'uso di opportune interfacce permette un'elevata flessibilità a livello di implementazione dei servizi e di combinazione delle stesse implementazioni al fine di individuare la configurazione che meglio si adatta a ciascun caso d'uso.

La stessa concezione modulare si ritrova, scendendo più in dettaglio, nell'implementazione di ciascuno dei moduli descritti.

5.1.2 Le DHT di Overlay Weaver: utilizzo ed estensione

Overlay Weaver fornisce agli sviluppatori i servizi di DHT, tramite le classi **BasicDHTImpl** e **ChurnTolerantDHTImpl** del package **ow.dht.impl**. Entrambe le classi implementano, indipendentemente dal livello di routing sottostante, l'interfaccia **ow.dht.DHT**, differenziandosi per il fatto che la seconda impiega, in modo trasparente rispetto all'utilizzatore, una politica di moderata replicazione dei contenuti al fine di attenuare gli effetti di eventuali fallimenti dei nodi.

L'interfaccia **DHT** espone i metodi di base necessari per l'ingresso e la disconnessione di un nodo della tabella distribuita e per la memorizzazione, la ricerca esatta e la rimozione di un valore, oltre a quello che consentono di accedere, tramite un oggetto che implementa l'interfaccia **ow.dht.DHTConfiguration**, alle configurazioni dei servizi di livello inferiore e di impostare i parametri come il tipo di routing driver, l'algoritmo di routing, il tipo di servizio di memorizzazione utilizzato il protocollo da impiegare per lo scambio di messaggi sulla rete.

Le applicazioni in esecuzione sulla stessa macchina che ospita un nodo della DHT possono accedere alle sue funzionalità tramite semplici invocazioni di metodo mentre, per quanto riguarda le applicazioni remote, la shell DHT fornita da Overlay Weaver accetta richieste di memorizzazione, ricerca e rimozione di valori in formato XML-RPC in accordo alla specifica di OpenDHT [34].

Le funzionalità della DHT possono essere modificate o estese reimplementando i metodi esposti dall'interfaccia, lasciandone inalterata la firma, come pure estendendo l'interfaccia stessa e una o più sue implementazioni con l'aggiunta di ulteriori metodi.

In entrambi i casi può essere necessario introdurre nuovi tipi di messaggi per permettere ai peer di effettuare le richieste relative alle funzionalità aggiunte o modificate e di ricevere le relative risposte.

Secondo l'interfaccia **ow.messaging.Message**, che tutti i tipi di messaggio devono implementare, un messaggio presenta tre campi fondamentali:

- La coppia $\langle ID$, *indirizzo IP*$\rangle$del mittente.
- Un numero intero, detto *TAG* che identifica univocamente il tipo del messaggio.
- Un contenuto, costituito da un array di oggetti di tipo *java.io.Serializable* di dimensione arbitraria tra 0 e 255.

La gestione dei tag è resa trasparente all'utente delegando la creazione effettiva degli oggetti di tipo Message alla classe **ow.dht.impl.DHTMessageFactory**, che, per ciascun tipo di messaggio, espone un metodo che, presi in input i dati del mittente e il contenuto, restituisce l'oggetto Message contraddistinto dal tag opportuno.

Per definire un nuovo tipo di messaggio è necessario estendere la classe **ow.messaging.Tag**, che mantiene in una tabella statica l'insieme dei tag di tutti i tipi di messaggio associati ai loro nomi simbolici, per assegnare una coppia $\langle TAG\ intero, nome\ simbolico\rangle$al nuovo messaggio, quindi estendere la classe DHTMessaageFactory aggiungendo il metodo per la creazione di

messaggi del nuovo tipo. In alcune versioni di OW è necessario modificare la classe Tag anziché estenderla dal momento che questa è dichiarata *final*.

La gestione delle richieste ricevute da ogni nodo è gestita dal Routing Service tramite una serie di handler (uno per ciascun tipo di messaggio) che implementano l'interfaccia **ow.messaging.MessageHandler**. La gestione del messaggio da parte dell'handler, che prevede sempre un messaggio di risposta, deve essere implementata dal metodo *Message process(Message msg)*, esposto dall'interfaccia.

Una volta definito un nuovo messaggio è quindi necessario implementare il relativo handler e registrare quest'ultimo presso il Routing Service tramite il metodo *void addHandler(int tag, MessageHandler handler)*, che deve essere invocato al momento dell'inizializzazione dell'oggetto DHT.

5.1.3 Tool di sviluppo

Lo strumento principale fornito da Overlay Weaver per lo sviluppo e il testing delle applicazioni è l'*emulatoredi rete*, che permette di effettuare test in ambiente virtuale. All'interno dell'emulatore ciascun nodo della DHT viene eseguito come un thread ed è possibile condurre i test su una sola macchina oppure distribuire gruppi di nodi su più elaboratori collegati in rete, affidando al livello di messaging la gestione delle comunicazioni tra i thread e tra le diverse macchine, che viene effettuata in modo trasparente rispetto ai livelli superiori.

L'interazione con l'emulatore può avvenire in due modalità, una, detta *interattiva*, nella quale i comandi vengono impartiti dall'utente attraverso una shell, l'altra, detta *batch*, in cui l'emulatore legge i comandi e la loro scansione temporale da un apposito file detto *scenario*.

La modalità più interessante è senza dubbio quella batch, dal momento che permette la creazione e la gestione di reti di notevoli dimensioni.

In figura 5.4 è mostrato un esempio di file scenario, nel quale si può notare come sia possibile far schedulare all'emulatore l'esecuzione di sequenze di comandi definendone l'istante di inizio e la periodicità.

```
# Invokes an Overlay Visualizer and 20 instances of DHT shell

timeoffset 2000

# invokes an Ovelray Visualizer
class ow.tool.visualizer.Main
schedule 0 invoke

# invokes the 1st DHT shell
class ow.tool.dhtshell.Main
#class ow.tool.mcastshell.Main
arg -m emu0
schedule 6000 invoke

# invokes 19 DHT shells
arg -m emu0 emu1
schedule 6000,1000,19 invoke

# emu1, emu13, emu6 and emu12 periodically keep querying
schedule 25000,4000 control 1 get mno
schedule 26000,4000 control 13 get yyy
schedule 27000,4000 control 6 get def
schedule 28000,4000 control 12 get ddd
```

Figura 5.4: Overlay Weaver: esempio di file scenario.

La creazione di uno scenario può essere assistita da un apposito tool presente nel framework. L'ultimo strumento fornito dal framework, è un visualizzatore grafico di rete che consente di rappresentare, in varie modalità grafiche, la struttura dell'overlay network in ogni istante e di visualizzare in tempo reale le comunicazioni tra i nodi, distinguendo graficamente le varie tipologie di messaggi scambiati. Un esempio di utilizzo del visualizzatore è mostrato in figura 5.5.

5.2 Il dataset PubMed e la creazione dei profili

Il dataset impiegato per la sperimentazione è costituito da una lista di articoli di letteratura medica consultati da utenti di PubMed[4], dove ogni pubblicazione è individuata da titolo, *keywords* e *abstract*.

Il dataset contiene informazioni relative a 5177 utenti e 211878 documenti, con una media di 48,8 documenti consultati per ciascun utente.

I documenti contengono in tutto 127492 termini distinti che sono stati filtrati eliminando le *stopword* (termini particolarmente frequenti, ma non

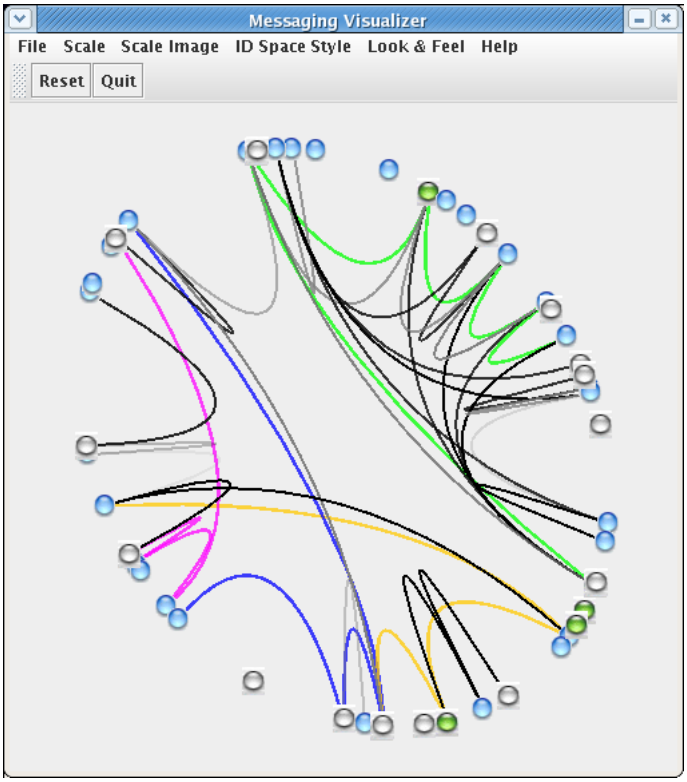


Figura 5.5: Overlay Weaver: visualizzatore di rete.

significativi, ad esempio articoli, congiunzioni, pronomi), ottenendo 127075 termini utili dai quali, tramite un procedimento di *stemming*, consistente nell'estrazione della *radice* di ciascun termine (ad esempio, alle parole *waited, waits* e *waiting* corrisponderà la sola radice *wait*), sono state estratte 107257 radici.

Per questo scopo sono stati utilizzate la lista di *stopword* e l'implementazione del "Porter Stemmer"[5] fornite dal motore di ricerca open source Terrier[6].

Nel seguito con l'espressione "termini" verranno indicate, per semplicità, le radici risultanti dallo *stemming* dei dati di partenza.

Per ciascun documento è stata eseguita una pesatura dei termini ad esso associati, attribuendo una rilevanza maggiore a quelli che occorrono tra le *keyword* rispetto a quelli contenuti solamente nell'*abstract*, e a questi ultimi rispetto ai termini che occorrono unicamente nei titoli.

La lista dei termini associata a ciascun utente è stata estratta dalla collezione di documenti consultati da tale utente, ad ogni termine è stato attribuito un peso pari alla media dei suoi pesi all'interno di tale collezione.

Per costruire i profili di tipo vettore n-dimensionale degli utenti sono stati selezionati i termini più significativi da ciascuna di tali liste.

A ciascun ciascun utente è stata quindi associata un profilo costituito da una matrice di adiacenza i cui elementi sono stati determinati individuando, all'interno dell' insieme di documenti da questi consultati, le co-occorrenze dei termini contenuti nel suo profilo unidimensionale.

La correlazione tra ciascuna coppia di termini t_a,t_b è stata valutata come

$$Rel_{a,b} = \frac{N_{a,b} \cdot avg(W_a, W_b)}{max(N_a, N_b)}$$

dove $N_{a,b}$ è il numero di co-occorrenze dei termini, $avg(W_a, W_b)$ la media dei pesi ad essi associati, N_a ed N_b il numero di occorrenze di ciascun termine. I pesi associati ai singoli termini sono rimasti invariati rispetto al caso dei vettori di termini pesati.

Da queste matrici di co-occorrenza è infine estratta l'ultima serie di profili, costruiti estraendone le componenti connesse applicando una politica di

potatura delle relazioni di peso inferiore ad una soglia fissata $minFreq$ e selezionando quindi, tra le componenti ottenute, quelle di dimensione maggiore o uguale a un valore fissato $minSize$.

Ogni matrice, vista come un grafo, è stata visitata in profondità a partire da ciascun nodo, percorrendo solamente gli archi di peso superiore a $minFreq$ ed unendo nella stessa componente connessa tutti i nodi raggiunti.

I nodi a partire dai quali era già stata effettuata la visita del grafo non venivano nuovamente visitati, dato che il grafo non era diretto e che le matrici erano simmetriche.Anche in questo caso il peso associato ad ogni termine non è cambiato al caso dei vettori.

Dal dataset sono state estratte due serie di profili, la prima, relativa agli utenti ai quali sono stati associati almeno 300 termini pesati, costituita da 2515 elementi, la seconda, relativa agli utenti ai quali sono stati associati almeno 150 termini pesati, costituita da 2754 elementi.

Le componenti connesse sono state estratte dalle matrici definite a partire da 300 termini. Relativamente a questo tipo di profilo sono disponibili dei risultati preliminari relativi al consumo di banda e spazio di memorizzazione per i due metodi di indicizzazione. Per quanto riguarda la correttezza dei risultati ottenuti dalle interrogazioni per similarità, una valida indicazione sul comportamento da apettarsi per questo tipo di profili viene dai test relativi ai vettori di termini pesati, dal momento che, ai fini della ricerca per similarità, ciascuna componente connessa è un vettore pesato a sé stante.

5.3 Validazione dell'implementazione delle funzioni LSH

Per validare l'implementazione delle *min-wise independent permutations* realizzata in questa tesi e verificare la possibilità dell'impiego del metodo di indicizzazione basato su funzioni *LSH* sono stati eseguiti, in ambiente centralizzato e per diverse combinazioni dei parametri n ed m che caratterizzano il metodo LSH, test riguardanti la probabilità di generare identificatori coincidenti per oggetti simili e la ricerca per similarità di profili di comunità. La

creazione delle comunità è stata simulata applicando l'algoritmo di clustering centralizzato K-Means [7] ai profili estratti dal dataset PubMed.

