PARISTORAGE: IMMAGAZZINAMENTO ROBUSTO DELL'INFORMAZIONE

RELATORE: Ch.mo Prof. Enoch Peserico Stecchini Negri De Salvi
CORRELATORE: Ing. Michele Bonazza
LAUREANDO: Andrea Luongo

A.A. 2011-2012

UNIVERSITÀ DEGLI STUDI DI PADOVA
DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE
TESI DI LAUREA

PariStorage: Immagazzinamento robusto dell'informazione

RELATORE: Ch.mo Prof. Enoch Peserico Stecchini Negri De Salvi

CORRELATORE: Ing. Michele Bonazza

LAUREANDO: *Andrea Luongo*

A.A. 2011-2012

"If you can dream it, you can do it"
Walt Disney

Indice

Sommario

I sistemi P2P(Peer-to-Peer) hanno ricevuto molta attenzione negli ultimi anni. In particolare c'è stato un interesse crescente verso l'utilizzo di sistemi P2P per il salvataggio di file. DistributedStorage, plugin del progetto PariPari, si pone l'obiettivo di offrire questo tipo di servizio garantendo un elevato grado di efficienza, sicurezza e affidabilità. In questo elaborato illustrerò il lavoro da me svolto all'interno del gruppo DistributedStorage, in particolare presenterò le modifiche apportate al sistema di codifica dei file.

Introduzione

L'affidabilità è l'obiettivo principale nella progettazione di un sistema di storage. Durante il salvataggio, l'integrità dei dati può essere compromessa da errori di trasmissione: alcuni dei pacchetti inviati potrebbero risultare corrotti, andare persi o potrebbero causare il fallimento dell'intero processo. L'avvenimento di una qualsiasi di queste situazioni comprometterebbe il salvataggio del file e un suo eventuale tentativo di recupero.
Per ovviare a questi problemi è necessario ricorrere ad una *codifica di canale* [1]. Si tratta di un insieme di tecniche per introdurre un certo grado di ridondanza, in modo da rendere il flusso di dati meno soggetto a questo tipo di errori.
Il più semplice esempio di ridondanza è la *replicazione*, caratterizzata da un alto grado di affidabilità e da una bassa efficienza in termini di quantità di dati salvati. Una tecnica più avanzata consiste nell'utilizzo di una categoria di codici, nota come *erasure codes* [2], in grado di garantire lo stesso livello di affidabilità della replicazione con una minor occupazione di spazio.
Mentre per la replicazione, la ridondanza consiste appunto in una "replica" dei dati, per gli erasure codes è più complicato: ogni nuovo bit è ottenuto attraverso operazioni di codifica svolte su diversi bit già esistenti. Per sostituire eventuali pacchetti danneggiati è necessario recuperare tutti i dati utilizzati nella codifica, con un conseguente aumento del costo di I/O, e rieffettuare tutte le operazioni, aumentando il costo computazionale.
Mentre in un sistema di storage tradizionale il numero di riparazioni effettuate è relativamente basso, in un sistema P2P come *PariPari*, in cui i vari nodi possono disconnettersi liberamente, questi costi possono diventare molto elevati. Il recupero dei dati usati per la codifica si traduce in un aumento del traffico di rete. Ed è proprio dal punto di vista delle risorse di rete che i sistemi P2P sono limitati, mentre possono contare su un'elevata quantità di memoria e buone risorse computazionali.

Nel gruppo DistributedStorage ci siamo occupati dell'implementazione di un sistema di codifica in grado di ridurre questo tipo di costi senza penalizzare troppo l'occupazione di memoria. Questa codifica, chiamata Hierarchical Codes [4], si presenta come una sorta di compromesso tra replicazione e erasure codes.
Nel Capitolo 1 verrà data una breve presentazione del progetto PariPari e, in particolare, del plugin DistributedStorage. Nel Capitolo 2 presenterò le principali caratteristiche della replicazione, degli erasure codes,in particolare del codice LDPC[1], e degli Hierarchical Codes. Nel Capitolo 3 verranno mostrate le modifiche apportate al plugin Nel Capitolo 4 verranno tratte delle conclusioni.

[1]Low Density Parity Check code

Capitolo 1

PariPari

Figura 1.1: Il logo di PariPari

PariPari è una rete peer-to-peer pura multifunzionale, realizzata in linguaggio Java™, basata su una tabella hash distribuita simile a Kademlia[3].
Il progetto si pone l'obiettivo di rendere disponibili sia i più tipici servizi di Internet (IRC[1], IM[2], VoIP[3], DBMS[4], Web Server, DNS[5] ...) sia quelle funzioni realizzate dai più famosi programmi P2P (file sharing, storage distribuito, backup distribuito), il tutto garantendo l'anonimato degli utenti e mantenendo una struttura modulare.

[1]Instant Relay Chat
[2]Instant Messaging
[3]Voice over Ip
[4]Database Management System
[5]Domain Name System

1.1 Struttura di PariPari

La multifunzionalità e la modularità di PariPari sono ottenute tramite l'utilizzo di *plugin*.
Un plugin è un programma non autonomo che sviluppa funzionalità utili o aggiuntive per il programma principale con il quale interagisce. Per realizzare un nuovo modulo non è necessario conoscere la struttura interna degli altri plugin, ma grazie ad un sistema di API (Application Programming Interface) è possibile trattarli come delle *black-box*. In questo modo anche degli utenti esterni possono creare nuovi plugin ed espandere le funzionalità di PariPari.

