Università degli Studi di Padova
Facoltà di Ingegneria

Finito di scrivere il giorno 22 luglio 2012 utilizzando LaTeX 2ε

UNIVERSITÀ DEGLI STUDI DI PADOVA
FACOLTÀ DI INGEGNERIA

—

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

—

TESI DI LAUREA TRIENNALE IN INGEGNERIA
DELL'INFORMAZIONE

DISTRIBUTED STORAGE: CODICI GERARCHICI

RELATORE: CH.MO PROF. ENOCH PESERICO
CORRELATORE: ING. MICHELE BONAZZA

LAUREANDO: NICOLA CORSO

ANNO ACCADEMICO 2011-2012

A Lorenzo e Ilaria...

“ Gli eroi son tutti giovani e belli”

Francesco Guccini, La locomotiva

Indice

Sommario

In un sistema di storage distribuito la necessità principale per garantire un buon funzionamento consiste nell'avere un'alta probabilità nel recupero dei dati salvati nel cloud. Nel gruppo *DistributedStorage* del progetto *PariPari* mi sono occupato proprio di questo aspetto, implementando la gestione dei file nella rete *peer-to-peer* utilizzando dei codici gerarchici che per alcuni aspetti, sono migliori dei codici *LDPC (Low Density Parity Check)* inizialmente utilizzati nel progetto.

Introduzione

Le applicazioni *Desktop* basate sulla rete internet possono essere di due tipologie, in base a come i componenti terminali di una rete interagiscono tra loro. Il primo modello di applicazione è rappresentato dall'architettura *client-server* in cui i terminali della rete hanno funzioni tra loro differenti. Il server ha un ruolo passivo che consiste nell'aspettare delle richieste di servizio dai client e nel servirle spedendo una risposta. Il client è il terminale utilizzato dall'utente ed ha un ruolo attivo, esso spedisce le richieste al server ed attende una sua risposta. In questa tipologia di rete tutte le risorse e i costi da esse derivati sono concentrate nel server; inoltre con l'aumento del numero di client le prestazioni del sistema possono peggiorare, e nel caso peggiore in cui il server si guasti, i servizi e le risorse messe a disposizione da quel particolare host sarebbero inaccessibili. Questo potrebbe addirittura causare il non funzionamento dell'intero sistema. Un esempio di un servizio di questa tipologia è rappresentato dal *WWW* (World Wide Web) in cui il client è il *browser*, quella particolare applicazione con cui navighiamo le pagine web, mentre il server è rappresentato dal computer che risiede all'indirizzo web richiesto, che custodisce le pagine che visualizziamo.

In un modello distribuito, invece, ogni terminale della rete funziona sia come client che come server, per cui ogni nodo può richiedere e scambiare dati con gli altri nodi della rete. In questo modo le risorse non risiederanno più in un computer centrale, ma saranno distribuite tra le varie componenti della rete. I servizi di *file sharing* sono esempi di modelli distribuiti.

I computer sono col tempo diventati molto potenti e utilizzarli come dei semplici client significherebbe sacrificare molte risorse che hanno a disposizione.

Con il termine *peer-to-peer* originalmente si intendeva una qualsiasi comunicazione tra due nodi, come avviene per una conversazione telefonica. In questi termini anche la prima rete *ARPANET* era un sistema peer-to-peer in quanto ogni computer aveva ugual ruolo. Agli inizi degli anni 90 però la rete internet evolse in un sistema client-server, dovuto principalmente agli indirizzi *IP* dinamici degli utenti e alla carenza di applicazioni che si interfacciavano con la rete.

Con *P2P (Peer-To-Peer)* oggi s'intende, invece, tutta quella classe di applicazioni che sfruttano le risorse di ogni nodo nella rete. Essendo l'accesso alle risorse completamente decentralizzato, e lavorando in un ambiente caratterizzato da connessioni instabili e con indirizzi *IP* imprevedibili, i nodi P2P devono operare al di fuori del sistema *DNS* (Domain Name System) e hanno una significante o totale autonomia da server centrali.

Le applicazioni *P2P* possono essere classificate in base al servizio che offrono:

Comunicazione e collaborazione. Sono le applicazioni di chat e instant messaging.

Computazione distribuita. Sono i sistemi per cui i nodi della rete vengono utilizzati per processare dei dati, ovvero condividono la loro potenza di calcolo.

Database. Sono sistemi in cui i nodi vengono organizzati per gestire dati per formare un database.

Applicazioni per la distribuzione di contenuti. Vi fanno parte i programmi di *file sharing*, di *VoP2P* (Voice Over Peer-To-Peer) e *P2PTV*.

Il *file sharing* ha rappresentato negli anni il *P2P* per eccellenza. Esso prevede la possibilità di condividere file multimediali, che vengono registrati attraverso alcune proprietà come titolo, artista, data o formato, permettendo una ricerca e un download attraverso questi parametri.

Due esempi importanti di *file sharing* sono:

- *Napster* (1999-2001) fu un'applicazione molto popolare. La sua architettura non era completamente decentralizzata in quanto si basava su una ricerca iniziale basata su dei server centrali. Solo dopo i clients si connettevano l'un l'altro per lo scambio diretto dei dati. La presenza dei server centrali rappresentò la debolezza principale di questa applicazione che nel luglio 2001 fu chiusa.
- *Gnutella*, *FastTrack* e *BitTorrent* sono dei protocolli per il file sharing caratterizzati dalla totale assenza di server centrali.

Il *VoP2P* comprende tutte quelle applicazioni per la trasmissione della voce tra nodi della rete, ovvero le applicazioni per le conversazioni. La più famosa è certamente *Skype* (2003), un *VoP2P* client che permette chiamate gratuite tra computer desktop, caratterizzata da un'alta qualità della chiamata e dalla presenza dell'*Instant messaging*.
Le applicazioni *P2PTV* riguardano invece lo *stream* video.

PariPari *PariPari* è un progetto accademico che si sta sviluppando da alcuni anni nell'università di Padova. Esso consiste in una rete *peer-to-peer* realizzata completamente in Java™ e caratterizzata dalla totale assenza di servers centrali. Oltre ai classici servizi offerti da una rete *peer-to-peer* quali *file sharing* e *storage distribuito*, *PariPari* si occupa di fornire anche i tipici servizi di Internet quali *IRC* (Internet Relay Chat), *IM* (Instant Messaging), *VoIP* (Voice Over Internet Protocol), *DBMS* (DataBase Management System), *Web Server* e *DNS* (Domain Name System).

In questo elaborato fornirò una descrizione generale del funzionamento di *PariPari*, analizzando più nel dettaglio la parte riguardante lo *storage distribuito*, in particolare sotto l'aspetto dell'affidabilità. Tratterò così i codici a correzione d'errore utilizzati per far fronte al problema dell'imprevedibilità con la quale i terminali si connettono alla rete, focalizzandomi sui codici gerarchici di cui ho contribuito all'implementazione. Infine, presenterò gli eventuali sviluppi futuri che possono portare ad un miglioramento del progetto.

Capitolo 1

PariPari

1.1 Quadro generale

PariPari è una rete *peer-to-peer serverless* completamente realizzata in *Java*. L'intero progetto è stato organizzato con il meccanismo dei *plugin*, per la quale l'avvio della applicazione si basa sul caricamento del *Core*, che rappresenta il vero nucleo e programma principale del nostro *client*. Questo mette a disposizione dell'utente un'interfaccia grafica costituita da una *console* testuale. Ogni servizio che vogliamo poi utilizzare si basa sul caricamento del *plugin* che lo implementa, attraverso il comando

```
add <nome plugin>
```

Lo sviluppo del programma sotto forma di *plugin* porta con sé il vantaggio della modularità. Gli sviluppatori sono infatti stati divisi e assegnati ad un singolo *plugin*; in questo modo vengono facilitate le fasi di progettazione e di programmazione, anche per il fatto che ogni *plugin* mette a disposizione delle *API* (Application Program Interface) che possono venire utilizzate da tutti gli altri.
Va da sè che alcuni *plugin* saranno di fondamentale importanza per il funzionamento dell'applicazione e saranno quei *plugin* che mettono a disposizione le funzioni base senza le quali gli altri moduli non potrebbero funzionare. Si può parlare in questo senso di *plugin* appartenenti alla cerchia interna, ovvero quelli fondamentali, oppure di *plugin* della cerchia esterna. Quest'ultimi sono quei