Gli esperimenti ed i risultati ottenuti sono esposti nel seguito di questa sezione.

5.3.1 Valutazione del mantenimento della località

Questo esperimento è stato eseguito su vettori costituiti da 150 termini pesati, allo scopo di verificare la bontà dell'approssimazione delle *min-wise independent permutations* impiegata.

Scelti casualmente 90 profili reali, dagli elementi di ciascuno di essi sono stati derivati, alterandone un numero progressivamente crescente di termini, 14 vettori di termini caratterizzati da una similarità di Jaccard decrescente (da circa 0,910 a circa 0.176) con gli elementi del profilo corrispondente.

Gli identificatori generati tramite funzioni *LSH* per i vettori di termini corrispondenti a ognuno dei profili originali sono stati confrontati con quelli associati ai corrispondenti vettori alterati allo scopo di calcolare, per ciascun valore di similarità s, le probabilità, riportate nei grafici da 5.6 a 5.11, di ottenere almeno una corrispondenza.

L'esperimento è stato ripetuto al variare del numero, n, di identificatori calcolati per ciascun vettore di termini e del numero, m, di funzioni *LSH* combinate per ottenere ciascun identificatore, confrontando la probabilità di collisione ottenuta con quella teorica, pari a $1 - (1 - s^m)^n$.

Le probabilità di collisione ottenute risultano molto vicine a quelle teoriche, validando quindi l'impiego dell'implementazione approssimata delle min-wise indepentent permutations in questo sistema.

5.3.2 Ricerca dei profili di comunità

Per verificare la validità del metodo di indicizzazione dei profili tramite identificatori derivati da funzioni *LSH* è stata simulata la ricerca per similarità di profili di comunità a partire dai profili costituiti da vettori di 150 termini pesati ciascuno, estratti dal dataset PubMed.

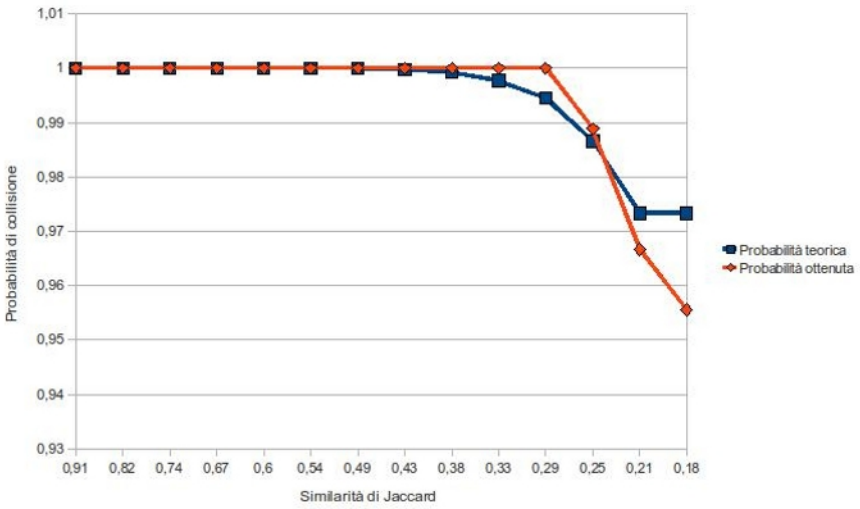


Figura 5.6: Confronto delle probabilità di collisione, n=15, m=1

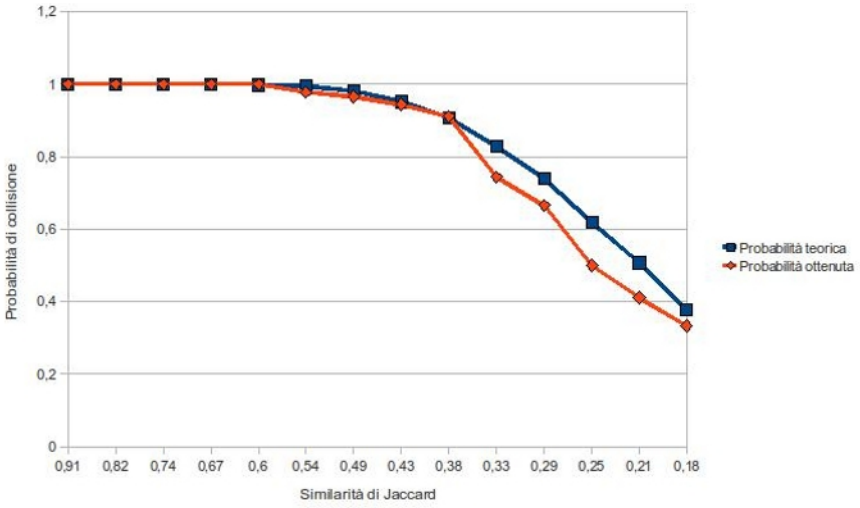


Figura 5.7: Confronto delle probabilità di collisione, n=15, m=2

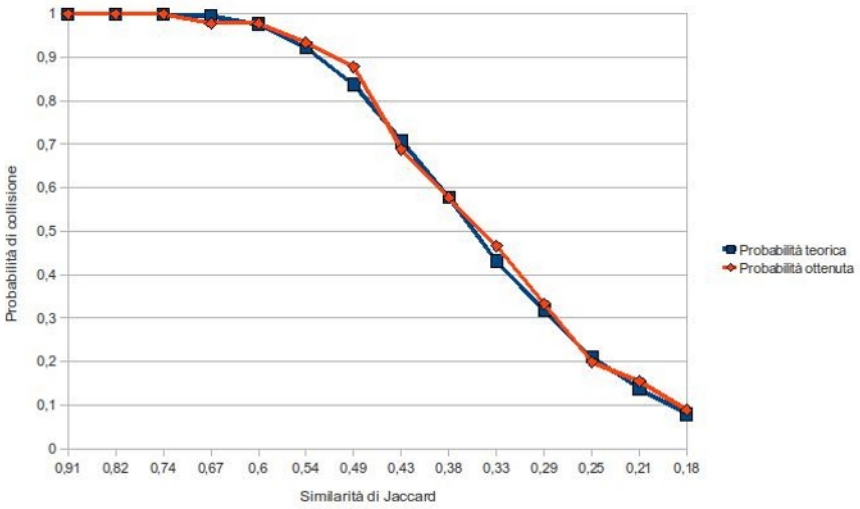


Figura 5.8: Confronto delle probabilità di collisione, n=15, m=3

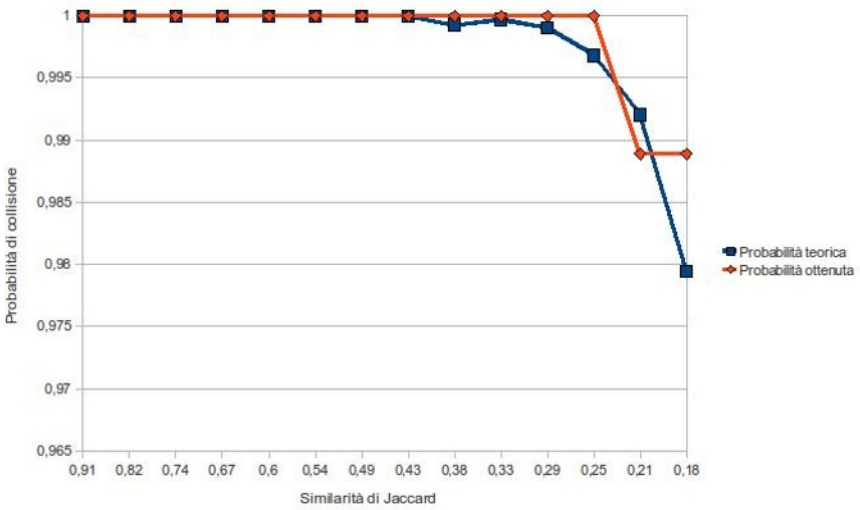


Figura 5.9: Confronto delle probabilità di collisione, n=20, m=1

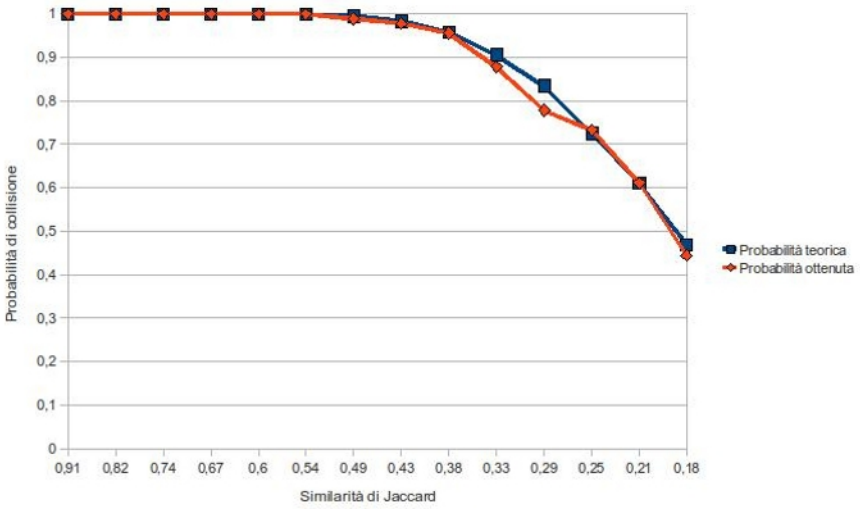


Figura 5.10: Confronto delle probabilità di collisione, n=20, m=2

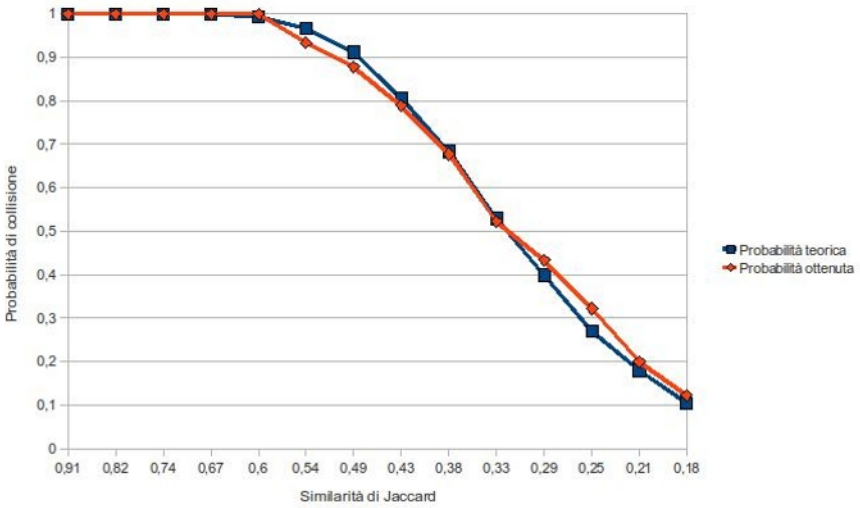


Figura 5.11: Confronto delle probabilità di collisione, n=20, m=3

I test sono stati eseguiti in ambiente centralizzato, utilizzando tabelle hash.

5.3.2.1 Simulazione della creazione di comunità tramite k-means

Per simulare la creazione di comunità di utenti è stato impiegato l'algoritmo di clustering centralizzato K-Means [7], basato sulla similarità di Jaccard tra i termini che costituiscono i profili degli utenti.

A causa della sparsità dei dati sono stati rilevati problemi di convergenza dell'algoritmo di clustering, pertanto è stato deciso di ripeterne l'esecuzione e di considerare come cluster significativi quelli che si mantengono invarianti rispetto a tale ripetizione.

Dopo varie esecuzioni di K-Means sono emersi diversi cluster, 7 dei quali, contenenti circa 1800 profili sui 2754 iniziali, presenti con elevata frequenza. Ciascuno di tali cluster è stato considerato come una comunità di utenti il cui profilo è stato identificato con quello del relativo *medoide*, considerato l'utente più rappresentativo del gruppo.

5.3.2.2 Ricerca dei medoidi per similarità

I 7 profili di comunità ottenuti dall'applicazione di K-Means sono stati memorizzati su una tabella hash calcolando per ognuno n identificatori ottenuti ciascuno dalla composizione di m funzioni LSH.

Successivamente, per ciascuno dei rimantenti profili sono state effettuate n ricerche sulla tabella hash impiegando identificatori calcolati tramite lo stesso insieme di funzioni utilizzato per indicizzare i profili delle comunità.

Per ciascun profilo si verifica se la ricerca produce risultati e se si ottiene almeno una volta il profilo del medoide del suo cluster di appartenenza e, in caso contrario, si calcola la distanza di Jaccard tra i profili ottenuti e quello desiderato.

L'esperimento è stato ripetuto al variare dei parametri n ed m, calcolando ogni volta il numero complessivo di profili per i quali non sono stati ottenuti risultati (indicato con $No - res$), il numero di profili per i quali non sono stati ottenuti i profili dei medoidi (indicato con $No - Leader$), la media e la

varianza della distanza dei profili ottenuti rispetto a quelli desiderati (indicati rispettivamente con *Media* e *Varianza*) e la percentuale di profili utente per i quali è stato ottenuto il profilo della comunità di appartenenza (indicata con *Successo*).