Alla base di questo sistema di plugin si trova il *Core*. Il suo compito è quello di gestire tutte le risorse e fare in modo che i plugin lavorino correttamente. Il Core si occupa anche di gestire le comunicazioni tra plugin diversi: tutti i messaggi prima di arrivare a destinazione transitano attraverso il Core, il quale si occupa di inoltrare il messaggio al destinatario.
I plugin si dividono in due grossi gruppi: plugin della *cerchia interna*, i quali garantiscono il funzionamento della rete, e plugin della *cerchia esterna*, i quali forniscono i servizi utilizzati dagli utenti.
La cerchia interna comprende:

- Crediti: ha il compito di gestire in modo opportuno la distribuzione delle risorse tra i diversi plugin;
- Connectivity: ha il compito di amministrare l'utilizzo della rete;
- Storage: si occupa della gestione del file system;
- DHT: tabella hash distribuita contenente informazioni sui nodi connessi e le risorse da loro ospitate;

La cerchia esterna è più ampia e comprende:

- Torrent;
- Mulo;
- DistributedStorage;
- Web Server;

- IRC;
- IM;
- GUI;
- . . . ;

1.2 DistributedStorage

Lo scopo di DistributedStorage è quello di fornire all'utente la possibilità di salvare e recuperare i propri file dalla rete. Ovviamente, essendo in un contesto distribuito, il tutto è realizzato utilizzando un'architettura serverless.
Questo tipo di architettura comporta alcuni problemi che devono essere risolti. Il primo è che non essendoci un server al quale possiamo fare affidamento, dobbiamo in qualche modo sapere quali sono i nodi disponibili a ospitare i nostri dati. Per aiutarci a risolvere questo problema viene in nostro soccorso un altro plugin di PariPari: è il plugin DHT a fornirci una lista contenente tutti gli indirizzi necessari ed è sempre DHT, una volta completato il processo di store, a indicarci i nodi che contengono informazioni utili per il corretto recupero del file.
Altri problemi legati all'assenza di server sono la garanzia di affidabilità e di privacy. Per evitare che la disconnessione da parte di alcuni nodi o il danneggiamento di parte dei dati possa compromettere il recupero dei file, è necessario utilizzare un efficace sistema di codifica[6]. Se qualche pacchetto fosse perso o danneggiato, tramite un processo di *rigenerazione*[7] potremmo ricreare il file originale e garantire il corretto funzionamento del plugin.
Inoltre, per tutelare la privacy degli utenti ed evitare che i nodi ospitanti i nostri dati possano in qualche modo leggerli o modificarli, il file, prima di essere codificato, viene anche cifrato. In questo modo solo chi è in possesso delle corrette chiavi di lettura e scrittura è in grado di accedere al file.

Le funzioni attualmente disponibili sono le seguenti:

[6] le tecniche di codifica saranno approfondite e discusse nel Capitolo 2
[7] il processo di rigenerazione è strettamente collegato al particolare tipo di codifica utlizzata

- Store: come è facile intuire è il comando che si occupa di salvare i nostri dati. Come parametro deve essere passata anche la data fino a cui li vogliamo conservare;
- Retrieve: per recuperare i nostri file;
- Delete: per rimuovere definitivamente un file dalla rete;
- getFiles: restituisce un elenco contenente i nomi di tutti i nostri file attualmente in rete e una lista dei nodi che ne sono in possesso;

Capitolo 2

Sistemi di codifica

Come è già stato accennato più volte, per il corretto funzionamento del plugin è fondamentale la presenza di un efficace e affidabile sistema di codifica. Ciò che differenzia un codice da un altro è il modo in cui viene aggiunta *ridondanza*. É grazie a questo overhead che siamo in grado di ricostruire il nostro file in presenza di errori.

2.1 Replicazione

Il più intuitivo modo di aggiungere ridondanza è la *replicazione*. Il funzionamento è molto semplice: vengono create delle copie del file e ciascuna copia è inviata ad un diverso nodo della rete. In questo modo, per recuperare il file, è sufficiente che almeno un nodo sia raggiungibile.
Nel caso in cui uno dei nodi contenente una delle copie dovesse disconnettersi, è sufficiente creare una nuova copia e inviarla ad un altro nodo. In questo modo il numero di copie resta invariato. Ovviamente nel caso in cui tutti i nodi dovessero sparire, il nostro file sarebbe irrecuperabile.
Supponendo di replicare il nostro file N volte e che in un dato istante uno qualsiasi degli N nodi ospitanti una delle copie sia ancora disponibile con probabilità μ, allora la probabilità di recupero del file, $P(N)$, è data dalla somma da 1 a N delle distribuzioni binomiali:

$$P(N) = \sum_{i=1}^{N} \binom{N}{i} \mu^i (1-\mu)^{N-i} \tag{2.1}$$

La semplicità con cui possiamo recuperare un file usando questa codifica è controbilanciata da uno spreco dal punto di vista della quantità di dati salvati. Guardando Figura 2.1 è possibile notare come, per valori di μ relativamente piccoli, ad es. $\mu = 0.3$, sia necessario replicare il file almeno 7 volte per avere una probabilità di recupero maggiore del 90%, mentre per valori ancora più piccoli, ad es. $\mu = 0.2$, una replicazione di 10 volte non è sufficiente.