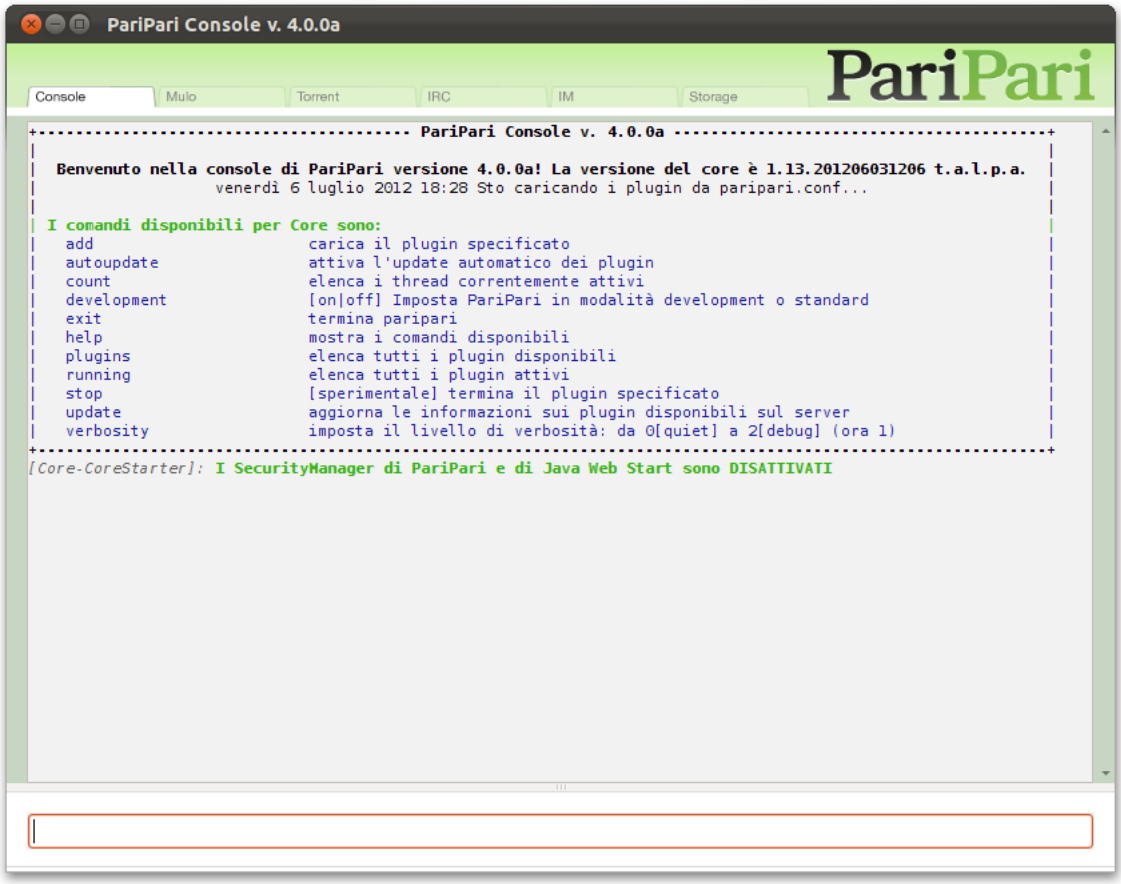

Figura 1.1: La Console di PariPari

plugin che garantiscono i servizi desiderati dall'utente. I *plugin* della cerchia interna di *PariPari* sono:

Core. È il componente principale che si occupa di avviare il programma e di gestire e coordinare le risorse e le varie richieste degli altri plugin .

Connectivity. Ha lo scopo di gestire l'accesso alla rete.

DHT (Distributed Hash Table). È la componente che permette ai nodi nella rete di identificarsi e di scambiarsi informazioni. Grazie alla *DHT* i nodi possono coordinarsi tra loro senza la presenza di *server* centrali.

Local Storage. Si occupa dell'immagazzinamento dei file nell'*hard disk* locale. Questo *plugin* è di fondamentale importanza per tutti i *plugin* che necessitano di salvare file nel disco e dati necessari per gli avvii futuri del *client*.

I *plugin* della cerchia esterna offrono i servizi *Torrent, DBMS, DistributedStorage, DNS, Mulo, Webserver, IM, IRC, VoIP*. Sono tutti servizi già descritti in questo elaborato, tranne *Mulo* che si occupa di fornire un *client* per la rete *eDonkey*.

1.2 Distributed Storage

Un sistema di storage distribuito dà la possibilità all'utente di salvare un file in dispositivi di archiviazione distribuiti in una rete informatica. È proprio questo quello di cui si occupa il *plugin DistributedStorage* di *PariPari*. Al momento il servizio consente agli utenti di salvare e recuperare i propri file nella rete, ma si sta pensando all'implementazione della condivisione di dati per il futuro. Quando il *plugin* viene caricato all'interno del *client* vengono messi a disposizione i nuovi comandi riguardanti lo storage distribuito.
Questi sono:

```
store <file> <scadenza>
```

Serve per salvare il file nella rete, ricevendo come parametro il percorso del file da memorizzare e la data oltre la quale il file verrà definitivamente eliminato dal *cloud*.

```
retrieve <file> <utente> <destinazione>
```

È il comando per recuperare un file precedentemente salvato. I parametri necessari sono il nome del file che si trova in rete, il nome del proprietario del file e la posizione dove avverrà il download.

```
remove <file> <utente>
```

Dato il nome del file e del proprietario rimuove il file dalla rete.

`getFiles`

Permette di ottenere la lista dei file che l'utente ha salvato nella rete di *PariPari*, e la posizione che questi occupano.

Per permettere una collaborazione tra i vari nodi e plugin nella rete il programma si basa su una *DHT* implementata dal *plugin* omonimo. Questa permette lo scambio di informazioni tramite *entry* (o note). Ogni entry è costituita da una o più chiavi in grado di identificarla all'interno della tabella e da vari campi (generalmente stringhe) che rappresentano l'informazione che la nota sulla DHT deve contenere. Caratteristica delle note utilizzate in *DistributedStorage* è la presenza tra i *campi* dell'indirizzo *IP* e della *porta* del nodo che l'ha pubblicata, in modo da facilitare le operazioni che coinvolgono le ricerche delle risorse nella rete.
A titolo d'esempio, quando un nuovo nodo entra nella rete questo pubblica una nota nella DHT con chiave *DistributedStorage* e per campo un oggetto *InetSocketAddress* che contiene l'*IP* e la porta del nuovo arrivato. Per ottenere una lista di tutti i nodi connessi alla rete che offrono il servizio di storage distribuito basterà una ricerca attraverso la chiave *DistributedStorage* e analizzando le entry possiamo ottenere gli indirizzi *IP* dei nodi in questione. Questo è reso disponibile dall'utilizzo delle *API* che il *plugin DHT* mette a disposizione. Ogni nota sarà poi utilizzata per una funzione particolare, determinata

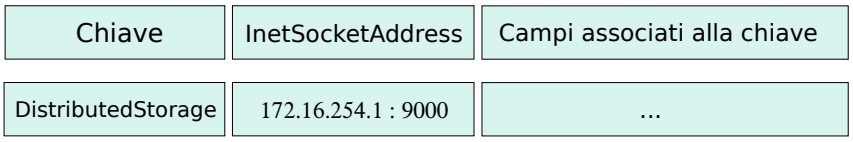

Figura 1.2: Struttura di una nota DHT

dal significato della chiave con cui è stata salvata. Ci saranno delle note che verranno utilizzate per contenere le informazioni sui file salvati in rete e altre usate per gestire alcune operazioni. Per finire, ogni *entry* nella *DHT* ha una durata prestabilita di alcuni minuti, dopo la quale verrà eliminata. Per le note che devono durare nel tempo, il *client* è stato progettato implementando un *Thread* che si occupa di rinnovare le note per mantenerle valide più a lungo.

Non essendovi una struttura a server centrali, e basandosi il salvataggio su nodi nella rete, per la loro imprevidibilità sul tempo di permanenza non è possibile offrire garanzie deterministiche sulla recuperabilità di un file salvato nella rete: il tempo di permanenza di un nodo della rete non può essere previsto a priori. Se, infatti, il nodo in cui abbiamo memorizzato un determinato file si disconnettesse poco dopo, quel file sarebbe inevitabilmente irrecuperabile. Inoltre ogni nodo della rete mette a disposizione una piccola quantità di spazio nel proprio disco rigido; sarebbe impensabile voler memorizzare grandi quantità di dati utilizzando solamente un nodo della rete, in quanto lo priveremmo di una gran quantità di memoria e nel caso in cui il terminale diventasse irreperibile ci sarebbe una perdita irreversibile di una grande quantità di informazione

Per ovviare a questi problemi si è scelto di memorizzare i file nella rete attraverso delle tecniche *FEC* (Forward Error Correction), in particolare tramite i codici gerarchici. Queste cifrature fanno parte delle codifiche di canale studiate nel campo delle telecomunicazioni, ed hanno lo scopo di garantire una trasmissione di dati affidabile su un canale binario rumoroso. Il prezzo da pagare per questa implementazione sarà l'aggiunta di ridondanza nel dato da memorizzare, ma è un prezzo irrisorio se paragonato ai miglioramenti che trarremo da questa soluzione.

1.3 Funzionamento generale del plugin

In questa sezione verrà descritto il funzionamento generale del *plugin DistributedStorage* per dare al lettore una visione d'insieme su come opera lo storage distribuito all'interno di *PariPari*.