La similarità media dei profili con i relativi medoidi era circa 0.324.

Le tabelle 5.1 e 5.2 riportano i risultati ottenuti rispettivamente per $n = 15$ e per $n = 20$.

	$m = 1$	$m = 2$	$m = 3$
$No - res$	0	344	634
$No - Leader$	8	340	348
$Media$	0,06013	0,10188	0,11417
$Varianza$	0,00093	0,00581	0,00551
$Successo$	99,554%	61,87%	45,261%

Tabella 5.1: Ricercadei medoidi: risultati per n=15

	$m = 1$	$m = 2$	$m = 3$
$No - Res$	0	116	712
$No - Leader$	22	251	234
$Media$	0,05543	0,07145	0,08067
$Varianza$	0,00169	0,00275	0,00365
$Successo$	98,77%	79,54%	47,26%

Tabella 5.2: Ricerca dei medoidi: risultati per n=20

5.3.2.3 Ricerca dei medoidi per similarità con profili utente filtrati e profili di comunità casuali

I test di ricerca sono stati ripetuti impiegando come profili di comunità, oltre a quelli dei medoidi individuati da K-Means, un ulteriore insieme di 1500 profili generati casualmente.

A causa della sparsità dei dati, che comportava una bassa similarità dei profili con i medoidi dei rispettivi cluster, i profili impiegati per le interrogazioni sono stati filtrati in base alla loro similarità con gli stessi.

A causa della sparsità dei dati, vari profili, caratterizzati da una similarità particolarmente bassa con i medoidi dei rispettivi cluster, risultavano poco

significativi ai fini di una ricerca per similarità, costituendo una fonte di perturbazione dei dati. Di conseguenza è stato scelto di filtrare i profili da impiegare in questo test per le interrogazioni, scegliendo solamente quelli che superavano una soglia minima di similarità con i medoidi dei loro cluster di appartenenza. L'esperimento è stato ripetuto per diversi valori, comunque contenuti entro margini realistici, di similarità minima.

Imponendo una soglia minima di similarità pari a 0,2 sono stati mantenuti 1573 profili, per una similarità media con i rispettivi medoidi pari a 0,34656, il valore di soglia 0,3 è stato superato da 984 profili, per una similarità media pari a 0.40298, mentre, aumentando il valore di soglia a 0,35, il numero di profili residui è sceso a 702, mentre la similarità media con i medoidi è salita a 0.43421.

Il grafico 5.12 riporta, per ogni combinazione dei parametri n ed m, il numero di ricerche che non hanno prodotto risultati al variare della soglia di similarità, il grafico 5.13 mostra il numero di ricerche che hanno prodotto solamente risultati diversi dal medoide cercato, il grafico 5.14 e il grafico 5.15 mostrano rispettivamente la media e la varianza della differenza di similarità dei profili ottenuti nel caso precedente con i rispettivi medoidi, infine, il grafico 5.16 riporta, per ciascuna combinazione di n ed m, la percentuale di ricerche concluse con successo.

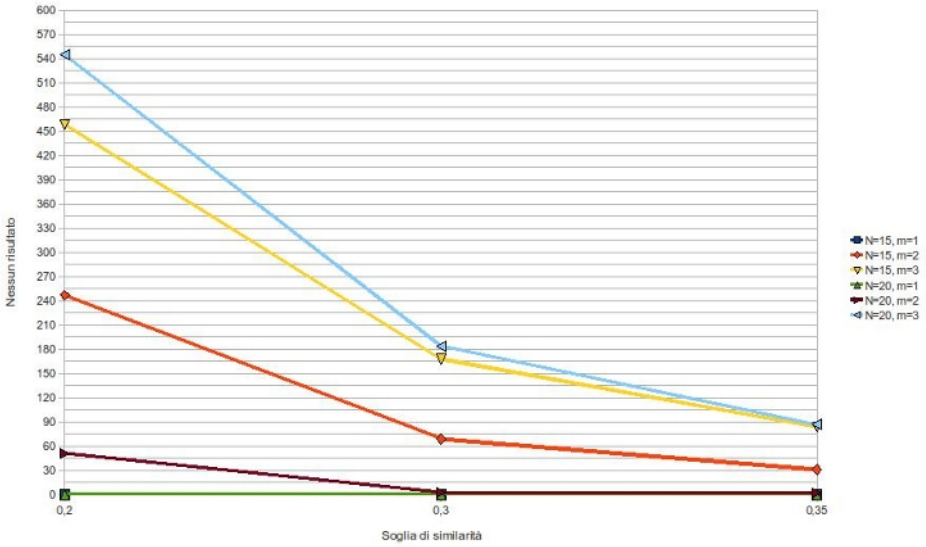


Figura 5.12: Ricerche senza alcun risultato, test 5.3.2.3

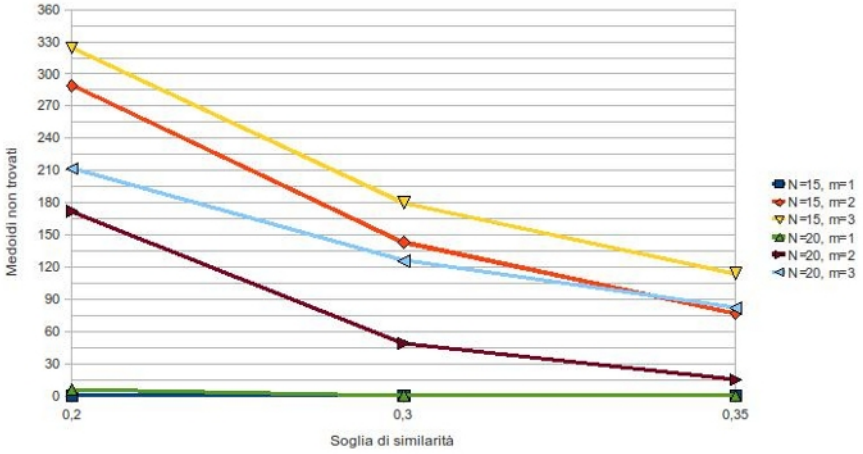


Figura 5.13: Ricerche con soli risultati diversi dal medoide, test 5.3.2.3

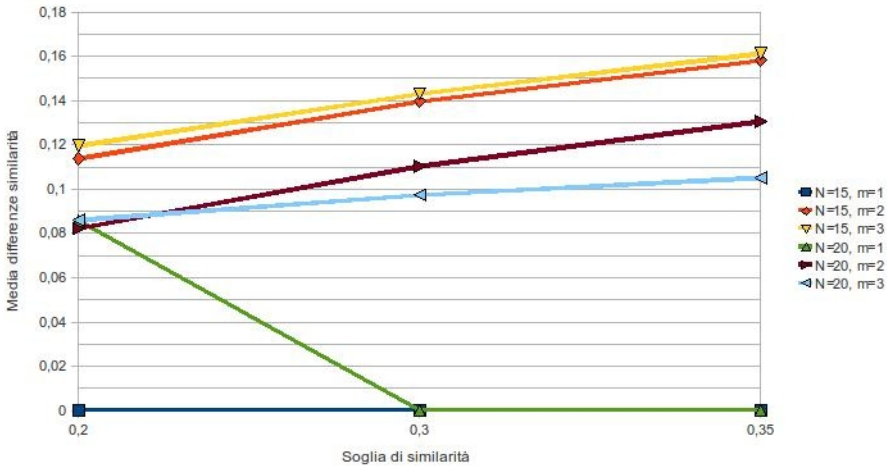


Figura 5.14: Media delle differenze di similarità tra i risultati ottenuti e i medoidi, test 5.3.2.3

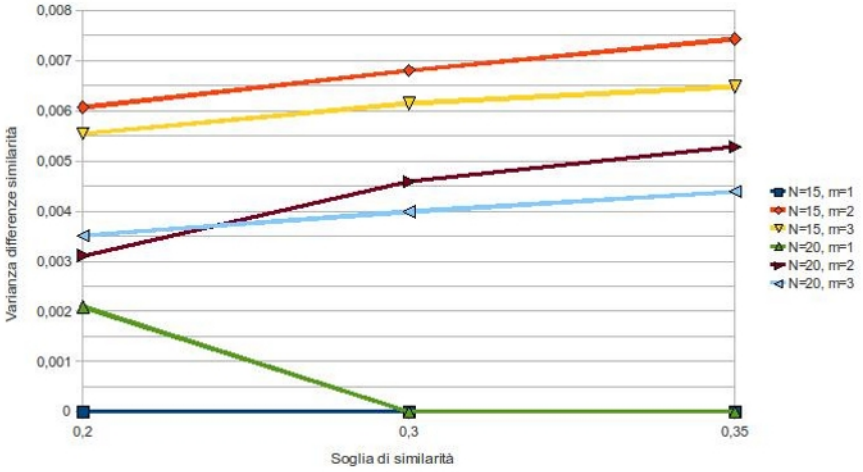


Figura 5.15: Varianza della differenza di similarità tra i risultati ottenuti e i medoidi, test 5.3.2.3

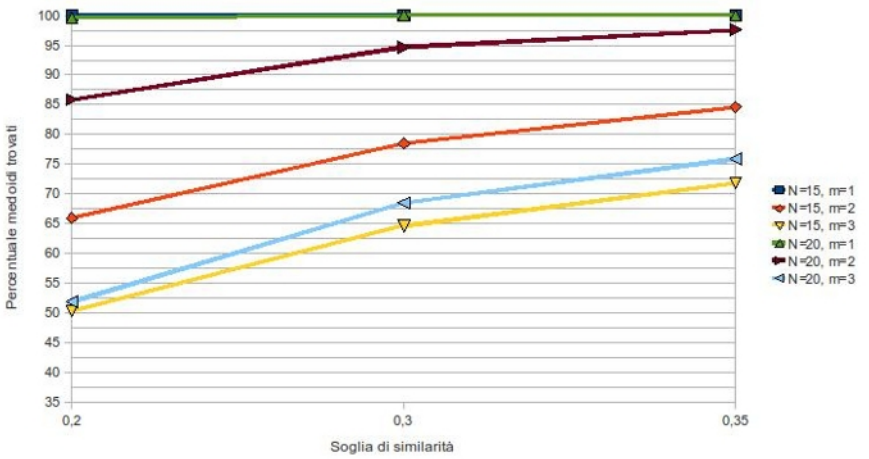


Figura 5.16: Percentuale di medoidi ottenuti, test 5.3.2.3

5.3.2.4 Ricerca dei medoidi per similarità con profili utente filtrati, profili di comunità casuali e alterati

L'ultimo test eseguito per validare l'implementazione delle *min-wise independent permutations* è stato eseguito a partire dai dati impiegati per l'esperimento precedente, aggiungendo ai profili utilizzati per simulare quelli delle comunità, un insieme di vettori di termini pesati generati alterando progressivamente i medoidi individuati da K-Means. A partire dai 150 termini di ciascun medoide è stata creata una successione di profili alterati caratterizzati da una similarità decrescente con il profilo di partenza, variando per il primo 40 termini e progredendo di 10 in 10 fino a 140, in modo da ottenere profili sufficientemente distinti dall'originale e tra di loro, ma comunque correlati. In questo modo, da ciascun medoide sono stati ricavati 11 profili alterati.

Anche in questo caso sono stati impiegati, come chiavi per le ricerche i profili filtrati in base alla similarità minima con i rispettivi medoidi.

I risultati sono riportati, con le stesse modalità dei casi precedenti, nei grafici da 5.17 a 5.21.