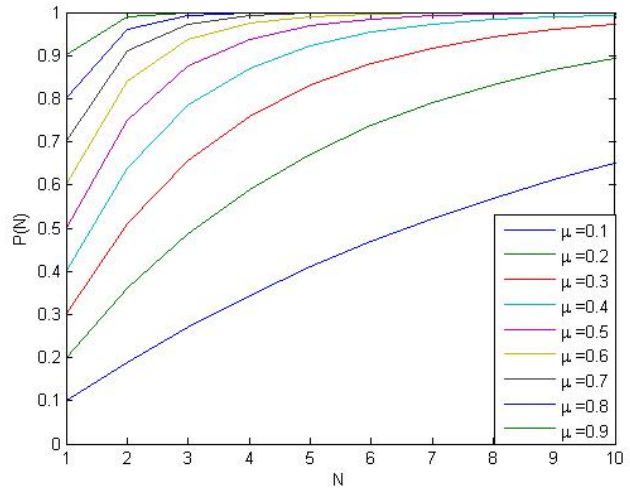

Figura 2.1: probabilità di recupero del file al variare di μ.

Per file di una certa grandezza, una replicazione di 10 o più volte è sicuramente inaccettabile. Basti pensare come un file di 100MB possa comportare una quantità di dati salvata superiore al GB.

In un sistema distribuito è conveniente usare una variante della replicazione in modo da aumentare il numero di nodi coinvolti. Prima di essere replicato, il file viene spezzato in k blocchi, dopodiché ciascun blocco viene replicato N volte. In questo modo i nodi coinvolti saranno kN.
Il calcolo della probabilità in questo caso è più complicato: non c'è un numero fisso che indica il numero di nodi che possiamo perdere senza compromettere il recupero del file. Nel caso migliore siamo in grado di ricostruire il file anche con una perdita di $k(N-1)$ nodi, mentre nel caso peggiore, se tutte le copie di uno stesso blocco dovessero sparire, la perdita di N nodi comporterebbe il fallimento del recupero.

2.2 Erasure Codes

Gli erasure codes sono una classe di codici più elaborata. Il concetto base di un generico *erasure (n,m)-code* è quello di spezzare il file in n blocchi, chiamati *sourceblock*, e da questi costruire altri m blocchi, detti *checkblock*, per un totale di $n + m$ blocchi.

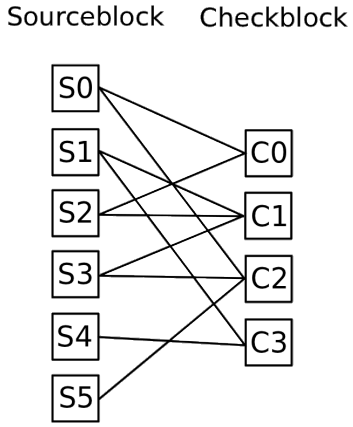

Figura 2.2: Grafo bipartito di un erasure (6,4)-code.

Gli erasure codes sono rappresentanti da grafi bipartiti, come quello in Figura 2.2. A sinistra sono rappresentati gli n sourceblock, mentre a destra ci sono gli m checkblock. Gli archi mostrano quali sourceblock sono stati utilizzati per creare un determinato checkblock.

Gli erasure codes si dividono in due classi: *optimal erasure codes* e *near-optimal erasure codes*.
Per un *optimal erasure code* sono sufficienti n blocchi qualsiasi per ricostruire il file originale. Il numero massimo di nodi che possiamo perdere senza compromettere il sistema è m, e la probabilità di recupero del file è data, come per la 2.1, dalla somma da n a $n + m$ delle distribuzioni binomiali:

$$P(n+m) = \sum_{i=n}^{n+m} \binom{n+m}{i} \mu^i (1-\mu)^{n+m-i} \tag{2.2}$$

Dove μ rappresenta la probabilità che un certo nodo sia ancora disponibile.

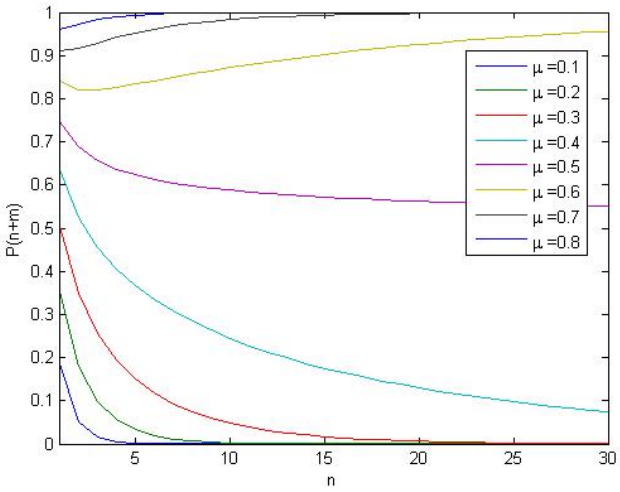

Figura 2.3: Andamento della 2.2 con $m = n$ e μ variabile.

Osservando il grafico di Figura 2.3 è possibile notare come l'andamento della 2.2 sia fortemente legato al valore di μ: per valori minori di 0.6 si ha un andamento decrescente e quindi una diminuzione della probabilità di recupero, mentre per valori maggiori si ha un andamento crescente. Quindi per sistemi con μ elevata, l'utilizzo di un erasure code aumenta sensibilmente la probabilità di recupero rispetto ad una semplice replicazione a blocchi. Si osservi come il caso $n = 1$ coincida con la replicazione a blocchi di parametri: $k = 1$, $N = 2$.