Il programma è stato strutturato attraverso l'impiego di vari *Thread*, ognuno dei quali si occupa di gestire una determinata funzione all'interno dell'applicazione, in modo indipendente dagli altri. Quando il *plugin* viene caricato nel *client* viene fatto partire un *Thread* denominato *MainThread* con lo scopo di inizializzare il *plugin*. Questo si occupa di creare o leggere i file di configurazione, di verificare l'esistenza dei database contenenti le chiavi di cifratura *RSA* e *AES* e di lanciare gli altri *Thread* che svolgono le funzioni principali all'interno del *client*. È qui che vengono lanciati i *Thread* che si occupano di

ricevere i messaggi interni ed esterni al *plugin*, ovvero quelli che fanno da server per l'applicazione, *Renewer* che si occupa di mantenere le note sulla *DHT* valide nel tempo e *Regenerator* che si occupa di rigenerare un dato perso dalla rete, di cui parlerò nella sezione dedicata alla rigenerazione. All'interno del *plugin* le richieste sono rappresentate da dei messaggi, così, se un nodo della rete vuole dialogare con un altro nodo, questo avverrà attraverso lo scambio di messaggi che rappresentano sostanzialmente delle richieste che un nodo fa ad un altro nodo. Altri tipi di messaggi sono i messaggi interni che un nodo manda a se stesso per alcune richieste come per esempio la comunicazione tra *plugin* diversi caricati nello stesso *client* o le richieste che vengono prodotte dalla *console* testuale dopo aver inserito un comando valido. Anche la *console* viene inizializzata dal *MainThread*, definendone i comandi accettati dal *plugin*. Ogni comando che viene definito viene processato dal server per i messaggi interni, che lancia un *Thread* che si occupa di eseguire il comando. Anche qui l'utilizzo dei *Thread* è fondamentale per permettere all'utente l'esecuzione di più comandi simultaneamente.

Quando si digita un comando di *store* viene lanciato un *Thread* che si occupa della codifica del file secondo lo schema dettato dai codici gerarchici (Capitolo 2), per cui nella rete verranno salvati vari blocchi e non un singolo file. Ogni file è accompagnato da una *nota globale* nella *DHT*, salvata con una chiave del tipo *<nome file>@<nome proprietario>* che contiene le informazioni fondamentali per permettere la ricostruzione del file dopo la codifica, la data di scadenza del file, quando è stata creata la nota e la firma digitale della nota stessa. Anche ogni blocco relativo al file ha una nota sulla *DHT* che chiameremo *nota particolare*. La chiave per questa *entry* è del tipo *<nome file>-<ID blocco>@<nome proprietario>* e contiene informazioni relative al singolo blocco. Infine i blocchi vengono salvati nei nodi della rete in posizioni casuali.

Il comando di *retrieve* invece fa partire un *Thread* che si occupa di eseguire una ricerca dei blocchi da recuperare per ricostruire il file precedentemente salvato. I dati scaricati vengono salvati in un *buffer* sul quale lavora anche il *Thread Decoding* che si occupa di decodificare i blocchi e ricostruire così il file iniziale (Capitolo 3). Per effettuare la ricerca del file il *plugin* interroga la *DHT* alla ricerca della nota globale ad esso associata. Se questa è presente il

file si trova nella rete ed è possibile cercare i blocchi che lo compongono. Per determinare se un blocco è presente e a che indirizzo lo si può trovare bisogna affidarsi alle note particolari.

Infine il comando per rimuovere il file elimina semplicemente la *nota globale* ad esso relativa. I *client*, che periodicamente controllano la presenza della *entry* per i blocchi che custodiscono, non trovandola eliminano i blocchi presenti nel loro *hard disk* e la loro *nota particolare* dalla *DTH*. In questo modo il file verrà eliminato definitivamente dalla rete.

Per garantire un buon funzionamento del sistema è di fondamentale importanza *Renewer* citato prima. Una *entry* immagazzinata nella *DHT* ha un tempo di vita limitato, di default a cinque minuti, trascorso il quale scompare. Grazie al campo *timeStamp* contenuto in ogni nota, che rappresenta quando la nota è stata creata, il *Thread* in questione calcola la durata residua di ogni nota pubblicata dal *client* e provvede a rigenerarla nel caso ve ne sia bisogno.

Per ovviare alla scomparsa di blocchi nella rete si fa riferimento al processo di rigenerazione. Questo coinvolge diversi *Thread* e si basa su un procedimento distribuito ovvero ogni nodo della rete si occupa di controllare e garantire la presenza dei blocchi "vicini" a quelli immagazzinati nel nodo che ospita il *client*.

Capitolo 2

Codici Gerarchici

2.1 Introduzione

Presentiamo in questa sezione la codifica utilizzata in *PariPari* per garantire un'alta probabilità di recupero di un file memorizzato nella rete. La codifica, come già citato, serve per ovviare al problema dell'imprevedibilità sul comportamento dei nodi del *P2P*, determinata dal tempo di permanenza stocastico di ogni *client* nella rete.
Nella teoria dell'informazione una codifica di canale ha lo scopo di garantire una trasmissione affidabile di un segnale numerico su un canale rumoroso prevedendo l'aggiunta di ridondanza all'informazione da trasmettere. Grazie all'aggiunta di ridondanza il ricevitore può riconoscere un numero limitato di errori e spesso è in grado di correggerli. Si distinguono così due modi principali di operare per contrastare il rumore: la modalità *ARQ* (Automatic Repeat Request) e *FEC* (Forward error correction). La prima prevede la ritrasmissione dell'intera informazione in caso d'errore, la seconda prevede la correzione degli errori da parte del ricevitore. Per ovviare al problema della perdita di informazioni nella rete *P2P* si opera nell'ambito *FEC*, applicando la codifica al file da memorizzare nel *cloud*. In questo modo potremmo essere in grado di recuperare un dato precedentemente salvato anche nel caso in cui alcuni nodi che lo memorizzavano si disconnettessero.
Parleremo in questo ambito di replicazione, degli *erasure codes* e infine di *codici gerarchici*.

2.2 Replicazione

La replicazione come soluzione al problema di reperibilità di un file precedentemente salvato nella rete prevede il salvataggio dello stesso file in più nodi della rete. In questo modo il file sarà recuperabile se almeno un nodo che lo custodisce è online. Definiamo μ la probabilità che un *peer* sia online e S il numero di copie in cui un file viene replicato. Assumendo che ogni nodo abbia ugual probabilità di essere connesso, la probabilità di recupero si calcola come la probabilità che almeno uno degli S nodi sia online:

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

Questa probabilità è data dalla somma (essendo gli eventi indipendenti) di variabili aleatorie binomiali. Una variabile binomiale rappresenta la probabilità che si verifichi un evento i volte su S prove, avendo questo probabilità μ. Per garantire un'alta probabilità del recupero del file il numero delle replicazioni deve essere abbastanza elevato.

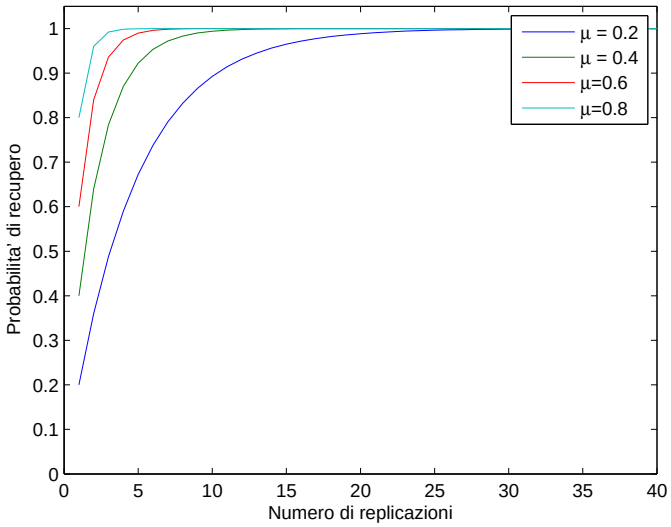

Figura 2.1: Probabilità di recupero in un istante generico di tempo del file, in base al numero di repliche nella rete, per diversi valori di μ. Per piccoli valori di μ il numero di repliche in rete deve essere elevato. Per garantire una probabilità di recupero prossima a 1, con $\mu = 0.2$, servono 25 repliche del file nella rete. Per file grandi ciò non è praticabile.

Introdurre un processo di rigenerazione che si occupa di verificare che in rete vi siano abbastanza repliche dei file e di crearne altre nel caso queste scendessero sotto un numero critico potrebbe essere sufficiente a garantire una probabilità del recupero prossima a 1.
Questa soluzione porta con se però alcune problematiche che di fatto la rendono meno preferibile rispetto alla soluzione da noi adottata. Lo svantaggio principale consiste nella grande quantità di dati aggiuntivi che vengono immessi nella rete; per file di dimensioni elevate la replicazione non è implementabile con buone prestazioni a causa degli elevati tempi di upload necessari per terminare il salvataggio e la rigenerazione di un file, mentre per il recupero di un file avrebbe buone prestazioni in quanto richiederebbe un semplice download da un singolo nodo.

Altri parametri che vengono definiti per caratterizzare la codifica sono il fattore di ridondanza $\beta = S$ e il grado di riparazione $d = 1$ che indica quanti file servono per la rigenerazione. In questo caso essendo una replica per rigenerare un file andato perso basta rimettere in rete una copia.

2.3 Erasure Codes

Un generico *erasure (k,h)-code* si costruisce partendo da un file iniziale e dividendolo in k frammenti. Questi vengono poi processati per produrre un totale di $k + h$ blocchi. In una rete *P2P* ciascun blocco viene poi salvato in un *peer* diverso e il numero massimo di perdite che consentono la ricostruzione dell'informazione è dato da h. Il fattore di ridondanza, che indica quanta informazione andiamo a salvare rispetto a quanta informazione è contenuta nel dato iniziale, è $\beta = \frac{k+h}{k}$.