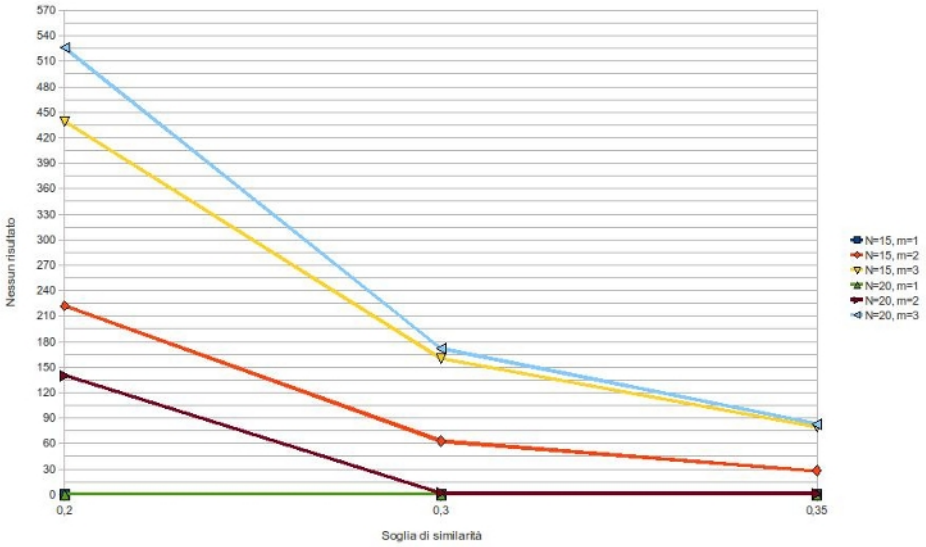


Figura 5.17: Ricerche senza alcun risultato, test 5.3.2.4

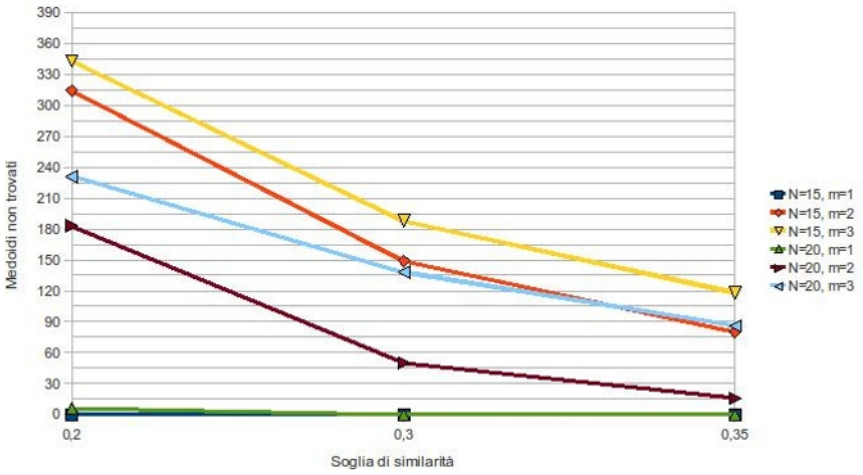


Figura 5.18: Ricerche con soli risultati diversi dal medoide, test 5.3.2.4

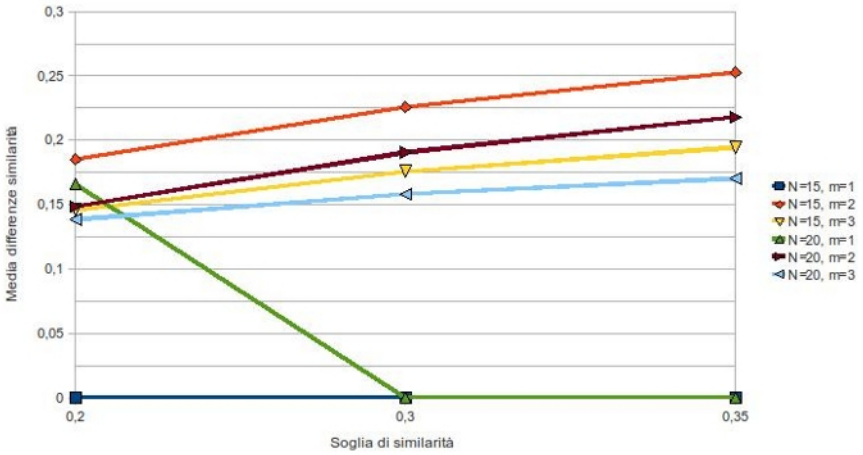


Figura 5.19: Media delle differenze di similarità tra i risultati ottenuti e i medoidi, test 5.3.2.4

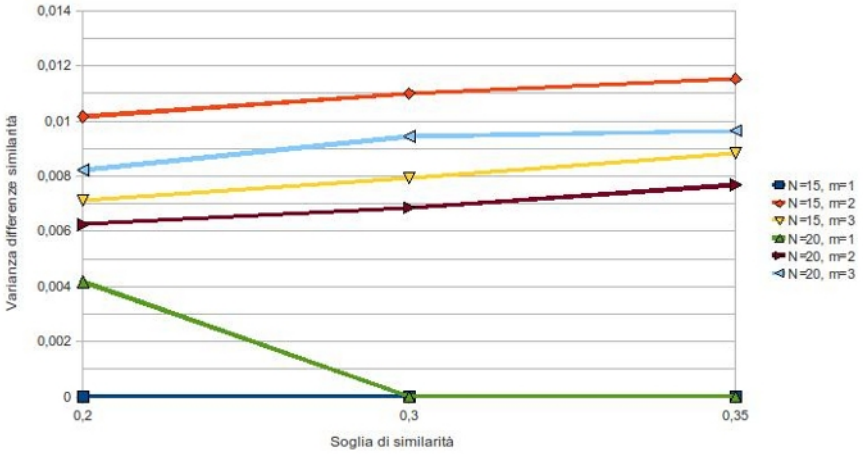


Figura 5.20: Varianza della differenza di similarità tra i risultati ottenuti e i medoidi, test 5.3.2.4

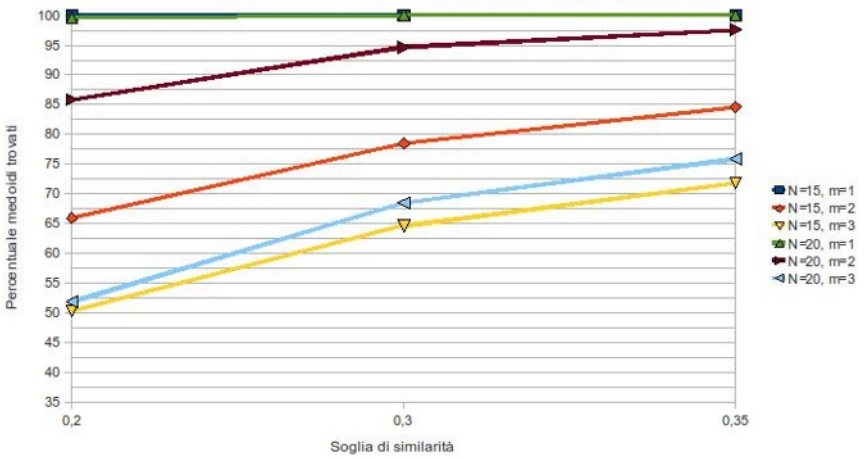


Figura 5.21: Percentuale di medoidi ottenuti, test 5.3.2.4

5.3.2.5 Risultati dei test

Questi test hanno mostrato risultati coerenti con le proprietà delle funzioni LSH per quanto riguarda il numero di medoidi individuati rapportato alla similarità tra i medoidi stessi e i profili. La perturbazione dei dati introdotta nell'ultimo test ha prodotto cambiamenti assolutamente marginali nei risultati, dal momento che l'unica differenza riscontrata è stata relativa al fatto che pochissime ricerche che precedentemente non avevano avuto alcun esito hanno in questo caso ottenuto come risultato dei profili perturbati, mentre niente è cambiato riguardo alle altre.

L'introduzione dei profili casuali, inoltre, non ha avuto in pratica effetti sull'esito delle ricerche, dal momento che in nessun caso in cui non è stato individuato il medoide le ricerche hanno avuto come risultato uno di tali profili.

Le percentuali di successo ottenute, specie per $m = 1$, e i margini di errore contenuti confermano che, impiegando le funzioni LSH definite a partire dall'approssimazione delle min-wise independent permutations, è possibile individuare a partire dal profilo di un utente, profili di comunità soddisfacentemente simili anche nel caso in cui i dati siano molto sparsi.

5.4 Test distribuiti di ricerca per similarità

In questa sezione sono descritti i test effettuati per mettere a confronto la capacità del sistema di fornire, in risposta ad una interrogazione per similarità, i risultati più adeguati, impiegando i due metodi di indicizzazione considerati e al variare dei parametri relativi alla generazione degli identificatori con funzioni LSH.

I test sono stati effettuati su una rete Pastry [27] composta da 2000 nodi impiegando l'emulatore di Overlay Weaver.

5.4.1 Vettori n-dimensionali

Per questo test sono stati selezionati casualmente 300 profili di 150 termini, tra i 2754 disposizione, che sono stati memorizzati sulla DHT, impiegando successivamente i restanti profili per effettuare ricerche per similarità.

Inizialmente queste operazioni sono state effettuate impiegando l'indicizzazione dei singoli attributi e i risultati ottenuti dalle ricerche per similarità sono stati presi come riferimento in quanto ottenuti tramite un metodo *esatto*.

In seguito la memorizzazione dei profili delle 300 comunità e le successive 2454 ricerche per similarità sono state ripetute utilizzando l'indicizzazione dei profili con funzioni LSH, al variare del numero di indentificatori associati a ciascun profilo (parametro n dell'algoritmo di indicizzazione) e del numero di funzioni LSH composte per generare ciascun identificatore (parametro m dell'algoritmo).

La qualità dei risultati ottenuti impiegando questo metodo di indicizzazione è stata valutata, al variare dei due parametri, confrontando i risultati ottenuti con quelli prodotti dalle ricerche esatte condotte in precedenza utilizzando l'indicizzazione dei singoli attributi.

I dati misurati sono:

- il numero di volte in cui con entrambi i metodi è stato otenuto lo stesso profilo come risultato più significativo, mostrato nel grafico 5.22;

- la media e la varianza delle differenze di similarità tra i profili più significativi ottenuti con i due metodi, mostrate rispettivamente nel grafico 5.23 e nel grafico 5.24;

- il valore medio dell'errore relativo sulla similarità dei profili più significativi ottenuti con i due metodi, mostrato nel grafico 5.25

- la media e la varianza delle differenza di similarità tra gli interi insiemi dei risultati ottenuti, per ciascuna query, con i due metodi, mostrate rispettivamente nel grafico 5.27 e nel grafico 5.28;

- il valore medio dell'errore relativo sulla similarità tra gli interi insiemi dei risultati ottenuti ottenuti con i due metodi, mostrato nel grafico 5.29

- una stima della precisione dei risultati ottenuti con l'indicizzazione tramite funzioni LSH rispetto a quelli ottenuti con l'indicizzazione per singoli attributi, definita come $P = \sum_{i=1}^{2454} \frac{|Res-SA_i \cap Res-LSH_i|}{|Res-LSH_i|}$ dove $Res - SA_i$ è l'insieme dei risultati ottenuti per l'i-esima interrogazione impiegando l'indicizzazione per singoli attributi e $Res - LSH_i$ quello dei risultati ottenuti per la stessa query utilizzando il metodo basato su funzioni LSH, mostrata nel grafico 5.26;

- il numero di interrogazioni non risolte calcolando gli identificatori tramite funzioni LSH, mostrato nel grafico 5.30.

Il valore medio della similarità di Jaccard tra i profili utilizzati per le interrogazioni e quelli memorizzati sulla DHT era circa 0,1414.

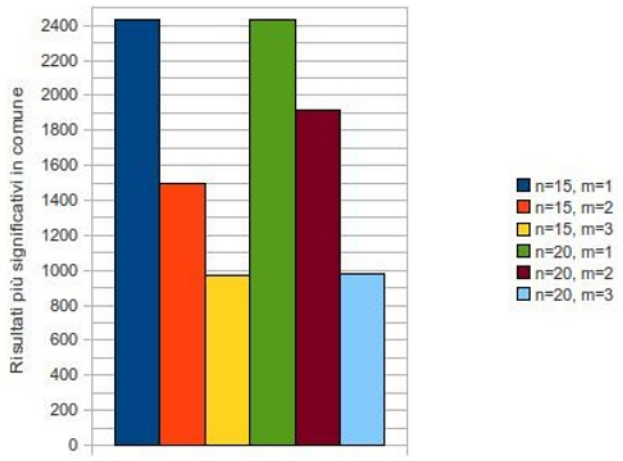


Figura 5.22: Numero di risultati più significativi in comune, test 5.4.1

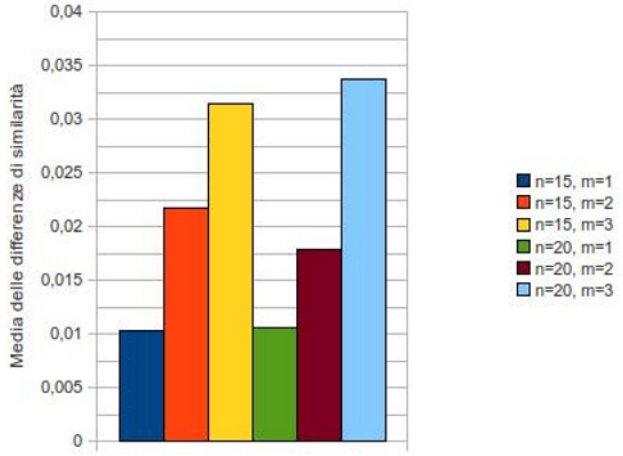


Figura 5.23: Media delle differenze di similarità tra i risultati più significativi, test 5.4.1

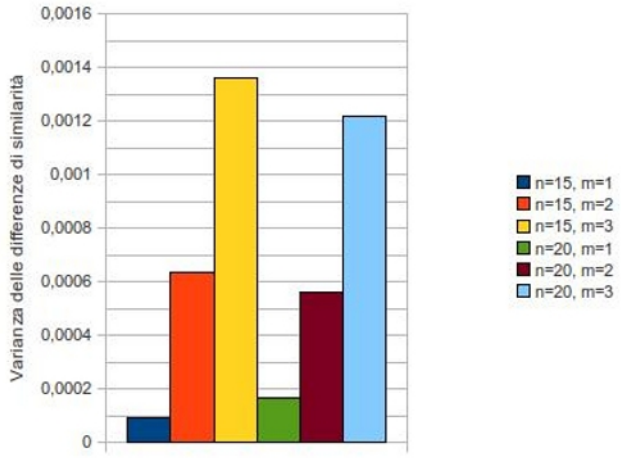


Figura 5.24: Varianza delle differenze di similarità tra i risultati più significativi, test 5.4.1