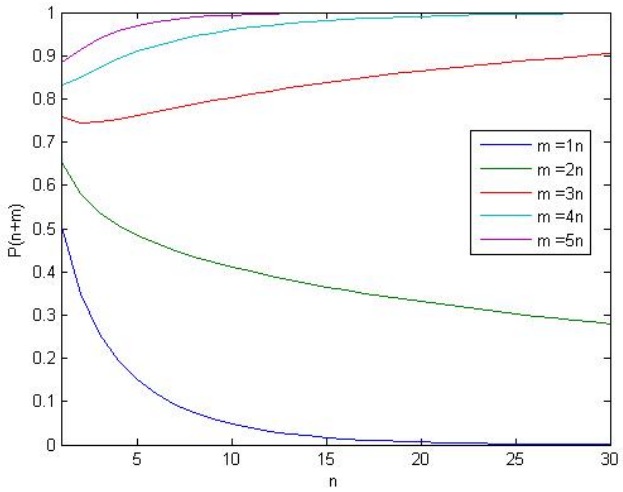

Figura 2.4: Andamento della 2.2 con $\mu = 0.3$ e m variabile.

Aumentando il valore di m è possibile aumentare sensibilmente la probabilità di

recupero anche in caso di valori di μ piccoli. Quindi la scelta dei parametri è fondamentale per l'affidabilità del nostro sistema.
Si faccia attenzione alla differenza tra la n degli erasure codes e la N della replicazione:

- n rappresenta il numero di blocchi in cui viene diviso il file, quindi, fissato m, all'aumentare di n la quantità di dati salvata resta costante;
- N rappresenta quante volte viene replicato il file, per cui l'occupazione di memoria è proporzionale a N;

Quindi il vantaggio degli erasure codes rispetto alla replicazione sta nella possibilità di aumentare a piacere la probabilità di recupero senza penalizzare l'occupazione di memoria.

Data la complessità computazionale di questo tipo di codici, spesso è preferibile usare la seconda classe. I *near-optimal erasure codes* riescono a svolgere le operazioni di codifica e decodifica in tempo lineare. Il prezzo da pagare è la presenza di un *overhead* f, maggiore di 1, tale da rendere necessari fn blocchi per la ricostruzione del file. La formula 2.2 diventa:

$$P(n+m) = \sum_{i=fn}^{n+m} \binom{n+m}{i} \mu^i (1-\mu)^{n+m-i} \tag{2.3}$$

All'interno di *PariPari* si era scelto di usare il *near-optimal erasure code* LDPC (Low-Density-Parity-Check). Un LDPC code è un codice lineare a correzione d'errore in cui i vari checkblock sono ottenuti tramite l'operatore XOR applicato ai sourceblock ad essi legati. Ad es. facendo riferimento alla Figura 2.2:

$$C0 = S0 \oplus S2$$

$$C1 = S1 \oplus S2 \oplus S3$$

$$C2 = S0 \oplus S3 \oplus S5$$

$$C3 = S1 \oplus S4$$

Nell'implementazione usata in *PariPari* si è scelto di fare in modo che i nodi rappresentanti i checkblock abbiano tutti lo stesso grado (numero di archi entranti). Quindi siamo in presenza di un grafo bipartito regolare. Un'altra caratteristica di questo tipo di codici consiste nel fatto che il grado dei checkblock è

inferiore rispetto al numero totale di sourceblock: si può parlare di grafo bipartito sparso regolare.

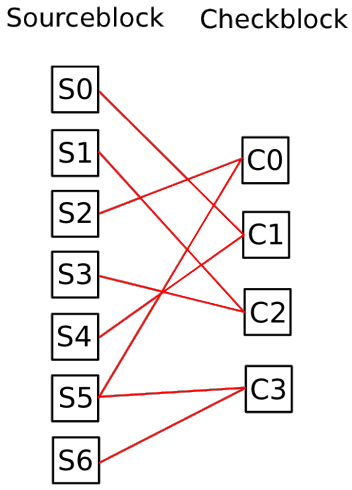

Figura 2.5: Grafo bipartito regolare di un LDPC (7,4)-code.

In alternativa al grafo si può scegliere di rappresentare il codice tramite una matrice, detta *matrice di parità*. Le righe di questa matrice rappresentano i checkblock mentre le colonne rappresentano i sourceblock. L'elemento in posizione (i, j) è un 1 se il sourceblock $(j-1)$-esimo è usato per costruire il checkblock $(i-1)$-esimo, altrimenti è uno 0. La matrice del grafo di Figura 2.4 è la seguente:

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

Poiché anche la matrice di parità di un LDPC code è sparsa, si può usare una rappresentazione più compatta: le righe rappresentano sempre i checkblock, mentre il numero di colonne rappresenta il grado dei checkblock. In posizione (i, j) è rappresentato il numero di uno dei sourceblock usati per costruire il checkblock $(i-1)$-esimo. Sempre in riferimento a Figura 2.4:

$$\begin{bmatrix} 2 & 5 \\ 0 & 4 \\ 1 & 3 \\ 5 & 6 \end{bmatrix}$$

Per recuperare il nostro file sono necessari tutti i sourceblock. Se uno di essi dovesse mancare, potremmo ricostruirlo partendo da uno dei checkblock che sono legati a quel sourceblock. Facendo riferimento al grafo di Figura 2.3, se ad es. dovesse mancare S5, potremmo ricostruirlo partendo da C0:

$$C0 = S2 \oplus S5 \rightarrow S5 = C0 \oplus S2$$

Quindi per ricostruire un blocco mancante è sufficiente applicare nuovamente l'operatore XOR. Se oltre a mancare un sourceblock, cominciano a mancare anche altri sourceblock e checkblock le operazioni di rigenerazione e recupero iniziano a farsi piuttosto dispendiose. Sempre prendendo come esempio il grafo di Figura 2.4, se dovessero mancare i sourceblock S5 e S6, dovremmo prima recuperare S2 e C0 e da essi ricostruire S5, dopodiché da S5 e C3 potremmo risalire a S6:

$$S5 = S2 \oplus C0$$

$$S6 = C3 \oplus S5$$

Ovviamente più il grafo è complicato più saranno numerose le operazioni necessarie a mantenere "vivo" il nostro file.