Assumendo che ogni blocco venga salvato su un *peer* diverso e che sia indipendente da ogni altro blocco, per recuperare il file precedentemente archiviato servono almeno k blocchi, per cui la probabilità di recupero del file è data da:

$$R(k+h) = \sum_{i=k}^{k+h} \binom{k+h}{i} \mu^i (1-\mu)^{(k+h-i)} \tag{2.2}$$

Da notare che (2.1) equivale a (2.2) quando $k = 0$ ($h = S$), essendo la re-

plicazione un caso particolare degli *erasure codes*. Analizzando gli andamenti delle probabilità di recupero di un file si può vedere che negli *erasure codes* questa aumenta all'aumentare dei blocchi ridondanti salvati in rete, ovvero all'aumentare di h (Figura 2.3). Invece non è buona cosa aumentare troppo k se l'affidabilità dei nodi μ non è particolarmente elevata (Figura 2.4). Per finire la replicazione, in termini di affidabilità, se fatta su larga scala è sicuramente migliore, ma non risulta implementabile per file di dimensioni elevate, dato l'enorme traffico di rete che genererebbe. Infatti, a parità di probabilità di recupero tra replicazione e *erasure codes*, la replicazione ha bisogno di una quantità di dati salvati nella rete molto maggiore. Lo scopo principale degli *erasure codes* consiste perciò nel migliorare il traffico di rete, pur garantendo una buona affidabilità al sistema.

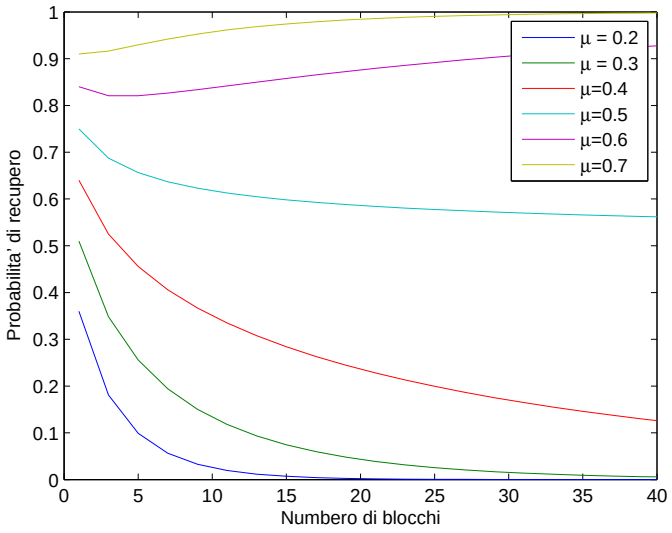

Figura 2.2: Probabilità di recupero per alcuni valori fissati di μ, al variare del numero di blocchi in rete. I grafici fanno riferimento ad *erasure codes* con $k = h$. Si noti che all'aumentare dei blocchi nella rete, aumenta anche k e la probabilità di recupero diminuisce se l'affidabilità dei nodi non è abbastanza elevata. I grafici partono dalla situazione in cui vi sono due blocchi nella rete, il blocco originale e il blocco replicato.

Lo svantaggio principale di questa tipologia di codici risiede nell'alto grado di riparazione. Come risulterà chiaro dall'analisi dell'implementazione si ha $d = k$, mentre per la replicazione si aveva solamente $d = 1$. Questo rende più onerosa la fase di recupero del file e di rigenerazione dei blocchi, che possono

richiedere il download di k blocchi per poter recuperare un solo blocco di interesse. Ciò si traduce in un sostanziale aumento del traffico di rete. Il parametro cruciale nei sistemi di *storage P2P* è perciò rappresentato da d. Ed è proprio questa la ragione per la quale molte reti *P2P* preferiscono mantenere alto il costo in termini di occupazione di memoria a discapito del traffico di rete. Per far ciò il file iniziale viene diviso in frammenti e si applica la replicazione a questi, piuttosto che all'intero file.

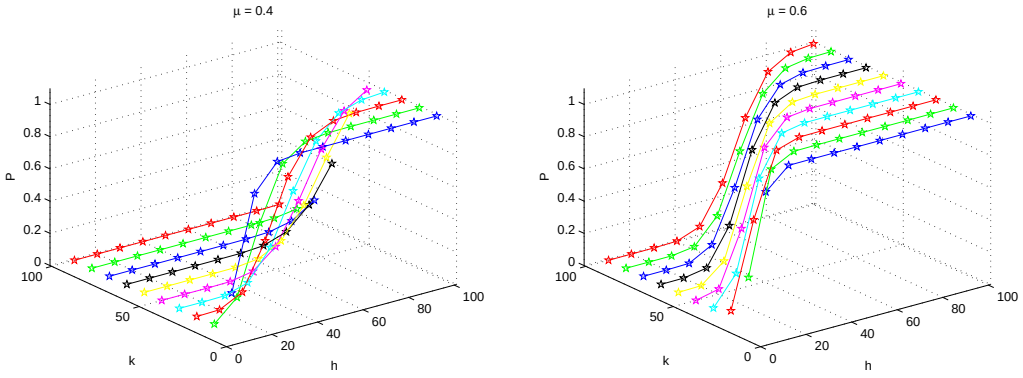

Figura 2.3: La probabilità di recupero di un file aumenta all' aumentare di h, ovvero all'aumentare della ridondanza salvata nella rete. Nei grafici si illustra questa probabilità per diversi valori di k, il primo fa riferimento ad un *peer aviability* $\mu = 0.4$, il secondo a $\mu = 0.6$.

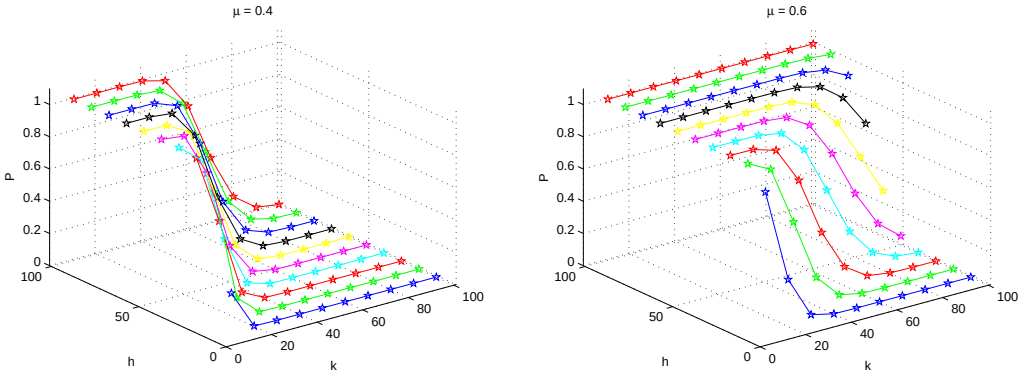

Figura 2.4: La probabilità di recupero peggiora all'aumentare del numero di frammenti iniziali k. Per h piccoli e *peer aviability* basse la ricostruzione del file diviene molto improbabile. I due grafici sono stati tracciati per alcuni valori di h, il primo con $\mu = 0.4$, il secondo con $\mu = 0.6$.

Inizialmente in *PariPari* il plugin *DistributedStorage* si basava sui codici *Low-Density Parity-Check (LDPC)*. I codici *LDPC* fanno parte dei *near-optimal erasure codes*, che rispetto agli *optimal erasure codes* hanno una minore complessità computazionale, ma richiedono $(1+\epsilon)k$ blocchi per la ricostruzione del file, dove $\epsilon > 0$ e può essere reso piccolo a piacere, aumentando però le complessità computazionali. La costruzione di un codice *LDPC* si basa su un grafo bipartito. [1] Dato un codice (k, h), nel primo sottoinsieme si trovano k blocchi, mentre nel secondo sottoinsieme vi sono h blocchi. In *DistributedStorage* i k blocchi sono denominati *source blocks* mentre gli h blocchi prendono il nome di *check blocks*. Dato il file iniziale e fissato un valore di k, i *source blocks* vengono determinati dividendo il file in k parti in modo sequenziale, ciascuna delle quali compone un *source block*. Il valore di k viene deciso in base alla grandezza del file e se la dimensione totale dei *source blocks* eccede la dimensione del file, il (k-1)-esimo *source blocks* avrà i bit finali a 0.

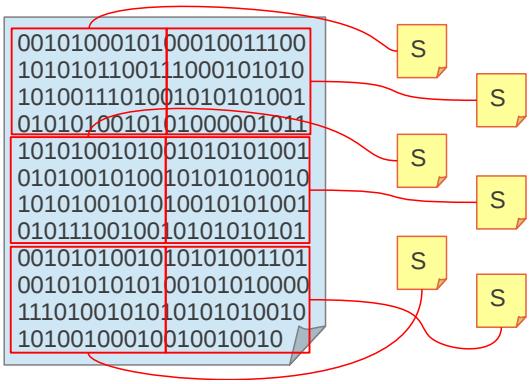

Figura 2.5: Creazione dei *source blocks* dal file da memorizzare

Consideriamo ora un nuovo parametro N *(Neighbors)* che indica quanti rami nel grafo bipartito collegano ogni *check block* e aggiungiamo nel grafo N rami che partono da ogni *check block* e che terminano in un *source block*. Una volta determinato il grafo, ogni *check block* si ottiene applicando la funzione logica *XOR* ai *source block* ad esso collegato.