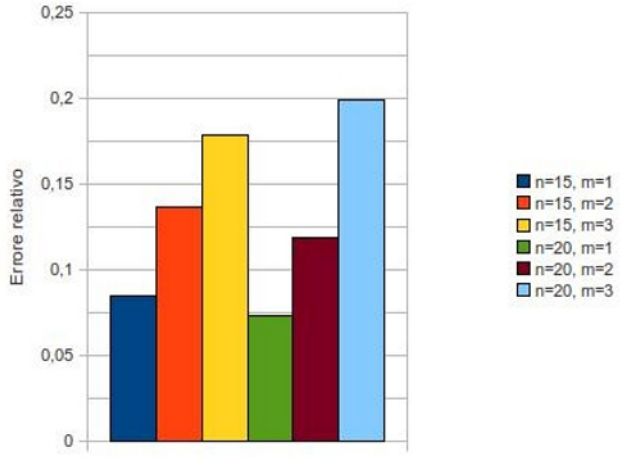


Figura 5.25: Media degli errori relativi sulla differenza di similarità tra i risultati più significativi, test 5.4.1

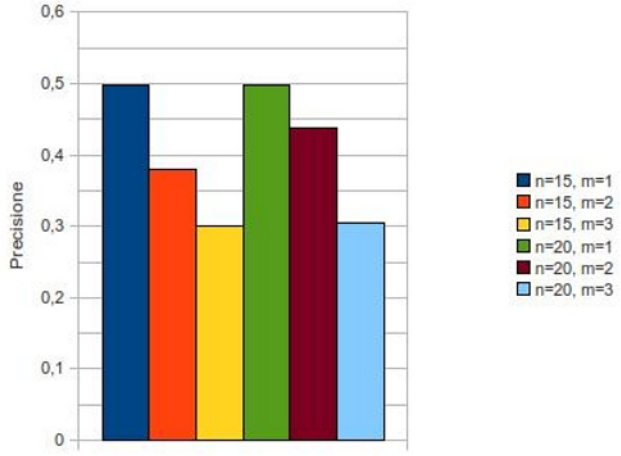


Figura 5.26: Stima della precisione complessiva dei risultati, test 5.4.1

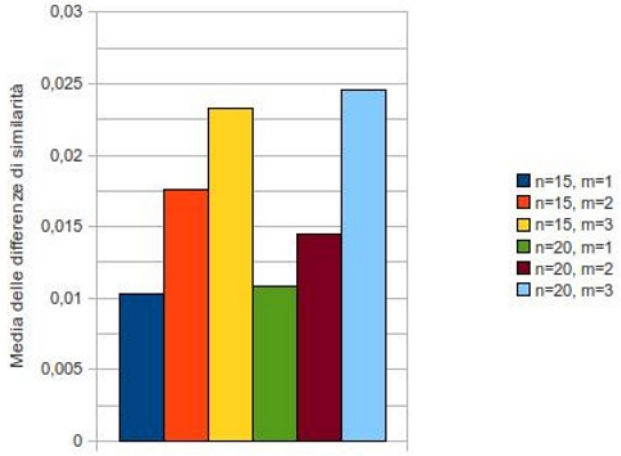


Figura 5.27: Media delle differenze di similarità tra tutti i risultati, test 5.4.1

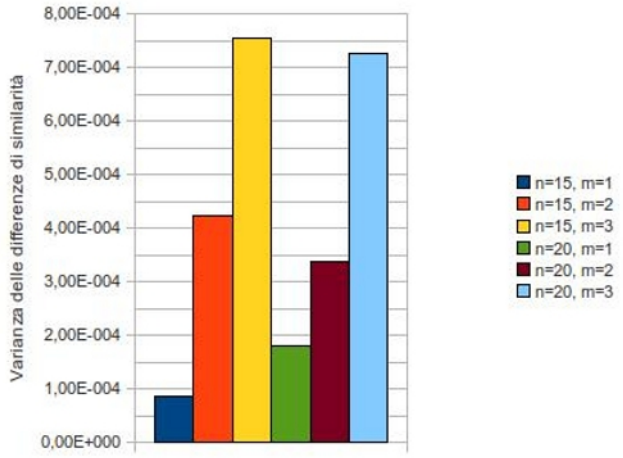


Figura 5.28: Varianza delle differenze di similarità tra tutti i risultati test 5.4.1

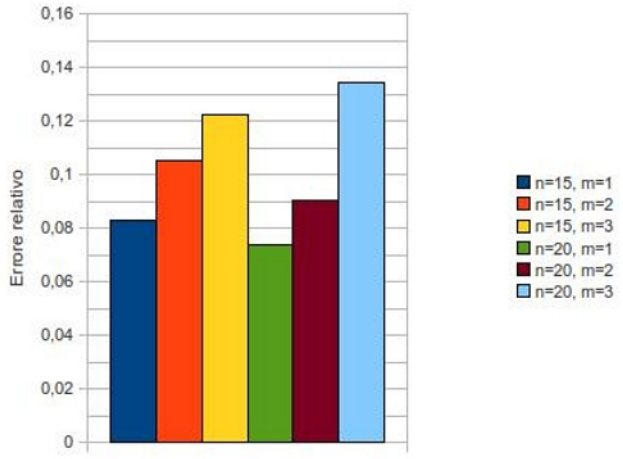


Figura 5.29: Media degli errori relativi sulla differenza di similarità tra tutti i risultati, test 5.4.1

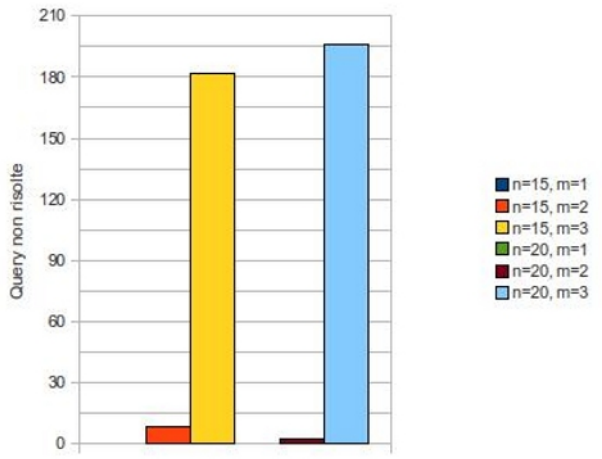


Figura 5.30: Numero di interrogazioni non risolte, test 5.4.1

I risultati ottenuti impiegando il metodo di indicizzazione e ricerca tramite funzioni LSH sono, per m pari a 1, perfettamente assimilabili a quelli ottenuti utilizzando l'indicizzazione dei singoli attributi.

Per quanto riguarda i risultati più significativi, infatti, la precisione è del 99,02% circa per $n = 15$ e del 99,1% circa per $n = 20$, mentre i margini di errore si mantengono in media molto bassi in termini assoluti e la media dell'errore relativo è limitata, nel caso peggiore (per $n = 15$ e$m = 3$), a circa 0,18, mentre si riduce a circa 0,07 per $n = 20$ e $m = 1$.

Relativamente alla totalità dei risultati ottenuti, sebbene la precisione complessiva cali significativamente, i margini di errore rimangono entro limiti che consentono di considerarne adeguata la qualità.

Anche in questo caso, infatti, la differenza media tra i risultati ottenuti con i due metodi di indicizzazione è estremamente bassa, visto che l'errore relativo è in media di circa 0,12 nel caso peggiore, mentre scende anche in questo caso a circa 0,07 per $n = 20$ e $m = 1$.

Il numero di query non risolte impiegando l'indicizzazione tramite funzioni LSH subisce un drastico aumento per $m = 3$, comportamento che, vista la bassa similarità media tra i profili impiegati per le interrogazioni e quelli memorizzati sulla DHT, rientra nella normalità, dato che lo scopo del combinare più funzioni per il calcolo di ciascun identificatore è appunto prevenire

l'ottenimento di falsi positivi in risposta alle interrogazioni. I risultati ottenuti possono quindi ritenersi corretti anche in questo caso, considerando che vengono comunque forniti risultati per il 92,58% circa delle interrogazioni, con margini di errore che si mantengono molto bassi.

5.4.2 Matrici di co-occorrenza tra termini

Per questo test, analogamente a quello effettuato per i vettori di termini pesati, l'indice distribuito è stato creato a partire da 300 profili, che simulavano quelli delle comunità, mentre i restanti sono stati utilizzati per le interrogazioni.

I profili memorizzati sulla DHT appartenevano agli stessi utenti selezionati per il test precedente, quindi i vettori impiegati in quel caso coincidevano con le diagonali principali delle matrici memorizzate sull'indice per questo test.

Anche in questo caso i risultati ottenuti utilizzando l'indicizzazione con funzioni LSH al variare dei parametri sono stati confrontati con quelli forniti dalle interrigazioni effettuate tramite l'indicizzazione dei singoli attributi.

I risultati di questo confronto sono riportati sui grafici seguenti, in particolare:

- il grafico 5.31 mostra il numero di volte in cui con entrambi i metodi è stato otenuto lo stesso profilo come risultato più significativo;
- i grafici 5.32 e 5.33 illustrano rispettivamente la media e la varianza delle differenze di similarità tra i profili più significativi ottenuti con i due metodi;
- il grafico 5.34 riporta il valore medio dell'errore relativo sulla similarità dei profili più significativi ottenuti con i due metodi;
- i grafici 5.35 e 5.36 mostrano rispettivamente la media e la varianza delle differenza di similarità tra gli interi insiemi dei risultati ottenuti, per ciascuna query, con i due metodi;

- il grafico 5.37 mostra il valore medio dell'errore relativo sulla similarità tra gli interi insiemi dei risultati ottenuti ottenuti con i due metodi;
- il grafico 5.38 riporta la stima della precisione dei risultati ottenuti con l'indicizzazione tramite funzioni LSH rispetto a quelli ottenuti con l'indicizzazione per singoli attributi;
- il grafico 5.39 mostra il numero di interrogazioni non risolte calcolando gli identificatori tramite funzioni LSH.

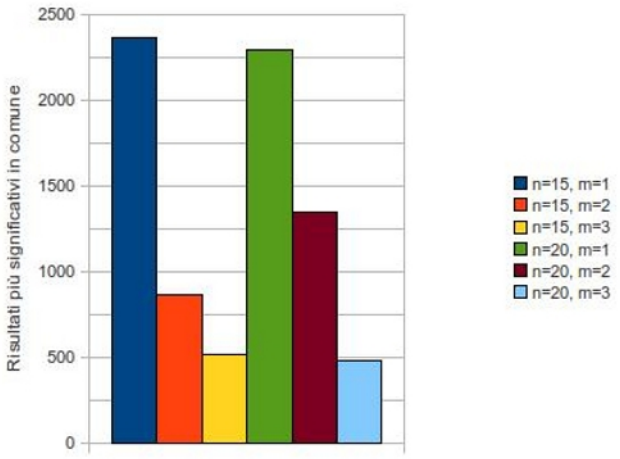


Figura 5.31: Numero di risultati più significativi in comune, test 5.4.2

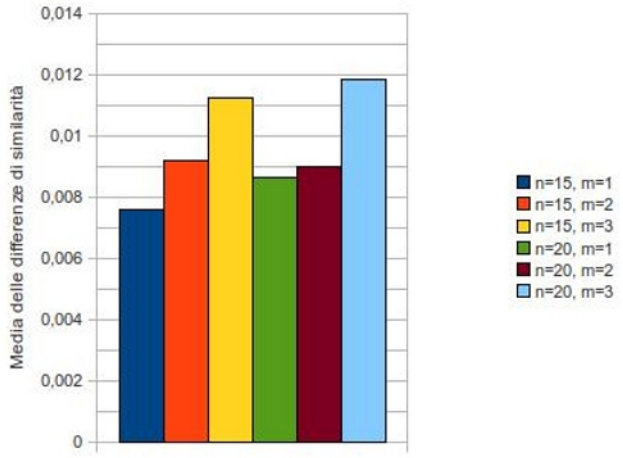


Figura 5.32: Media delle differenze di similarità tra i risultati più significativi, test 5.4.2

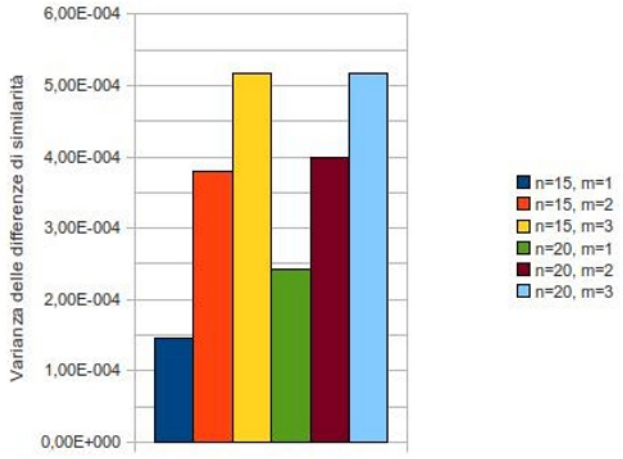


Figura 5.33: Varianza delle differenze di similarità tra i risultati più significativi, test 5.4.2