2.3 Hierarchical Codes

Questo tipo di codifica, presentata da A. Duminuco e E.W. Biersack in [4], vuole essere una via di mezzo tra replicazione e erasure codes in modo da ridurre i costi relativi al mantenimento dei dati senza penalizzare eccesivamente la quantità di dati salvata.
Il grafo di questo codice è ottenuto nel seguente modo:

1. Si scelgano due parametri k_0 e h_0. Si creino k_0 nodi F0,...,F(k_0-1) rappresentanti i frammenti originali in cui è stato spezzato il nostro file. Combinando questi nodi, si creino altri k_0+h_0 nodi B0,...,B(k_0+h_0-1) secondo la seguente regola:

$$B_i = \begin{cases} F_i & 0 \leq i < k_0 \\ \sum_{j=1}^{k} c_{i,j} F_j & k_0 \leq i < k_0 + h_0 \end{cases} \tag{2.4}$$

   dove i $c_{i,j}$ sono coefficienti scelti casualmente tra i valori 0 e 1. L'insieme dei

blocchi B_i così costruiti viene indicato con $G_{d_0,1}$, dove $d_0 = k_0$ è chiamato *combination degree* e indica il grado massimo dei blocchi appena costruiti. Un codice costruito in questo modo è chiamato hierarchical (k_0, h_0)-code;

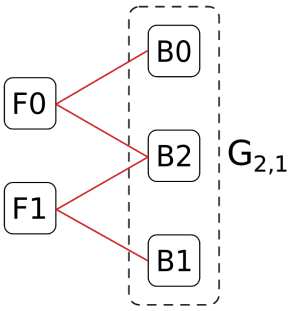

Figura 2.6: Grafo di un hierarchical (2,1)-code.

2. Si scelgano altri due parametri g_1, h_1. Si replichi g_1 volte la struttura del gruppo $G_{d_0,1}$ in modo da ottenere g_1 gruppi: $G_{d_0,1},\ldots,G_{d_0,g_1}$. In questo modo i frammenti originali diventano $g_1 k_0$. Dopodiché si aggiungano altri h_1 nodi ottenuti come combinazione lineare (sempre con coefficienti casuali) dei frammenti originali. In questo modo il nuovo combination degree diventa $d_1 = g_1 k_0$. Tutti i blocchi fanno parte del gruppo $G_{d_1,1}$ il quale corrisponde a un hierarchical (d_1, H_1)-code, con $H_1 = g_1 h_0 + h_1$;

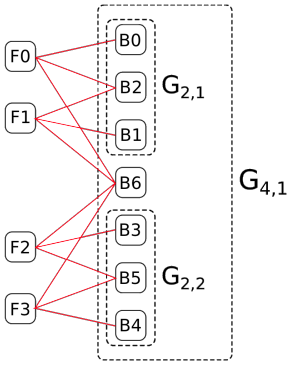

Figura 2.7: Grafo di un hierarchical (4,3)-code ottenuto con i seguenti parametri: $k_0 = 2, h_0 = 1, g_1 = 2, h_1 = 1$.

3. Il punto 2 può essere ripetuto più volte scegliendo nuovi valori dei parametri. Al passo s si scelgano g_s e h_s. Si replichi la struttura di $G_{d_{s-1},1}$ g_s volte e si aggiungano h_s blocchi costruiti come al punto 2, ottenendo così un combination degree pari a $d_s = g_s d_{s-1}$. I blocchi formano il gruppo $G_{d_s,1}$ corrispondente a un hierarchical (d_s, H_s)-code, con $H_s = g_s h_{s-1} + h_s$;

Come nel caso degli erasure codes, il numero di blocchi creati in un hierarchical (k, h)-code è $k + h$.
Per calcolare la probabilità di recupero è necessario introdurre le seguenti proposizioni [4][1]:

Proposizione 1: Si consideri un insieme B^k contenente k blocchi del grafo di un hierarchical (k, h)-code. *Se* i nodi in B^k sono scelti in modo da soddisfare la seguente condizione:

$$|G_{d,i} \cap B^k| \leq d, \ \forall G_{d,i} \tag{2.5}$$

allora i nodi in B^k sono sufficienti a ricostruire i frammenti originali; la condizione 2.5 significa che B^k può contenere al massimo d blocchi scelti da un qualsiasi gruppo $G_{d,i}$,

Proposizione 2: Sia b un nodo ricostruito all'istante t. Indichiamo con $G(b)$ l'insieme dei gruppi che contengono il nodo b e con $R(b)$ l'insieme dei nodi, presenti all'istante $t - 1$, usati per riparare b. *Se* $\forall t$ e $\forall b$, $R(b)$ soddisfa le seguenti condizioni:

$$|G_{d,i} \cap R(b)| \leq d, \ \forall G_{d,i} \tag{2.6}$$

$$\exists G_{d,i} \in G(b) : R(b) \subseteq G_{d,i}, \ |R(b)| = d \tag{2.7}$$

allora il codice non si degrada, cioè, resta valida la proprietà espressa nella Proposizione 1. La condizione 2.6 significa che in $R(b)$ possono esserci al massimo d blocchi presi da $G_{d,i}$, e la condizione 2.7 che deve esistere un gruppo contenuto in $G(b)$ che contenga tutti i blocchi usati per ricostruire b e il numero di questi blocchi deve essere pari a d.