[1] un grafo bipartito è un grafo tale che l'insieme dei suoi vertici si può partizionare in due sottoinsiemi tali che ogni vertice di una di queste due parti è collegato solo a vertici dell'altra.

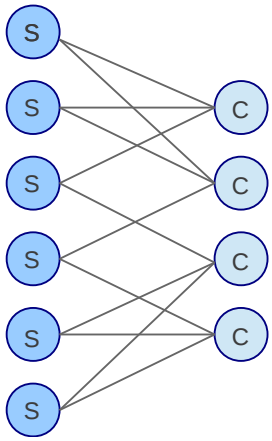

Figura 2.6: Esempio di grafo per la costruzione dei *check blocks* di un codice $(6, 4)$

In questo modo il recupero dei file avviene quando tutti i *source block* sono stati recuperati, e nel caso di perdita di uno di questi, recuperando i *check block* ad esso collegato è possibile la sua ricostruzione.

2.4 Codici gerarchici

Abbiamo visto che negli *erasure codes* lineari il grado di riparazione non può essere minore di k. Se prendiamo un numero qualsiasi di blocchi più piccolo di k non siamo in grado infatti di ricostruire il file archiviato inizialmente. Dividendo il file in blocchi e replicandoli in rete si ottiene un codice $(k, (S-1)k)$ dove S è il numero di repliche per blocco. In questo secondo caso $d = 1$, ma solamente un sottoinsieme dei blocchi salvati in rete è in grado di generare il file memorizzato[2]. I due casi citati rappresentano il caso limite per il grado di riparazione. Verranno presentati in questa sezione i *codici gerarchici* che permettono al grado di riparazione di spaziare tra i valori $1 \leq d \leq k$. L'utilizzo di un codice lineare casuale con $d < k$ pone due difficoltà principali:

(i) Non è facile analizzare la probabilità di recupero di un file, come abbiamo fatto per $d = 1$ e $d = k$.

[2]È necessario scaricare almeno una replica di ogni blocco in cui viene diviso il file (Figura 2.5)

(ii) Non c'è un facile criterio per determinare quali blocchi siano necessari per il recupero del file, quindi problemi legati al recupero e alla rigenerazione.

I codici gerarchici cercano di dare una soluzione a questi due problemi, dando un criterio per la costruzione di un grafo, basandosi su alcuni parametri decisi al momento dell'implementazione.

Costruzione del grafo del codice Vediamo come costruire il grafo da cui si ottiene la codifica finale, in modo simile a quanto fatto per gli *erasure codes*.

(i) Per prima cosa si scelgono due parametri k_0 e h_0 e si costruisce un codice (k_0, h_0). Se per esempio si fissa $k_0 = 2$ e $h_0 = 1$ si parte da due blocchi e se ne aggiunge un terzo, proprio come avviene per gli *erasure codes*. Il grafo sarà costituito dai due segmenti originali e dai tre blocchi sopra citati, come riportato in Figura 2.7. I blocchi generati costituiscono un gruppo che denominiamo $G_{d_0,1}$, dove $d_0 = k_0$ è detto *grado di combinazione*.

(ii) Si fissano ora altri due parametri g_1 e h_1 e si replica la struttura di $G_{d_0,1}$ g_1 volte, ottenendo in questo modo g_1 gruppi denominati $G_{d_0,1},\ldots,G_{d_0,g_1}$. Si aggiungono poi altri h_1 blocchi, ottenuti combinando in modo casuale i g_1k_0 frammenti originali. Questi h_1 blocchi hanno un grado di combinazione $d_1 = g_1k_0 = g_1d_0$ e fanno parte dello stesso gruppo denominato $G_{d_1,1}$. Il codice ottenuto è un (d_1, H_1)-code, dove $H_1 = g_1h_0 + h_1$. Nel nostro esempio fissando $g_1 = 2$ e $h_1 = 1$ si ottiene un (4,3)-code (Figura 2.7).

(iii) Si può ora ripetere il punto precedente aggiungendo altri gruppi al codice. Nel generico passo s, scelti due parametri g_s e h_s, si replica la struttura di $G_{d_s-1,1}$ per g_s volte e si aggiungono poi altri h_s blocchi ottenuti combinando casualmente i frammenti iniziali. Tutti i blocchi ottenuti fin'ora hanno grado di riparazione $d_s = g_sd_{s-1}$ e costituiscono il blocco $G_{d_s,1}$. Si è ottenuto in questo modo il grafo per un codice gerarchico (d_s, H_s), dove $H_s = g_sH_{s-1} + h_s$.

Il fattore di ridondanza per un codice gerarchico (k, h) generico è $\beta = \frac{k+h}{k}$ proprio come per un *erasure codes* tradizionale.

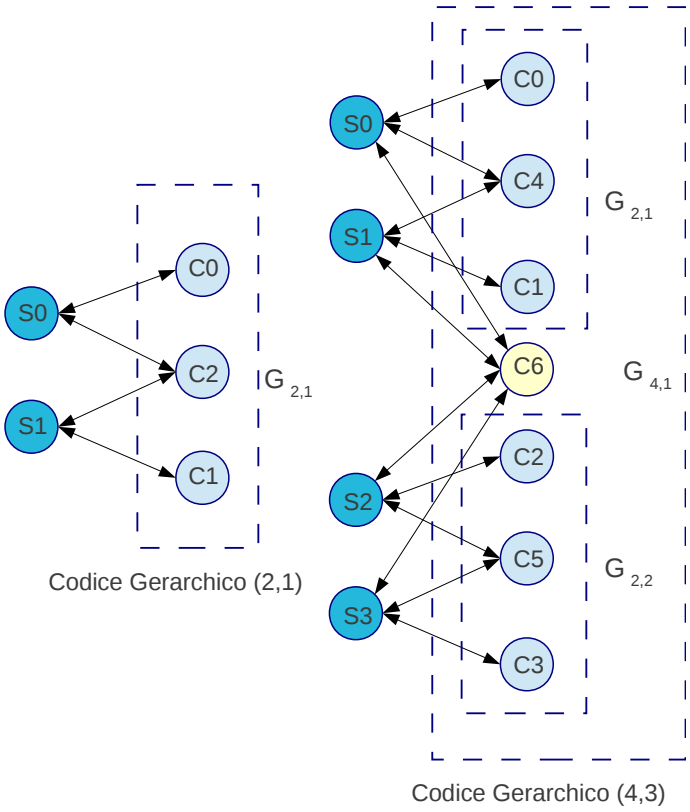

Figura 2.7: Costruzione del grafo relativo ai codici gerarchici

Implementazione della codifica Partendo dal grafo ottenuto applicando il procedimento sopra descritto si realizza la codifica vera e propria dei blocchi da salvare in rete. Per un codice gerarchico (k, h) il numero di frammenti in cui il file viene diviso, che chiameremo ancora *source blocks*, è rappresentato da k e nel grafo di Figura (2.7) sono etichettati con $S_1,\ldots,S_k$. Ogni blocco avrà la stessa dimensione in termini di bit e il procedimento per ottenere i *source blocks* è lo stesso già descritto per gli *erasure codes* (Figura 2.5). Ogni *source blocks* ha degli archi uscenti che terminano in altri vertici. Quest'ultimi prendono il nome di *check blocks* e sono etichettati con C_{ID}, dove ID è l'indice che identifica un qualsiasi *check block*. Fissato inizialmente il contenuto di ogni *check block* di soli bit 0 e della stessa dimensione dei *source blocks*, il contenuto finale si ottiene tramite la funzione logica *XOR* tra il *check block* in questione e ogni *source block* ad esso collegato nel grafo. In questo modo i *check blocks* che hanno un unico ramo entrante rappresentano la replicazione del *source block*

collegato e hanno perciò grado di riparazione $d = 1$, mentre *check blocks* con più rami hanno grado di riparazione pari al numero di rami stesso.

La convenzione utilizzata per numerare i blocchi all'interno di *PariPari* è la seguente:

I *source blocks* sono numerati da 0 a $(k-1)$ in modo progressivo.

I *check blocks* che rappresentano la replicazione di un *source block* hanno ID uguale al numero del *source block* di cui sono la replica.

Gli altri *check blocks* sono numerati a partire da k fino a $(k+h-1)$, in modo progressivo.