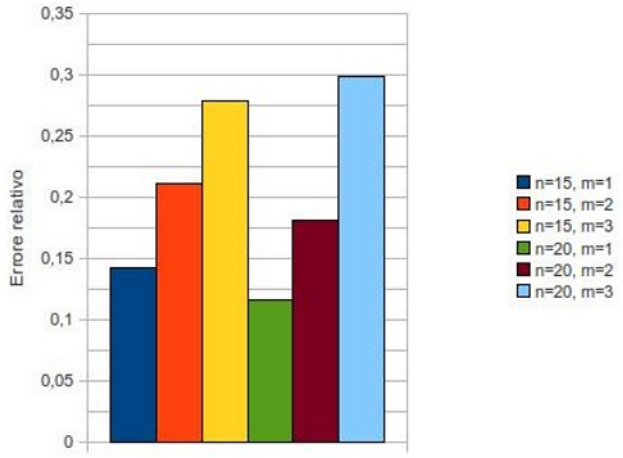


Figura 5.34: Media degli errori relativi sulla differenza di similarità tra i risultati più significativi, test 5.4.2

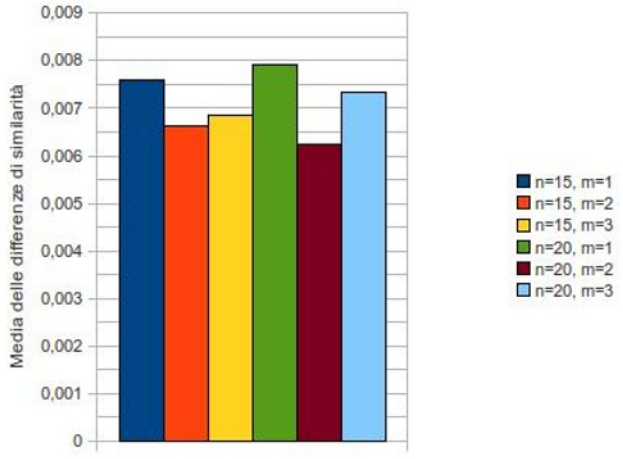


Figura 5.35: Media delle differenze di similarità tra tutti i risultati, test 5.4.2

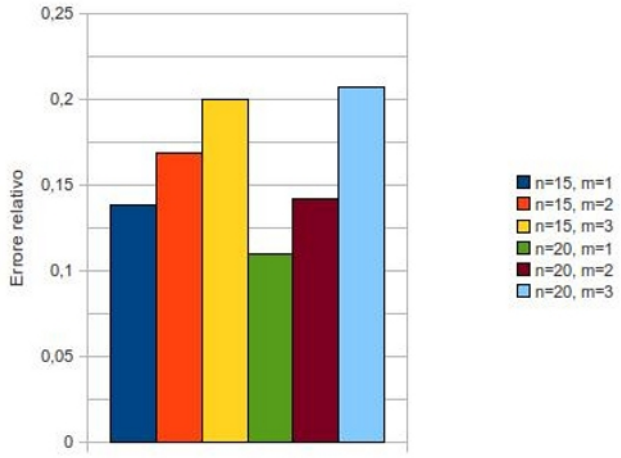


Figura 5.37: Media degli errori relativi sulla differenza di similarità tra tutti i risultati, test 5.4.2

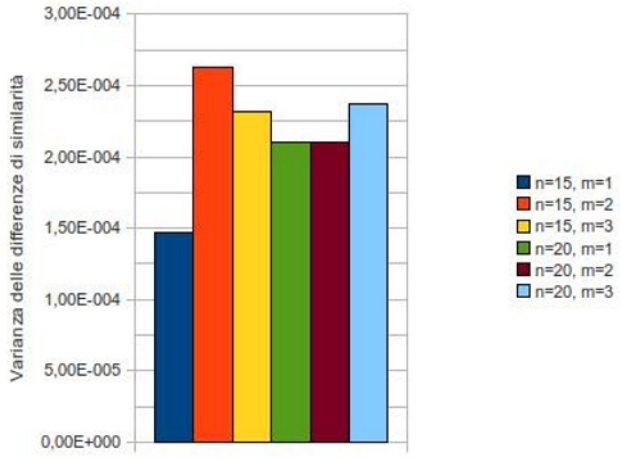


Figura 5.36: Varianza delle differenze di similarità tra tutti i risultati test 5.4.2

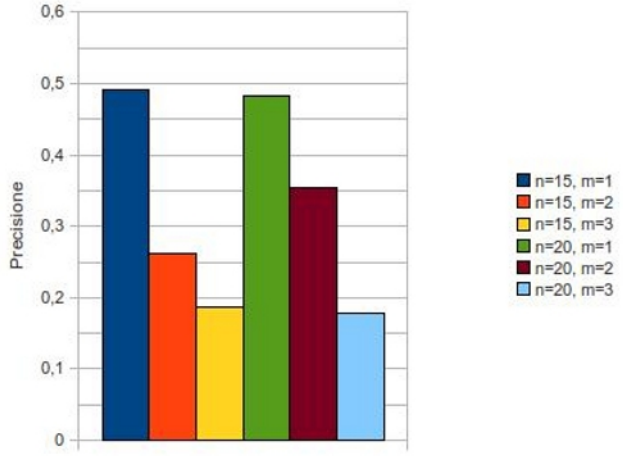


Figura 5.38: Stima della precisione complessiva dei risultati, test 5.4.2

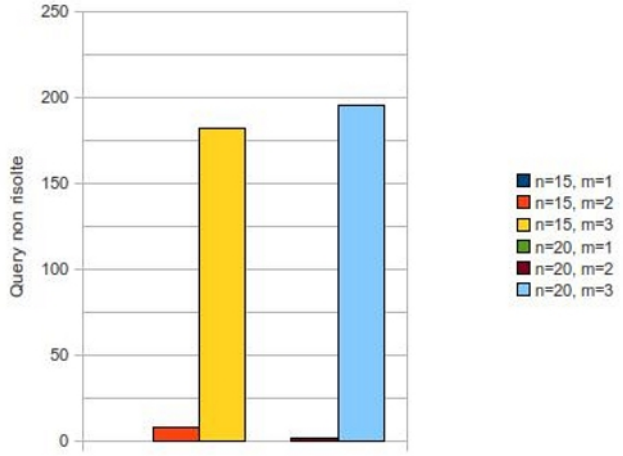


Figura 5.39: Numero di interrogazioni non risolte, test 5.4.2

I risultati ottenuti dimostrano che è possibile impiegare con ottimi risultati l'indicizzazione tramite funzioni LSH anche per profili di questo tipo, ma che, rispetto al caso dei profili espressi come vettori pesati, la perdita di precisione aumenta più velocemente al crescere del numero di funzioni utilizzate per comporre ciascun identificatore, con errori relativi più elevati, anche se non eccessivi.

Ciò può essere dovuto al fatto che, nella fase di indicizzazione, non vengono considerati i valori di co-occorrenza tra gli attributi, che invece vengono valutati dai nodi della DHT che rispondono alle interrogazioni per selezionare i profili di comunità in assoluto più simili. Il filtraggio dei risultati effettuato "a priori" utilizzando identificatori LSH composti può infatti aver portato, complice la bassa similarità dei profili in questione, a non considerare sufficientemente simili dei profili di comunità caratterizzati dalla condivisione con i profili degli utenti di insiemi di attributi di dimensioni limitate, ma legati da correlazioni particolarmente strette.

I risultati ottenuti impiegando l'indicizzazione con funzioni LSH, ma non la composizione di più funzioni per il calcolo di ciascun identificatore, sono comunque pienamente soddisfacenti, dal momento che la precisione raggiunta nel caso della ricerca della comunità più simile in assoluto è, per entrambi i valori del parametro n sopra il 93,5%, con margini di errore ampiamente tollerabili nei casi in cui non viene restituito il risultato ottimo.

5.4.3 Distribuzione del carico

Attraverso opportune misure sono stati valutati gli effetti di LSH sulla distribuzione degli elementi dell'indice generato per il test 5.4.1 sui nodi della rete, verificando se e quanto questo inducesse uno sbilanciamento del carico rispetto al caso in cui vengono indicizzati i singoli attributi di ciascun profilo di comunità.

Da tali misurazioni è emerso che il carico viene distribuito su poche decine dei 2000 nodi che costituiscono la rete, mentre indicizzando i singoli attributi vengono impiegati oltre 1660 nodi.

Il numero medio di oggetti assegnati a ciascun nodo, nel caso in cui gli identificatori vengono generati mediante funzioni LSH non scende sotto i 58 circa, mentre, nel caso dell'indicizzazione dei singoli attributi si ferma a circa 27.

Anche tenendo conto della differenza di dimensioni dell'indice indotta dall'impiego delle diverse modalità di indicizzazione (4500 oggetti indicizzando ciascun profilo con 15 identificatori calcolati con funzioni LSH, 6000 oggetti

impiegando 20 identificatori per ciascun profilo, 45000 indicizzando i singoli attributi), tali differenze sono apparse accettabili.

A livello globale, considerando le intere tabelle hash costruite a prescindere dalla loro distribuzione sulla rete, la suddivisione delle repliche dei profili tra le varie entry risulta soddisfacente per entrambi i metodi di indicizzazione.

Per motivare tale squilibrio nella distribuzione del carico si è ipotizzato che questa sia dovuta alla concentrazione degli identificatori calcolati in un sottoinsieme molto limitato dello spazio $[0, 2^{128} - 1]$ sul quale vengono mappati gli indentificatori dei nodi di una rete Pastry, costituito da valori molto piccoli, a causa del fatto che il valore di ciascuno degli identificatori associati a un profilo è dato dal minimo tra i molti valori ottenuti applicando la stessa permutazione ai singoli attributi dello stesso.

Per verificare questa ipotesi sono stati calcolati, al variare del numero di identificatori calcolati per ciascun profilo e del numero di funzioni LSH utilizzate per comporre ciascun identificatore, il massimo, il minimo e la media del rapporto tra i valori degli identificatori generati e $2^{128} - 1$, massimo valore possibile per un identificatore.

Questa verifica ha mostrato che, nel migliore dei casi, il valore medio degli identificatori generati era contenuto entro il 5% del massimo possibile, confermando quindi l'ipotesi di partenza.

Per ovviare a questo problema è stato scelto di sottoporre gli identificatori generati tramite funzioni LSH ad un ulteriore passo di hashing utilizzando la stessa funzione hash robusta impiegata per la generazione degli identificatori tradizionali.

In questo modo, con un procedimento sicuro e di basso costo computazionale, gli identificatori sono stati ridistribuiti uniformemente sul loro spazio, mantenendo al tempo stesso il mapping ottenuto tramite le funzioni LSH e, di conseguenza, senza intaccare il comportamento del procedimento di ricerca per similarità.

Le misure effettuate per verificare la distribuzione del carico sono state ripetute dopo l'introduzione di questa modifica, mostrando una situazione drasticamente migliore relativamente alla ripartizione del carico sui nodi e dei valori degli identificatori nel loro spazio numerico, mentre la suddivisio-

ne degli oggetti all'interno della tabella hash globale rimane correttamente invariata.

Questo ulteriore passaggio consente inoltre di svincolare la dimensione degli identificatori impiegati sulla DHT da quella degli identificatori calcolati tramite funzioni LSH.

Le misure, effettuate al variare della configurazione impiegata per l'indicizzazione dei profili, riguardano:

- il numero di nodi della DHT ai quali è stata assegnata almeno una copia di un profilo di comunità, mostrato nel grafico 5.40;
- il numero complessivo di entry che compongono la DHT, mostrato nel grafico 5.41;
- il numero medio di oggetti assegnati a ciascuno dei nodi responsabili di almeno una copia di un profilo di comunità, mostrato nel grafico 5.42;
- il numero medio di entry della DHT assegnate a ciascuno dei nodi responsabili di almeno una copia di un profilo di comunità, mostrato nel grafico 5.43;
- il numero medio di oggetti mappati in ciascuna entry della DHT, mostrato nel grafico 5.44;
- il valore medio del rapporto tra il valore degli identificatori e il massimo elemento dello spazio degli identificatori ($2^{128} - 1$), mostrato nel grafico 5.45.

In ogni grafico sono messi a confronto i valori misurati indicizzando i profili con gli identificatori calcolati tramite funzioni LSH, con gli stessi identificatori sottoposti ad un ulteriore hashing e con gli identificatori ottenuti dall'hashing robusto dei singoli attributi.