Usando queste proposizioni è possibile calcolare[2] le probabilità $P(d|l)$ e $P(failure|l)$,

[1] Per la dimostrazione si veda [4].
[2] Per il calcolo, abbastanza complicato, si rimanda a [4]

dove l indica il numero di nodi persi e d il numero di nodi necessari, nel caso peggiore, per la ricostruzione di un blocco.
In Figura 2.8 è riportato l'andamento della probabilità per un hierarchical (64,64)-code, costruito usando 6 livelli e i impostando $k_0 = 2, g_s = 2$ e $h_s = 1$ per tutti i livelli, tranne per $h_5 = 2$:

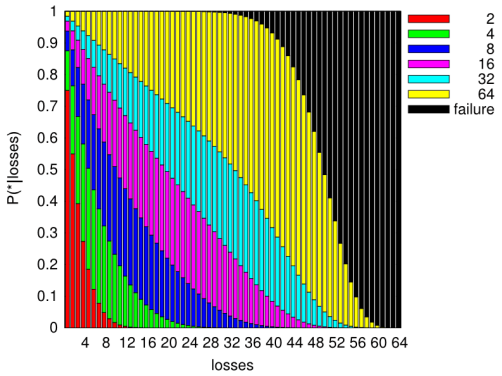

Figura 2.8: Andamento di $P(d|l)$ e $P(failure|l)$ di un hierarchical (64,64)-code in funzione del numero di perdite l.

Osservando il grafico, si può notare come il numero di nodi richiesti varia da 2 a 64, il che comporta una diminuzione dei costi di riparazione rispetto ad un erasure code. Mentre un erasure (64,64)-code non fallisce mai se i nodi persi sono meno di 64, lo hierachical code ha una probabilità di fallimento maggiore di 0 già per 32 nodi mancanti. A prima vista può sembrare che il prezzo da pagare per diminuire i costi sia una diminuzione dell'affidabilità.

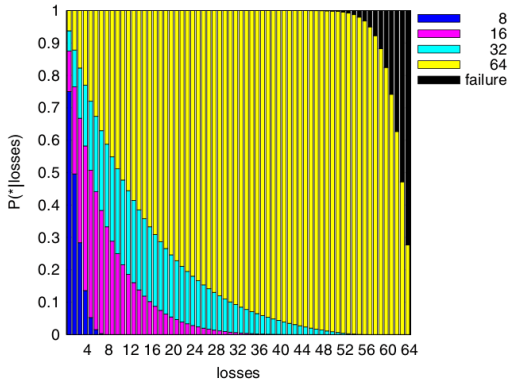

Figura 2.9: Andamento di $P(d|l)$ e $P(failure|l)$ di un hierarchical (64,64)-code in funzione del numero di perdite l.

In Figura 2.9 è mostrato l'andamento della probabilità, sempre per uno hierar-

chical (64,64)-code ma realizzato usando 4 livelli, $k_0 = 8, g_s = 2$ e $h_s = 4$ per tutti i livelli ad eccezione di $h_3 = 8$. Si vede chiaramente come l'andamento della probabilità di uno hierarchical code sia fortemente legato alla scelta dei parametri. La notevole diminuzione della probabilità di fallimento è ottenuta sacrificando in parte i costi di riparazione.
Quindi, giocando sui valori dei parametri, è possibile ottenere prestazioni migliori e affidabilità simile a quella di un erasure code.

Capitolo 3

Implementazione

In questo capitolo presenterò il lavoro da me svolto all'interno del progetto *PariPari*, in particolare all'interno del plugin *DistributedStorage*, per l'implementazione degli Hierarchical Codes.

3.1 Hierarchical Coding Info

Di fondamentale importanza in DistributedStorage è la classe *HierarchicalCodingInfo*[1]. Essa ci permette di creare degli oggetti in grado di contenere tutte le informazioni riguardanti i parametri usati nella codifica, ed è proprio grazie a queste informazioni che siamo in grado di ricostruire i nostri file. Una volta che il nostro file è stato salvato, nella DHT viene salvata una nota contenente tutte queste informazioni e una volta recuperata questa nota, possiamo recuperare il file.

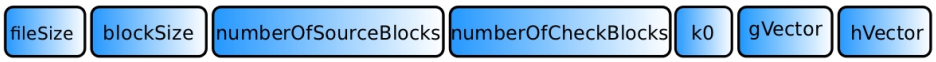

Figura 3.1: Struttura delle hierarchical coding info.

Le informazioni che necessitano di essere memorizzate sono le seguenti:

- fileSize: indica la dimensione originale del file che abbiamo salvato;
- blockSize: dimensione dei blocchi in cui è stato scomposto il file originale;

[1] ex-ErasureCodingInfo

- numberOfSourceBlocks: numero di sourceblock creati a partire dal file originale;

- numberOfCheckBlocks: numero di checkblock ottenuti a partire dai sourceblock;

- k0: coincide col parametro k_0 dello hierarchical code;

- gVector: array contenente tutti i parametri g_i usati per la costruzione del grafo dello hierarchical code;

- hVector: array contenente tutti i parametri h_i usati per la costruzione del grafo dello hierarchical code;

Sfruttando queste informazioni è possibile ricavare la matrice di parità per lo hierarchical code.
La matrice di parità dello hierarchical (4,3)-code di Figura 2.7 è la seguente:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

Le righe rappresentano i blocchi B_i, i quali d'ora in poi saranno indicati come checkblock C_i, mentre le colonne rappresentano i blocchi F_i, d'ora in poi indicati come sourceblock S_i. In posizione (i, j) avremo un 1 se esiste un arco che va da S_j a C_i.
La matrice di uno hierarchical (k,h)-code ha dimensione $(k + h) \times k$. Nominando in modo più furbo i blocchi è possibile ottenere una matrice di dimensioni

ridotte. Prendendo il grafo di Figura 2.7 e modificandolo nel seguente modo:

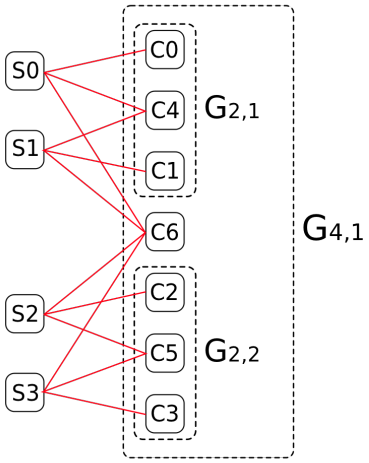

Figura 3.2: Grafo di Figura 2.7 modificato.

si ottiene una matrice del tipo:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

Cioè della struttura:

$$\begin{bmatrix} I_k \\ A \end{bmatrix}$$

dove I_k coincide con la matrice identità di dimensione $k \times k$, mentre A è una matrice $h \times k$. Essendo la struttura della matrice di parità nota a priori, è possibile tralasciare la scrittura di I_k e utilizzare solo i coefficienti presenti nella matrice A. Ed è proprio questo che viene fatto in *PariPari*. Quindi per il grafo di Figura 3.2 la matrice diventa:

$$\begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

Mentre per un codice di più grandi dimensioni, la matrice potrebbe essere:

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Si noti come la sottomatrice di dimensione 3×4 che parte dalla prima riga e dalla prima colonna, sia identica alla matrice 3×4 individuata dalla quarta riga e dalla quinta colonna. Questa ripetizone è dovuta alla struttura ripetitiva del grafo dello hierarchical code.

3.2 Dimensione dei blocchi

La scelta del numero di blocchi e delle loro dimensioni è molto importante. Al momento, l'implementazione dei codici gerarchici prevede che il valore dei parametri k_0, g_i e h_i siano costanti, in particolare $k_0 = 2$, $g_i = 2, h_i = 1 \forall i < j$[2]. Quello che ci rimane da determinare è il valore di j, cioè il numero di livelli in cui vogliamo suddividere il grafo.
Il numero di livelli è strettamente legato al numero di sourceblock: essendo $k_0 = 2, g_i = 2 \forall i$, il numero di sourceblock deve essere necessariamente una potenza di due.
Per un file di dimensione X, con $X > 1MB$, scegliamo come numero di sourceblock la potenza di due più vicina al valore di X. Se X fosse uguale a 21MB, il numero di sourceblock sarebbe 16, cioè un codice a 4 livelli, mentre se X fosse pari a 27MB, i livelli diventerebbero 5 e i sourceblock 32.

[2]Per altre informazioni si veda il Capitolo 4

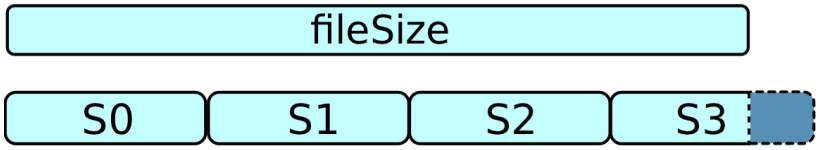

Figura 3.3: Calcolo del numero di blocchi e del padding.

A meno che la dimensione del file non sia un multiplo intero del numero di sourceblock, per determinare la dimensione dei blocchi è necessario aggiungere un piccolo overhead, chiamato padding, al file in modo che i blocchi abbiano tutti le stesse dimensioni. Il padding non è altro che una serie di zeri aggiunti in coda al file. Per determinare quanto deve essere grande è sufficiente calcolare il resto della divisione euclidea tra la dimensione del file e il numero di sourceblock, e la dimensione del padding non sarà altro che la differenza tra il numero di sourceblock e il resto così ottenuto.
Ad es. per un file di 27345231byte e 32 sourceblock:

$$resto = 27345231 mod 32 = 15$$

$$padding = (32 - 15)byte = 17byte$$

Il valore massimo che il padding può assumero è pari a numberOfSourceBlocks-1, quindi è di dimensioni molto piccole rispetto alla dimensione del file.
La scelta del numero di sourceblock influisce sulla dimensione dei blocchi: se è stata scelta la potenza di due superiore alla dimensione del file si avranno blocchi di dimensione inferiore al MB, mentre se è stata scelta la potenza inferiore i blocchi avranno dimensione maggiore.
Ad es. per un file di 27MB e 32 blocchi i blocchi avranno una dimensione di 864KB, mentre per un file di 21MB e 16 sourceblock la dimensione sarà 1344KB.

3.3 Encoding

Il cuore del processo di store all'interno di DistributedStorage è il thread Encoding. Una volta lanciato il comando di store, è proprio questo thread ad occuparsi della creazione dei blocchi e della loro codifica.
Tra i numerosi parametri ricevuti da Encoding, i principali sono: il file che stiamo

salvando, le erasure coding info e un buffer in cui inserire i blocchi.
Per prima cosa, il thread ricava dalle info tutti i dati necessari: numero e dimensione blocchi e matrice di parità. Bisogna fare attenzione alla struttura della matrice: infatti i blocchi ottenuti direttamente dai sourceblock non compaiono al suo interno.