La codifica oltre che dal grafo, può essere rappresentata da una *matrice delle adiacenze* in modo equivalente. Nell'implementazione algoritmica all'interno del *plugin DistributedStorage* si fa utilizzo proprio di questa matrice, che è ricostruibile leggendo la *nota globale* del file dalla *DHT*.[3] Ogni colonna di questa matrice corrisponde ad un *source block*, mentre ogni riga ad un *check block*. Essendo i primi $(k-1)$ *check blocks* la replica dei *source blocks* nella matrice non teniamo traccia di questi *check blocks*, in quanto la loro presenza e il loro scopo sono sottintesi. La matrice avrà perciò h righe corrispondenti a $C_k,\ldots,C_{k+h-1}$. Se indichiamo con a_{ij} un generico elemento nella matrice presente alla riga i e colonna j, questo vale:

$$a_{ij} = \begin{cases} 0 & \text{se } C_{i+k} \text{ non ha un ramo che lo collega a } S_j \\ 1 & \text{se } C_{i+k} \text{ ha un ramo che lo collega a } S_j \end{cases}$$

Per il grafo dell'esempio precedente la matrice associata è la seguente:

	S_0	S_1	S_2	S_3
C_5	1	1	0	0
C_6	0	0	1	1
C_7	1	1	1	1

[3] All'interno della *nota globale* della *DHT* vi è un campo che contiene un oggetto della classe *HierarchicalCodingInfo*, da cui è possibile ricostruire la matrice e accedere a informazioni basilari sulla codifica come il numero di *source blocks* e *check blocks* o la grandezza totale del file. Questo è necessario per permettere le operazioni di recupero e di rigenerazione del file.

Per esempio, è immediato vedere che C_6 è collegato a S_2 e S_3, ma non a S_0 e S_1. Le operazioni di recupero del file e di rigenerazione dei blocchi mancanti faranno uso proprio di questa matrice.

Capitolo 3

Recupero di un file

In questo capitolo verrà caratterizzato il recupero di un file enunciando alcune proposizioni che ne regolano la possibilità o l'impossibilità della ricostruzione. Seguirà poi l'analisi della probabilità di reperibilità di un file, in base alla codifica gerarchica usata ed al grado di riparazione necessario a tal fine. Infine verrà illustrata l'implementazione del *retrieve* in *PariPari*.

3.1 Caratterizzazione matematica del retrieve

Nel recupero dei file in una rete *P2P* che utilizza i codici gerarchici hanno un ruolo importante i gruppi $G_{d_i,j}$ a cui ogni blocco appartiene. Questi determinano il grado di combinazione di ogni blocco e quindi quanti blocchi servono per ricostruire un determinato *source block*. Mentre negli *erasure codes* (k, h) il download di k blocchi era sufficiente per il recupero del file, cosicché il *retrieve* del file poteva avvenire scaricando k blocchi qualsiasi, per i codici gerarchici c'è la necessità di implementare una ricerca mirata dei blocchi.

Proposizione 1. *Sia B^k un insieme di k blocchi presi dal grafo di un codice gerarchico (k, h). Se i vertici in B^k sono scelti rispettando la condizione $|G_{d,i} \cap B^k| \leq d \; \forall G_{d,i}$ all'interno del codice (il che significa che in B^k ci sono al massimo d blocchi scelti da ciascuno gruppo $G_{d,i}$) allora i blocchi in B^k sono sufficienti a ricostruire i frammenti originali.*

Riprendendo l'esempio di figura 2.7, per ricostruire i frammenti iniziali ci servono $k = 4$ blocchi, di cui al massimo due provenienti da $G_{2,1}$ e da $G_{2,2}$.

Questo è dovuto al fatto che i gruppi in questione hanno un *grado di combinazione* $d = 2$ e qualsiasi combinazione di due blocchi che vi appartengono sono in grado di ricostruire tutti i frammenti iniziali.

Definiamo ora il concetto di *grafo del flusso d'informazione* per un codice gerarchico (k, h), con lo scopo di caratterizzare nel tempo l'evoluzione dei dati memorizzati nella rete. Ci tornerà utile per enunciare la prossima proposizione al fine di caratterizzare il *retrieve* di un file. Ogni vertice di questo grafo rappresenta un blocco di dati allo specifico istante temporale t. A $t = 0$ il grafo contiene solamente i k nodi che rappresentano i frammenti iniziali mentre a $t = 1$ il grafo contiene tutti i $k + h$ blocchi dovuti alla codifica. Per istanti temporali $t > 1$ il grafo consiste di altri $k + h$ blocchi presenti nel sistema al tempo generico t. Le connessioni tra i vertici sono possibili solamente tra nodi che appartengono ad istanti temporali successivi, orientati da t a $t - 1$.

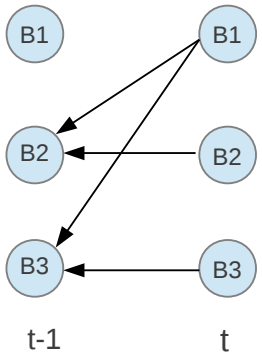

Figura 3.1: Possibile struttura del grafo del flusso dell'informazione

Queste connessioni possono avere tre diversi significati.
1. I nodi all'istante $t = 1$ sono collegati con i nodi dell'istante iniziale $t = 0$ attraverso le relazioni dettate dalla codifica.
2. Un vertice generico all'istante $t - 1$ può essere connesso ad un vertice t. In questo caso il vertice in t non deve essere connesso ad alcun altro vertice di $t - 1$. Questo significa che il nodo è sopravvissuto dall'istante $t - 1$ all'istante t.
3. In alternativa, un blocco generico all'istante $t - 1$ non è connesso a nessun nodo nell'istante temporale successivo. In questo caso vi sarà un nodo in t che

rappresenta il nodo in questione in $t-1$ e che è collegato a d nodi dell'istante precedente. Questo significa che che il nodo in $t-1$ è stato perso e nell'istante successivo è stato ricostruito attraverso d nodi.

Proposizione 2. *Consideriamo un* grafo del flusso d'informazione *di un codice gerarchico al istante temporale t. Sia un nodo b riparato in t. Denotiamo con $G(b)$ la gerarchia dei gruppi che contengono b e con $R(b)$ l'insieme dei nodi in $t-1$ che sono stati combinati per riparare b.*
Se $\forall t$ e $\forall b$, $R(b)$ soddisfa le seguenti condizioni:

$$|G_{d,i} \cap R(b)| \leq d \qquad \forall G_{d,i} \qquad \textit{nel codice} \tag{3.1}$$

e

$$\exists G_{d,i} \in G(b) : R(b) \subseteq G_{d,i}, |R(b)| = d \tag{3.2}$$

allora il codice non si è degradato, ovvero preserva la proprietà del grafo della codifica espressa dalla Proposizione 1.
La condizione (3.1) significa che nell'insieme dei blocchi combinati $R(b)$ ci possono essere un massimo di d blocchi scelti da ciascun gruppo $G_{d,i}$ e la condizione (3.2) significa che deve esistere un gruppo nella gerarchia $G(b)$ che contiene tutti i blocchi combinati e che la loro quantità deve essere uguale al grado di combinazione usato in quel gruppo.

Questa proposizione non solo caratterizza il recupero di un file, ma caratterizza anche la rigenerazione dei blocchi della rete. I due concetti sono strettamente legati tra loro in quanto solamente con la rigenerazione dei blocchi persi si può garantire la reperibilità del file, secondo la Proposizione 1. Nel recupero di un file inoltre, in caso di mancanza di un blocco necessario, la sua ricostruzione può essere effettuata in accordo alla Proposizione 1 con le stesse modalità che si utilizzano per la rigenerazione del blocco mancante. Inoltre la Proposizione 2 ci permette di computare le probabilità di recupero di un file per i codici gerarchici.

3.2 Probabilità di recupero e riparazione

Tramite la Proposizione 2 e analizzando tutte le possibili combinazioni sulle perdite dei blocchi, è possibile calcolare la probabilità $P(d|l)$. Questa indica la probabilità che in presenza di l perdite contemporanee la riparazione di un blocco, nel caso peggiore, richieda un grado d. Da notare che nel caso peggiore, tra gli l blocchi da riparare, si decide di ricostruire il blocco che richiede il grado di riparazione maggiore. Questa rappresenta una stima pessimistica. Nella realtà, il grado di riparazione che si utilizza sarà determinato dalle regole di riparazione impiegate nell'implementazione e esso può perciò essere minore di d. Per il codice gerarchico $(4, 3)$ di Figura 2.7 si ha il seguente risultato:

	$l(perdite)$		
	1	2	3
$P(d = 2\|l)$	0.86	0.42	0
$P(d = 4\|l)$	0.14	0.58	0.77
$P(failure\|l)$	0	0	0.23

Le prime due righe mostrano la probabilità del grado di riparazione avendo l perdite, mentre l'ultima riga rappresenta la probabilità di fallimento nel recupero dei frammenti iniziali. La tabella copre fino a 3 perdite, in quanto per un numero maggiore di perdite le riparazioni non sono mai possibili è la probabilità di fallimento diventa 1. Nella figura 3.2, vengono rappresentate le stesse probabilità, per un codice gerarchico $(64, 64)$, costruito usando 6 livelli e i parametri $k_0 = 2$, $g_s = 2$ e $h_s = 1$ per tutti i livelli, tranne che per l'ultimo dove $h_5 = 2$. Ogni barra nel grafico corrisponde ad una colonna nella tabella, dove le altezze delle sezioni nella barra rappresentano le probabilità di avere un determinato grado di riparazione o di fallimento, dato il numero di perdite.