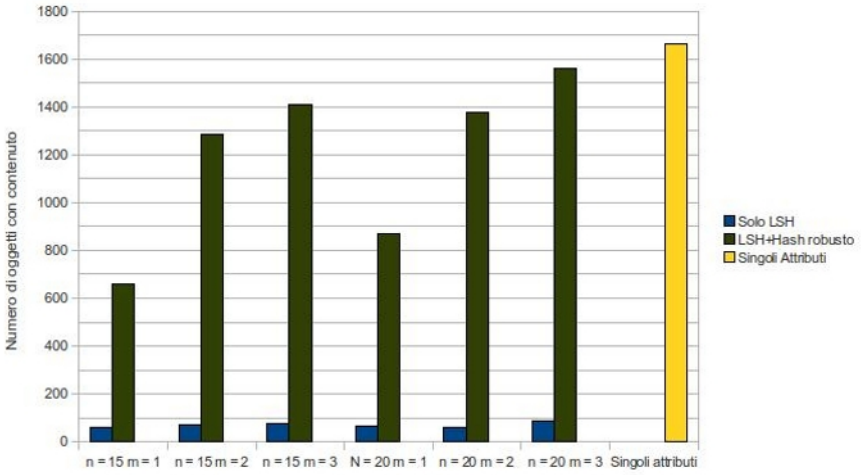


Figura 5.40: Numero di nodi carichi della DHT

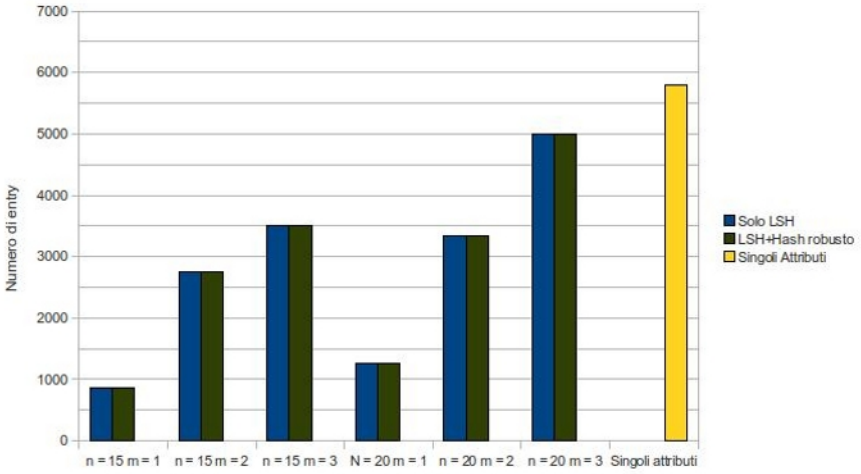


Figura 5.41: Numero di entry della DHT

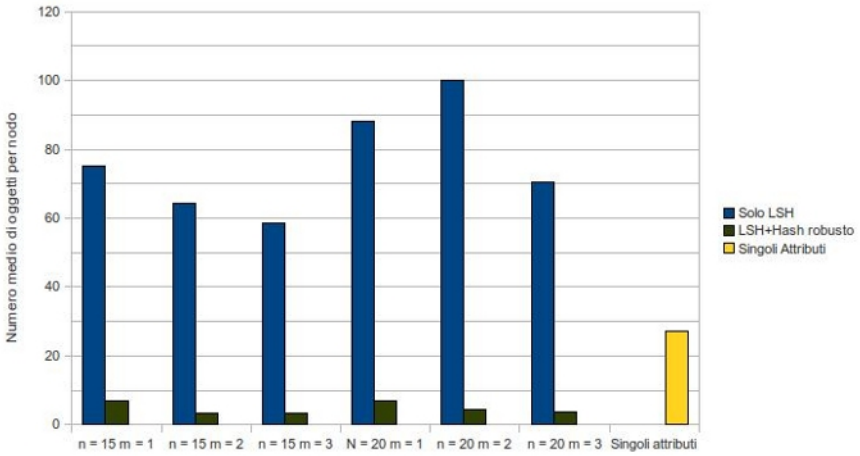


Figura 5.42: Numero medio di oggetti per nodo della DHT

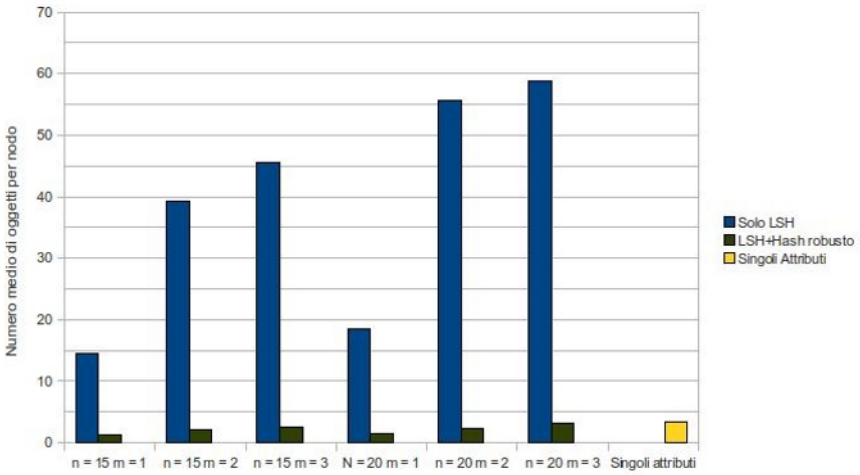


Figura 5.43: Numero medio di entry per ciascun nodo della DHT

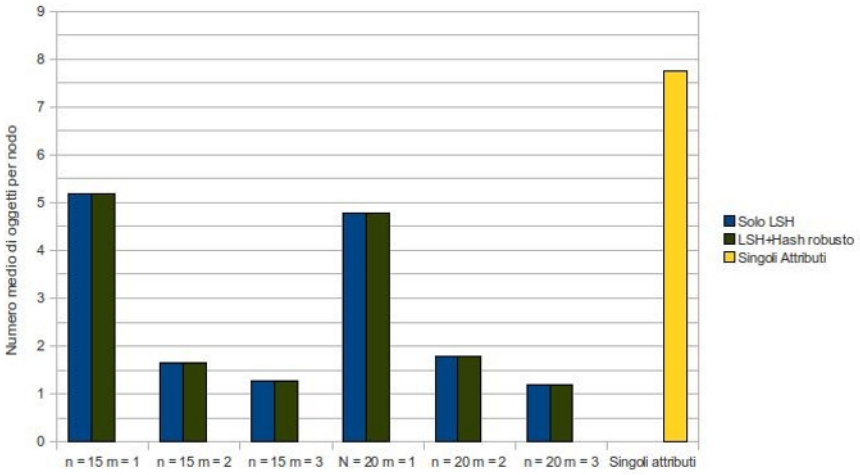


Figura 5.44: Numero medio di oggetti mappati su ciascuna entry della DHT

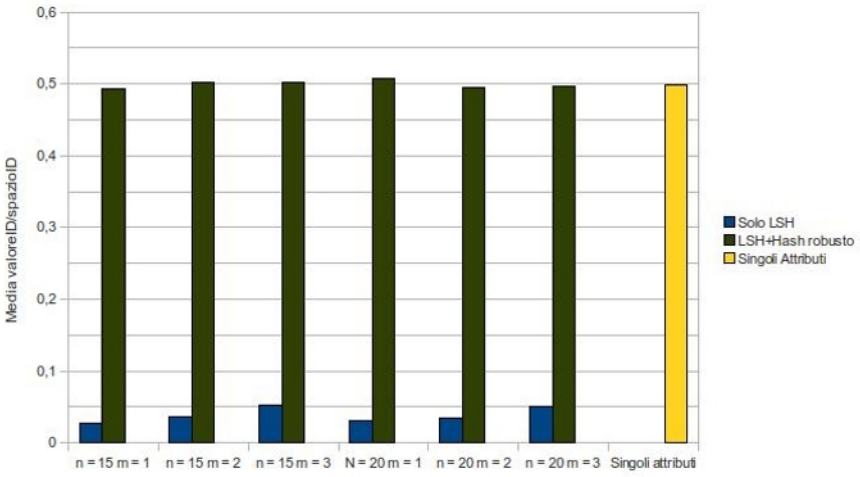


Figura 5.45: Rapporto medio tra valore degli indentificatori e dimensione dello spazio

5.5 Stima del consumo di banda e di spazio di memorizzazione

Per ciascun tipo di profilo è stata effettuata una stima dello spazio occupato dall'indice distribuito e del traffico generato per la memorizzazione dei profili sull'indice e per invio delle interrogazioni e dei messaggi di risposta alle stesse sulla DHT, non considerando il costo del routing. I messaggi di interrogazione includono i profili degli utenti.

Questi valori sono stati calcolati sia per il metodo di indicizzazione basato sui singoli attributi che per quello basato su funzioni LSH, prevedendo l'associazione, in quest'ultimo caso, di 15 o 20 identificatori a ciascun profilo.

Per il test riguardante i vettori n-dimensionali e le matrici di co-occorrenza sono stati impiegati 2515 profili di 300 elementi ciascuno, dalle matrici sono state estratte le componenti connesse imponendo le soglie $minFreq = 0,25$ e $minSize = 20$, ottenendo profili per 2072 utenti.

Ciascuna stima è relativa all'intero insieme dei profili, il traffico generato dalle risposte alle interrogazioni è stimato ipotizzando che ciascun peer interrogato invii al richiedente i propri 5 risultati più significativi.

I grafici 5.46 e 5.47 mettono a confronto i costi stimati, al variare del metodo di indicizzazione, rispettivamente per i vettori n-dimensionali e le matrici di co-occorenza.

I valori sono espressi in megabyte.

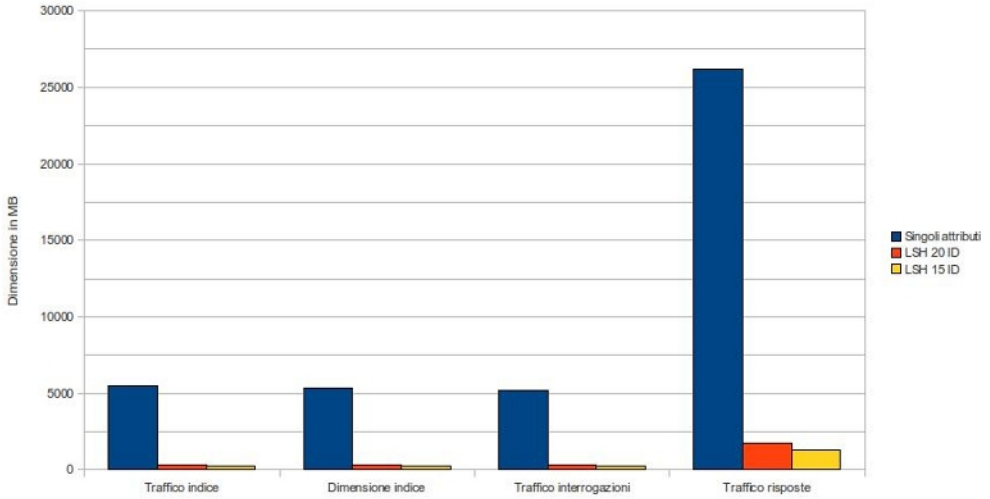


Figura 5.46: Consumo di banda e spazio di memorizzazione: vettori n-dimensionali

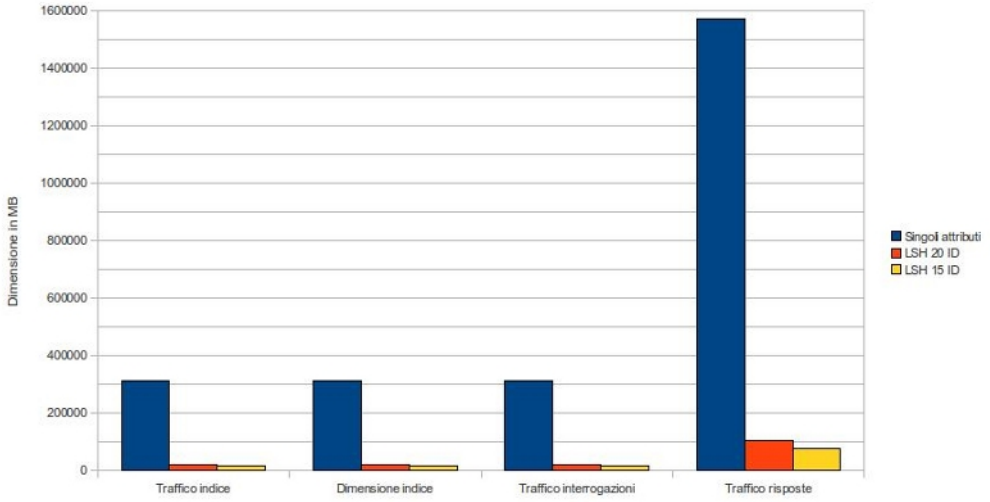


Figura 5.47: Consumo di banda e spazio di memorizzazione: matrici di co-occorrenza

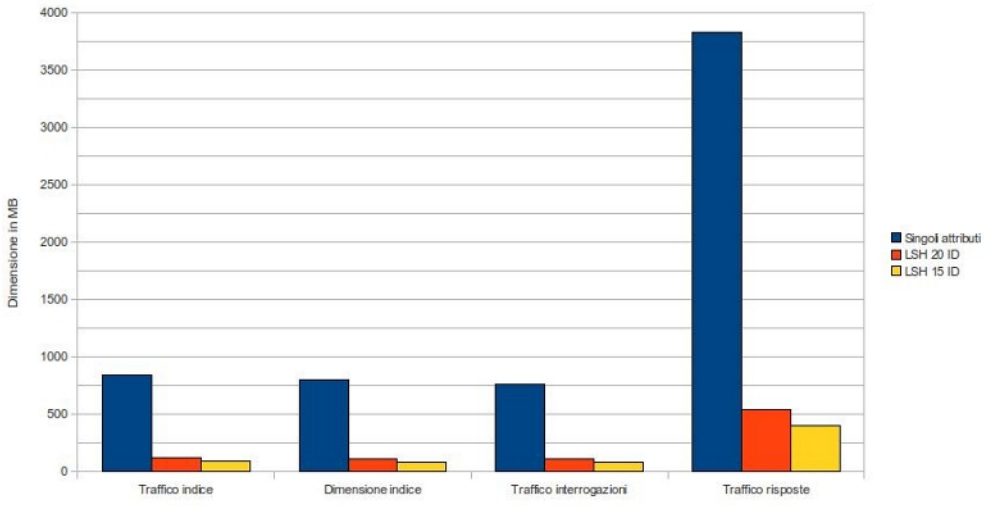


Figura 5.48: Consumo di banda e spazio di memorizzazione: componenti connesse.