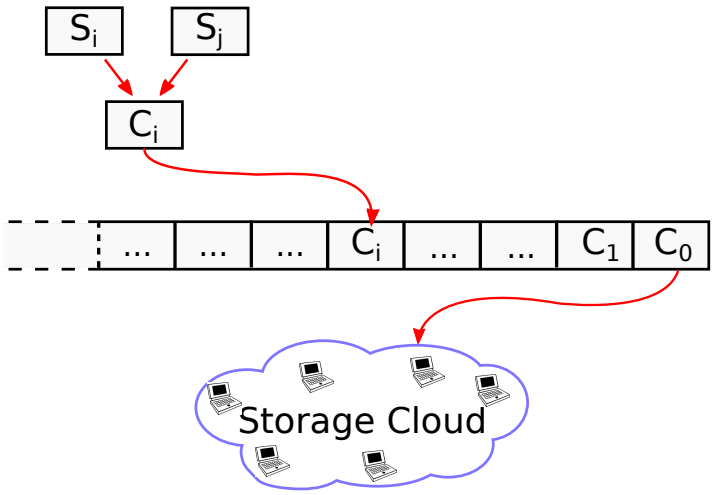

Figura 3.4: Funzionamento di Encoding.

Sono proprio questi blocchi ad essere creati e inviati per primi: all'interno di un array di byte di dimensione pari a quella dei blocchi, viene salvata una porzione del file. Non avendo questi dati bisogno di essere codificati, vengono immediatamente creati i blocchi e inseriti nel buffer. Questa operazione viene ripetuta per tutti i sourceblock e i checkblock che non compaiono nella matrice.
Per i blocchi ottenuti tramite codifica il procedimento è il seguente: leggiamo la riga della matrice corrispondente al checkblock che vogliamo creare. Per ogni cella (i,j) in cui compare un 1 creiamo un array di byte corrispondente al sourceblock S_j, dopodiché per ottenere il checkblock C_i effettuiamo lo XOR tra i sourceblock coinvolti:

$$C_i = \sum_{j=0}^{n-1} b_{i,j} Sj$$

con $n = numberOfSourceBlocks$ e $b_{i,j}$ elemento nella posizione (i,j) della matrice. Una volta ottenuto l'array, si crea il blocco corrispondente e si inserisce nel buffer.
Man mano che il buffer si riempie, i blocchi vengono inviati ai nodi disponibili in

rete. Una volta che il buffer è completamente svuotato, l'operazione di store è conclusa.

Capitolo 4

Conclusioni

In questo documento è stato presentato il funzionamento degli *Hierarchical Codes* e della loro implementazione in *PariPari*. Si è anche fatto un confronto con altri tipi di codifica, replicazione e erasure codes, e si è visto come scegliendo opportunamente i parametri, i codici gerarchici siano in grado di garantire un livello di affidabilità simile a quello degli erasure codes ma con prestazioni migliori.
Inoltre nell'implementazione in *PariPari* vengono anche salvati i sourceblock, mentre nella versione originale [4] il loro salvataggio non è previsto. Questo comporta un aumento di spazio occupato di circa un fattore 2,6. Usando una libreria di compressione, ad es. Snappy, è possibile ridurre notevolmente l'occupazione di spazio e ottenere valori simili a quelli degli erasure codes. A questo riguardo sono necessarie ulteriori analisi e la compressione sarà uno dei prossimi obiettivi all'interno del gruppo DistributedStorage.
Altre analisi sono anche necessarie per poter stabilire caso per caso quali sono i parametri migliori per il nostro codice. Infatti, attualmente i parametri sono definiti a priori e uguali per tutti i file, in particolare $k_0 = 2, g_i = 2, h_i = 1$. Abbiamo già discusso di come questa scelta sia fondamentale per garantire un alto grado di affidabilità.

In questa tesi è stata affrontata solo la parte riguardante lo store, per quanto riguarda retrieve e rigenerazione si veda la tesi di Nicola Corso[7].

Si ringraziano Enrico, per l'aiuto e la pazienza, e Sebastiano, per il supporto matematico :)

Bibliografia

[1] Wikipedia contributors, "Codifica di canale" *Wikipedia, The Free Encyclopedia*, `http://it.wikipedia.org/wiki/Codifica_di_canale` (in data 28 giugno 2012).

[2] Wikipedia contributors, "Erasure code", *Wikipedia, The Free Encyclopedia*, `http://en.wikipedia.org/wiki/Erasure_code` (in data 28 giugno 2012).

[3] Wikipedia contributors, "Kademlia", *Wikipedia, The Free Encyclopedia*, `http://it.wikipedia.org/wiki/Kademlia` (in data 28 giugno 2012).

[4] Duminuco, A., Biersack, E. W., 2010 *Hierarchical codes: A flexible trade-off for erasure codes in peer-to-peer storage systems*, Peer-to-Peer Networking and Applications 3(1), 52-66

[5] Benvenuto, N., Zorzi, M., 2011,*Principles of Communication Networks and Systems*,1° ed.: Wiley.

[6] "Snappy a fast compressor/decompressor", *Snappy a fast compressor/-decompressor*, `http://code.google.com/p/snappy/` (in data 16 luglio 2012).

[7] Corso, N., 2012, *DistributedStorage: Codici Gerarchici*, Dipartimento di Ingegneria dell'Informazione, Università di Padova

[8] Rodrigues, R., Liskov, B., 2005, *High Availability in DHTs: Erasure Coding vs. Replication*, Lecture Notes in Computer Science, Volume 3640/2005, 226-239

[9] Shokrollahi, A., 2004, *LDPC Codes: An Introduction*, Progress in Computer Science and Applied Logic, Volume 23, Part 1, 85-110

[10] Wikipedia contributors, "Replication (computing)", *Wikipedia, The Free Encyclopedia*, `http://en.wikipedia.org/wiki/Replication_(computing)` (in data 16 luglio 2012).