Da questa figura emerge la proprietà principale che caratterizza i codici gerarchici, ovvero la capacità di diminuire il costo di riparazione in modo significativo. In un tradizionale *erasure code* $(64, 64)$ il grado di riparazione vale sempre 64, per un codice gerarchico $(64, 64)$ essa varia da 2 a 64. Il prezzo da pagare per questa miglioria consiste nel ridurre la probabilità di recupero; un *erasure code* $(64, 64)$ non fallisce mai con un numero di perdite minore di 64, mentre il codice gerarchico analizzato può spesso fallire anche per 32 perdite. Tuttavia la probabilità di recupero può essere migliorata, a discapito del grado

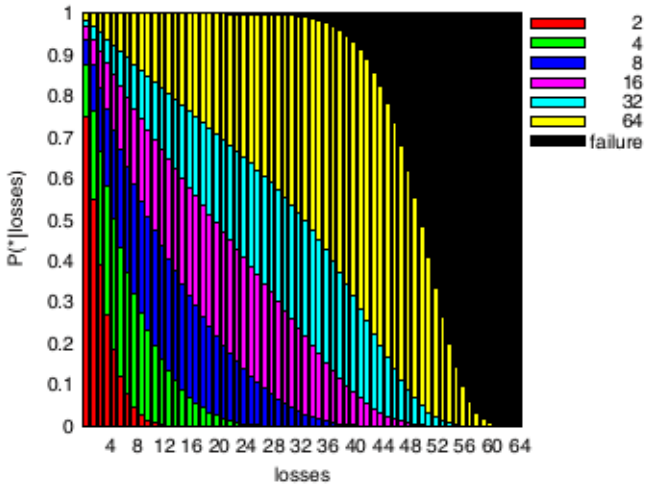

Figura 3.2: Probabilità di recupero e grado di riparazione necessario per un codice gerarchico $(64, 64)$ con 6 livelli e parametri $k_0 = 2$, $g_s = 2$ e $h_s = 1$ per tutti i livelli, tranne che per l'ultimo dove $h_5 = 2$.

di riparazione, scegliendo i parametri k_0, g_s e h_s in modo diverso. Nella figura 3.3 è riportato ancora un codice gerarchico $(64, 64)$ con 4 livelli, $k_0 = 8$, $g_s = 2$ e $hs = 4$ per tutti i livelli, tranne che per l'ultimo dove $h_3 = 8$. Con questo codice le prestazioni in termini di reperibilità sono paragonabili ad un *erasure code*, ma si peggiora notevolmente il grado di riparazione che varia da 8 a 64. Inoltre la regione con grado di riparazione $d = 64$ diventa particolarmente estesa.

I codici gerarchici in questo senso danno al progettatore del sistema la possibilità di determinare il giusto compromesso tra probabilità di recupero e grado di riparazione.

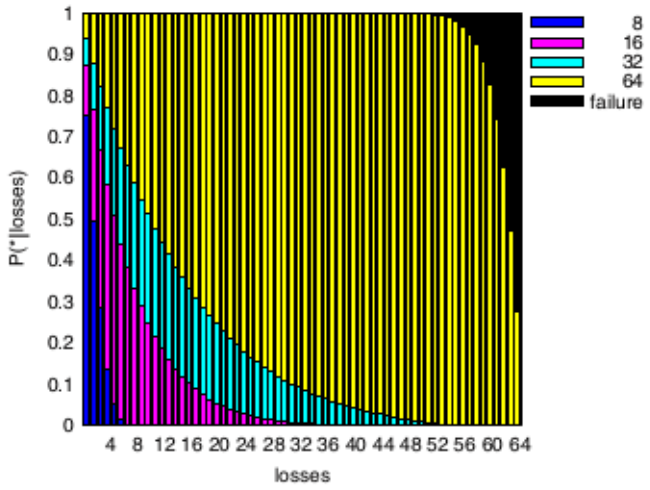

Figura 3.3: Probabilità di recupero e grado di riparazione necessario per un codice gerarchico $(64, 64)$ con 4 livelli e parametri $k_0 = 8$, $g_s = 2$ e $hs = 4$ per tutti i livelli, tranne che per l'ultimo dove $h_3 = 8$.

3.3 Implementazione in PariPari

Per l'implementazione del recupero del file in *PariPari* non si fa utilizzo della nozione di gruppo a cui appartengono i blocchi all'interno del grafo, ma si è scelto di lavorare direttamente sulla matrice delle adiacenze. L'algoritmo di recupero del file funziona in questo modo. Per prima cosa si cerca nella *DHT* la *entry globale* del file, per recuperare l'oggetto *HierarchicalCodingInfo* presente all'interno di ogni nota globale relativa al file. Questo oggetto contiene i campi quali *fileSize*, *numberOfSourceBlocks* e *numberOfCheckBlocks* di fondamentale importanza per poter implementare il recupero. Inoltre la classe *HierarchicalCodingInfo* mette a disposizione tra i metodi pubblici *getMatrix()* il quale fornisce la matrice delle adiacenze utilizzata per codificare il file precedentemente salvato nella rete. A questo punto, si inizia la ricerca dei blocchi di interesse all'interno della rete.

Ricerca dei blocchi Per accedere ad un blocco si cerca la sua *nota particolare* nella *DHT* che, se presente, ci da l'indirizzo *IP* del *client* che custodisce il dato. Grazie ad un messaggio tra i due *client* si recupera così il blocco inte-

ressato.
I blocchi di dati che vengono presi dalla rete vengono salvati in un *Buffer*. Si tenta per prima cosa di scaricare tutti i *source blocks* che costituiscono il file. Se tutti i frammenti sono nella rete e il download va a buon fine, siamo in grado di ricostruire il file iniziale. In caso contrario la ricerca dei *check blocks* migliori per la ricostruzione avviene dando la priorità ai blocchi con grado di combinazione minore. In quest'ottica, per ogni *source block* mancante si cerca il *check block* che ne rappresenta la replica. La realizzazione di questa parte è aiutata dagli identificativi che i *check blocks* hanno

$$S_i = C_i \qquad \forall\ 0 \leq i < numberOfSourceBlocks$$

Se anche dopo questo passaggio non sono stati recuperati tutti i *source blocks* serviranno dei *check blocks* di grado maggiore per la riparazione del file. Durante le due iterazioni precedenti si riempiva una lista con i *source blocks* mancanti, cosicché ora ci si può focalizzare sul loro recupero. Scandendo questa lista di interi che rappresentano l'identificativo dei *source blocks* mancanti, si accede alla matrice alla colonna rappresentata dall'indice del blocco in questione. Si scende poi attraverso la colonna finché non troviamo il primo valore 1. Sia i la riga in cui ci si siamo fermati; tentiamo di scaricare il *check block* identificato come $C_{i+numberOfSourceBlocks}$ [1] e impostiamo l'intera riga della matrice a 0. In questo modo nelle future iterazioni non si tenterà più lo scaricamento dello stesso *check block*. A questo punto si passa all'indice successivo nella lista dei frammenti mancanti e si ripete l'operazione fatta. Una volta raggiunta la fine della lista, si ripete il procedimento da capo, finché non si sono scandite tutte le colonne relative ai *source* mancanti. Così facendo, il buffer verrà popolato man mano dai *check blocks* con grado di riparazione minore, alternando un *check block* relativo ad un *source* mancante ad un *check* relativo ad un altro *source* mancante.

Viene qui riportato lo pseudo-codice per l'algoritmo per la ricerca sopra descritta.

[1] Il *check block* identificato dalla riga 0, corrisponde nella codifica a $C_{numberOfSourceBlocks}$

L : list Of Missing Source Blocks
M : adiancy Matrix
B : buffer
while not visited every row for every element of L **do**
  for $i = 1 \rightarrow size(L)$ **do**
    for $j = 1 \rightarrow height(M)$ **do**
      if $M[j][i] = 1$ **then**
        $B \leftarrow C_{i+weight(M)}$
        $M[j][0 : weight[M]] \leftarrow 0$
        $break$
      end if
    end for
  end for
end while

Decodifica La parte di decodifica viene svolta da un *Thread* che da qui in avanti chiameremo *Decoding*. Questo agisce direttamente sul *Buffer* prelevando i vari blocchi che vengono scaricati e processandoli in modo opportuno. Per prima cosa *Decoding* crea un file di dimensione *fileSize* [2] vuoto, ovvero con tutti i bit impostati a 0. Ogni qualvolta che il *Thread* preleva dal *Buffer* un *source block* ne prende il contenuto e lo va a scrivere nella porzione del file corretta. Per il *source block* S_i il contenuto va scritto a partire dal bit $i \cdot blockSize$, dove *blockSize* è anch'esso contenuto in *HierarchicalCodingInfo* (vedi Figura 2.5). Ogni volta che si esegue la scrittura di un *source block* si va ad incrementare un contatore; solo quando quest'ultimo raggiunge il valore *numberOfSourceBlocks* l'operazione di ricostruzione del file viene completata con successo. Per i *check blocks* che rappresentano una replicazione dei *source* iniziali si attua lo stesso procedimento.