È evidente, oltre al grande risparmio di risorse consentito dall'impiego dell'indicizzazione tramite funzioni LSH, il drastico aumento del consumo di risorse nel caso dei profili di tipo matrice e la sua riduzione di un ordine di grandezza nel caso dei profili costituiti dagli insiemi delle componenti connesse.

Capitolo 6

Conclusioni

In questa tesi è stato definito e realizzato un sistema peer to peer strutturato, basato su DHT, per la ricerca per similarità di profili di comunità di utenti individuate in base a un modello dei loro interessi.

Il sistema proposto implementa un indice distribuito che memorizza i profili delle comunità mantenendoli aggiornati in modo da velocizzare la ricerca da parte degli utenti delle comunità più adeguate ai loro interessi e la diffusione delle informazioni riguardanti la creazione di nuove comunità e l'aggiornamento dei profili di quelle esistenti.

Le funzionalità fondamentali offerte dall'indice implementato consistono nel supporto alla ricerca di valori multi attributo su una DHT in base alla loro similarità anziché all'identità e in un meccanismo che consente di notificare automaticamente la memorizzazione ex novo o l'aggiornamento di un profilo di comunità ai membri delle comunità simili.

Per modellare gli interessi degli utenti sono stati definiti tre tipi profilo, tutti basati su un insieme di generici *attributi*: il primo modello proposto prevede semplicemente l'associazione di un peso numerico a ciascun attributo per indicarne l'importanza, il secondo sfrutta, se disponibili, dei valori che stimano il livello di correlazione tra le coppie di attributi del profilo, definendo in questo modo una matrice di correlazione, mentre il terzo, se possibile, decompone ciascun profilo in più sottoinsiemi di attributi strettamente correlati tra loro, ma non con quelli appartenenti agli altri sottoinsiemi.

Anche se le funzionalità necessarie alla realizzazione dell'indice distribuito potevano risultare implementabili facilmente e in modo esatto con un metodo di indicizzazione dei profili basato sulla memorizzazione di una copia di ciascun profilo per ogni suo attributo, i costi in termini di banda e spazio di memorizzazione indotti dall'uso di questa soluzione al crescere del numero degli attributi, rende questa soluzione altamente inefficiente. Per ovviare a questo inconveniente è stata individuato un metodo *probabilistico* basato sull'impiego di *funzioni LSH* [13], definite, nel caso specifico, a partire da un'approssimazione delle *min-wise independent permutations* [28, 14] allo scopo di associare a ciascun profilo un insieme di identificatori di dimensioni contenute e costanti, indipendenti dal numero di attributi, che presentano un'alta probabilità di coincidere con quelli di profili definiti a partire da insiemi di attributi simili secondo la metrica di Jaccard.

L'indice distribuito delle comunità è stato implementato estendendo il servizio di DHT fornito da Overlay Weaver [3] per supportare la ricerca dei profili per similarità, sia tramite l'indicizzazione dei singoli attributi che tramite la generazione degli identificatori con funzioni LSH.

Il confronto tra i risultati ottenuti, a partire da *dati reali*, effettuando ricerche per similarità con entrambi i metodi di indicizzazione ha dimostrato che, utilizzando il metodo basato su funzioni LSH, è possibile risolvere le interrogazioni in modo assolutamente soddisfacente anche in una situazione in cui i dati risultano particolarmente sparsi.

Indicizzando i profili tramite funzioni LSH l'utilizzo di banda e spazio di memorizzazione cala di un ordine di grandezza già per profili di medie dimensioni rispetto a quanto accade indicizzando i singoli attributi, mentre la distribuzione del carico sui nodi della DHT viene mantenuta ottimale anche per questo metodo di indicizzazione con un costo trascurabile.

6.1 Sviluppi futuri

Attualmente le funzioni LSH [13] impiegate per generare gli identificatori dei profili sono implementate a partire da un'approssimazione delle *min-wise independent permutations [28, 14].* È interessante valutare i risultati otte-

nibili utilizzando, oltre ad altre approssimazioni delle *min-wise independent permutations*, altri tipi di funzioni LSH come, ad esempio, le *distribuzioni p-stabili* [39].

L'indice distribuito realizzato in questa tesi verrà integrato con *LEADER* [35] per realizzare il sistema di creazione e gestione delle comunità a due livelli descritto nel capitolo 3.

Una volta compiuta questa integrazione e sviluppati gli opportuni strumenti per estrerre i profili dai dati messi a disposizione dagli utenti, il sistema verrà messo liberamente a disposizione del pubblico in modo da valutarne il comportamento in casi d'uso reali.

In particolare, sfruttando un meccanismo di *feedback*, potrà essere valutata, la fedeltà con cui ciascun tipo di profilo modella gli interessi degli utenti.

Bibliografia

[1] Rüdiger Schollmeier: *A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architectures and Applications.* Proceedings of the First International Conference on Peer-to-Peer Computing, IEEE (2002).

[2] Upendra Shardanand and Pattie Maes. *Social information filtering: algorithms for automating "word of mouth".* In Proc. of the SIGCHI conference on Human factors in computing systems, pages 210–217, New York, NY, USA, 1995. ACM, Addison- Wesley.

[3] Kazuyuki Shudo, Yoshio Tanaka, Satoshi Sekiguchi: *Overlay Weaver: An overlay construction kit.* In Computer Communications, Volume 31, Issue 2, February 2008, Pages 402-412. Butterworth-Heinemann, Newton, MA, USA.

[4] http://www.ncbi.nlm.nih.gov/pubmed

[5] http://tartarus.org/~martin/PorterStemmer/

[6] http://terrier.org

[7] J. B. MacQueen (1967): *Some Methods for classification and Analysis of Multivariate Observations.* Proceedings of 5-th Berkeley Symposium on Mathematical Statistics and Probability, Berkeley, University of California Press, 1:281-297

[8] Stoica, Ion et al. (2001): *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications.* Proceedings of SIGCOMM'01 ACM Press New York, NY, USA

[9] Devavrat Shah: *Gossip Algorithms.* Massachusetts Institute of Technology, MA,USA, 2008.

[10] Spyros Voulgaris, Maarten van Steen, and Konrad Iwanicki: *Proactive gossip- based management of semantic overlay networks*, Department of Computer Science ETH Zurich, Department of Computer Science Vrije Universiteit Amsterdam; Concurrency and Computation: Practice and Experience, 2006.

[11] Q. Hieu Vu, M. Lupo, B. Chin Ooi. *Peer-to-Peer Computing: Principles and Applications.* Springer-Verlag 2010.

[12] Ranieri Baraglia and Fabrizio Silvestri: *An Online Recommender System for Large Web Sites*, Information Science and Technologies Institute (ISTI) National Research Council, Pisa, ITALY.

[13] Piotr Indyk, Rajeev Motwani: *Approximate nearest neighbors: towards removing the curse of dimensionality.* Proceedings of the thirtieth annual ACM symposium on Theory of computing.

[14] Abhishek Gupta, Divyakant Agrawal, Amr El Abbadi, University of California, Santa Barbara, U.S.A.: *Approximate Range Selection Queries in Peer-to-Peer Systems.*

[15] Zujie Ren, Lidan Shou, Gang Chen, Chun Chen, and Yijun Bei, *HAPS: Supporting Effective and Efficient Full-text P2P Search with Peer Dynamics*, Zujie Ren, Lidan Shou, Gang Chen, Chun Chen, and Yijun Bei Zhejiang University, China.

[16] Bender, M., Michel, S., Triantafillou, P., Weikum, G., Zimmer, C.: *Minerva: Collaborative p2p search.* In: VLDB. (2005) 1263–1266.

[17] Nottelmann, H., Fischer, G., Titarenko, A., Nurzenski, A.: *An integrated approach for searching and browsing in heterogeneous peer-to-peer networks.* In: ACM SIGIR WorkShop Het- ergeneous and Distributed Information Retrieval. (2006).

[18] I. Podnar, M. Rajman, T. Luu, K. Aberer, School of Computer and Communication Sciences, Ecole Politecnique Fédérale de Lausanne: *Scalable Peer-to-Peer Web Retrieval with Highly Discriminative Keys*, in ICDE 2007.

[19] A. Crainiceanu, J. Gehrke, P. Linga, A. Machanvajjala, J. Shanmugasundaram: *P-Ring: An Efficient and Robust P2P Range Index Structure.*

[20] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, I. Stoica: *Wide-area cooperative storage with CFS*, in SOSP, 2001.

[21] 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.

[22] Ian Foster, Carl Kesselman: *The Grid 2: Blueprint for a New Computing*, ISBN 1-55860-933-4

[23] http://terrier.org/

[24] Yingwu Zhu, Yiming Hu: *Efficient semantic search on DHT overlays*, Journal of Parellel and Distributed Computing vol. 67 no5, Maggio 2007.

[25] J. G. Siek, L. Lee, and A. Lumsdaine: *Boost Graph Library, The: User Guide and Reference Manual*, Addison Wesley Professional, 2001.

[26] http://v3.espacenet.com/publicationDetails/biblio?CC=US&NR=6285999&KC=&FT=E

[27] A. Rowstron and P. Druschel, *Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems.* IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), Heidelberg, Germany, pages 329-350, November, 2001.

[28] Andrei Z. Broder, Moses Charikar, Alan M. Frieze, Michael Mitzenmacher : *Min-Wise Independent Permutations.* Journal of Computer and System Sciences 60, 630 659 (2000) .

[29] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubia- towicz. *Tapestry: a resilient global-scale overlay for service deployment.* Selected Areas in Communications, IEEE Journal on, 22(1):41-53, 2004.

[30] Petar Maymounkov and David Mazieres. Kademlia: *A Peer-to-Peer Information System Based on the XOR Metric.* Electronic Proceedings for the 1st Interna- tional Workshop on Peer-to-Peer Systems (IPTPS '02), MIT Faculty Club - Cambridge, March 2002.

[31] F. Kaashoek, D. R. Karger: *Koorde: A Simple Degree-optimal Hash Table.* In *2nd International Workshop on Peer to Peer Systems* (IPTPS '03), February 2003, Berkeley, CA.

[32] F. Dabek, B. Zhao, P. Druschel, and I. Stoica. *Towards a common API for structured peer-to-peer overlays.* Peer-to-Peer Systems II, Second International Workshop, (IPTPS 2003), Berkeley - USA, February 2003.

[33] Michael A. Olson, Keith Bostic, Margo Seltzer: *Berkeley DB.* In Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference Monterey, California, USA, June 6–11, 1999.

[34] Sean Rhea, Brighten Godfrey, Brad Karp, John Kubiatowicz, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, Harlan Yu: *OpenDHT: a public DHT service and its uses.* In Proceeding SIGCOMM '05 Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications ACM New York, NY, USA ©2005 table of contents ISBN:1-59593-009-4 doi>10.1145/1080091.1080102

[35] Luca Alessi: *LEADER: un protocollo peer-to-peer per la costruzione dinamica di comunita' basato su Gossip*. Tesi di Laurea Specialistica in Informatica presso l'Università di Pisa, Anno Accademico 2009/2010. Relatori: Dr.ssa Laura Ricci, Dr. Matteo Mordacchini. Reperibile all'indirizzo: http://etd.adm.unipi.it/theses/available/etd-11242010-092243/

[36] *OWL: Web Ontology Language*. W3C Recommendation 10 February 2004. Disponibile all'indirizzo http://www.w3.org/TR/owl-features/

[37] http://wordnet.princeton.edu/

[38] K. Clarkson: *An algorithm for approximate-closest point queries*. In: *Proceedings of the Tenth AnnualACM Symposium on Computational Geometry*, 1994, pp. 160-164

[39] Datar, M., and N. Immorlica and P. Indyk and V. Mirrokni: *Locality-Sensitive Hashing Scheme Based on p-Stable Distributions*. Proceedings of the 20th Symposium on Computational Geometry , New York, NY, June 2004, pp. 253-262.

[40] Paul Jaccard: *Étude comparative de la distribution florale dans une portion des Alpes et des Jura*. Bulletin de la Société Vaudoise des Sciences Naturelles 37: 547–579 (1901).

Elenco degli algoritmi

Elenco delle figure