Introduciamo ora il concetto di *blocco di parità* o *parity block*. Questo ha lo scopo di tenere traccia degli *XOR* che si effettuano man mano che si ricevono blocchi con grado di riparazione $d \geq 2$. Nel *Thread* ogni *check block* deve avere un *parity block* associato. Ogni volta che un *check block* viene prelevato dal *Buffer*, il processo di decodifica prevede di effettuare tutti gli *XOR* tra il *check*

[2] ricavato sempre da *HierarchicalCodingInfo* contenuto nella *entry globale* del file

block e i *parity blocks* collegati. Ogni *parity block* poi contiene un contatore che viene inizializzato al numero di blocchi che vi fanno riferimento, cosicché ogni volta che eseguiamo uno *XOR* ne decrementiamo il valore. Oltre al contatore vi tiene una lista dei blocchi che vi sono collegati e quando si fa uno *XOR* si rimuove dalla lista il blocco processato. Quando il contatore raggiunge il valore 1 il *parity block* contiene il dato dell'ultimo blocco rimasto nella lista. L'aggiornamento dei *parity block* avviene anche quando si preleva un *source block* dal *buffer*. Questo procedimento permette di ricostruire dei blocchi man mano che si prelevano i dati dal *buffer*, ricostruendo non solo i *source blocks*, ma anche i *check blocks*, velocizzando la procedura di ricostruzione del file.

Il *Thread* termina quando il file viene ricostruito completamente, oppure quando il processo di ricerca dei blocchi si è concluso e *Decoding* ha finito di eseguire gli *XOR*. In quest'ultimo caso il file non è stato reperito.

Capitolo 4

Rigenerazione

4.1 Introduzione

Un'implementazione di uno *storage* distribuito tramite dei codici a correzione d'errore prevede anche l'introduzione di un processo di rigenerazione. Questo ha lo scopo di ricreare i blocchi che vengono a mancare all'interno della rete; solo in questo modo un file che viene salvato nel *cloud* può essere considerato quasi sempre reperibile.

Un procedimento per la rigenerazione prevede l'individuazione dei blocchi che vengono persi e la loro ricostruzione, che nelle codifiche presentate nel Capitolo 2 avviene tramite lo XOR tra i blocchi collegati nel grafo al blocco da ricostruire. Il dato così generato sarà consegnato ad un nuovo nodo nella rete.

Per i codici gerarchici la Proposizione 2 ci dà un criterio per effettuare la rigenerazione. Se per riparare un nodo b si utilizzano i nodi che indichiamo appartenenti all'insieme di riparazione $R(b)$, questi devono appartenere completamente ad un gruppo $G_{d,i}$ (quindi $|R(b)| = d$) e tutti i blocchi che fanno parte di altri gruppi con grado più piccolo della cardinalità di $R(b)$ non devono appartenervi in numero maggiore del grado di riparazione di quel particolare gruppo. Per il codice gerarchico $(4, 3)$ di Figura 2.7 questo significa che il blocco C_0 può essere riparato in due modi: usando altri due blocchi appartenenti allo stesso gruppo $G_{2,1}$ (per esempio tramite C_1 e C_4) oppure usando quattro blocchi del gruppo $G_{4,1}$ stando attenti a non prenderne più di due da $G_{2,1}$ e

$G_{2,2}$, per esempio tramite C_4, C_6, C_2 e C_5.

Quando si effettua una riparazione di un dato sono possibili varie combinazioni con gradi di riparazioni diversi, in base al blocco che deve essere generato. Il grado che viene utilizzato dipenderà dai blocchi disponibili in rete al momento della riparazione e da come la rigenerazione viene implementata all'interno del sistema.

4.2 La rigenerazione in PariPari

Nelle rete *PariPari* la rigenerazione di un blocco avviene in maniera distribuita tra i nodi. Rispetto ad una soluzione centralizzata in un unico host questa scelta porta alcuni vantaggi. Infatti, se la rigenerazione fosse affidata ad un singolo nodo, tutto il procedimento si baserebbe sulle risorse di quest'ultimo: in particolare inciderebbe la sua velocità di download e la sua potenza di computazione. In questo modo, un terminale con una banda o una potenza di calcolo limitata potrebbe impiegare molto tempo per concludere l'operazione, con il rischio che alcuni blocchi necessari diventino irreperibili dalla rete nel corso del processo. Con la soluzione distribuita invece, a contribuire al mantenimento dei blocchi nella rete partecipano tutti i nodi che ne custodiscono almeno uno. In questo modo tutti i *client* partecipano in modo equo e il processo non sarà più determinato dalle caratteristiche di un solo computer, ma da più calcolatori che mettono a disposizione le loro risorse. Così facendo le prestazioni temporali sono in media migliori. A sfavore dell'implementazione distribuita c'è però l'aumento del traffico di rete, dovuto allo scambio di più informazioni tra i nodi per coordinarsi. La soluzione adottata per attuare la decentralizzazione del processo di rigenerazione prevede che ogni *client* si occupi della rigenerazione dei blocchi che nel grafo della codifica sono collegati al blocco custodito. In questo modo solo una piccola quantità di terminali si occupa di controllare un dato blocco.

L'algoritmo Un *client* che custodisce un blocco di un file ha il compito di garantire la presenza dei blocchi ad esso vicini, ovvero che sono ad esso collegati nel grafo della codifica. Quando un blocco viene a mancare, un *client* della rete che vigila su di esso si accorge per primo della mancanza. Questo

client ha il compito di gestire la rigenerazione e lo chiameremo *manager*. Il *manager* deve per prima cosa dichiararsi responsabile della rigenerazione del blocco, inviandone notifica ad ogni altro client della rigenerazione. Per far ciò si fa utilizzo di una nota sulla *DHT* che ha lo scopo di indicare agli altri nodi che la rigenerazione del blocco è già in corso. Il *manager* sceglie in modo casuale un nodo sulla rete che avrà il compito di custodire il blocco che verrà generato. Questo nodo prende il nome di *regenerator*. Esso aspetterà di ricevere i blocchi necessari per recuperare il dato, eseguirà lo *XOR* tra di loro e pubblicherà la *nota particolare* sulla *DHT* dichiarando l'esistenza di quel blocco e di esserne il possessore. Il nodo *manager* nel frattempo cerca nella rete i blocchi online che servono per la creazione del blocco e tramite un messaggio avvisa i *client* che li custodiscono di inviarne una copia a *regenerator* per essere processati. Se questo procedimento va a buon fine il blocco perso in precedenza viene ricostruito.

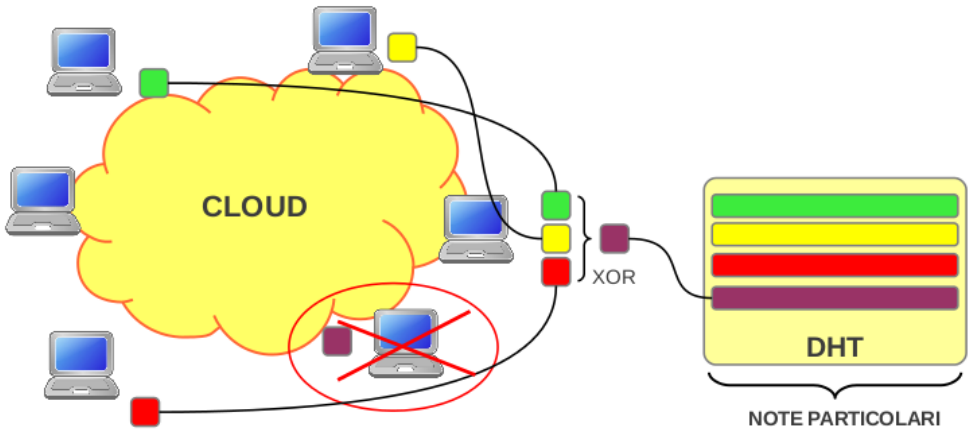

Figura 4.1: Procedimento di rigenerazione utilizzato in *PariPari*. Il nodo custode del blocco viola esce dalla rete. Se i blocchi ad esso collegati sono il blocco giallo, rosso e verde, dopo esser stato scelto il nuovo nodo custode, questi inviano i propri blocchi che vengono processati per rigenerare il blocco perso. Infine viene pubblica la entry particolare nella DHT.

Il grado di riparazione Il grado di riparazione è, come per il recupero di un file, il parametro delicato per la rigenerazione. Esso determina quanti blocchi sono necessari per la ricostruzione del dato perso. Per i codici gerarchici il grado di riparazione varia a seconda della disponibilità dei file in rete. Se il

blocco da riparare è un *source block* o un *check block* con identificativo minore del numero dei *source blocks* conviene provare a rigenerarlo attraverso la sua replica che prevede l'utilizzo di un solo blocco. Nel caso in cui questa non sia presente nella rete conviene tentare di riparare il dato con il minor numero di blocchi, che secondo la Proposizione 2 è pari al grado d del più piccolo gruppo $G_{d,i}$ a cui il blocco da riparare è collegato. Se anche questi blocchi non sono reperibili, si procede con il gruppo di grado successivo.

Per i *check blocks* con identificativo maggiore invece, la ricostruzione deve per forza avvenire tramite i *source blocks* ad esso collegato o tramite i *check blocks* che rappresentano le copie di questi. In questo caso il grado di riparazione coincide al grado del più piccolo gruppo a cui appartiene il blocco da rigenerare.

Se i blocchi da utilizzare per la rigenerazione non sono completamente reperibili, la rigenerazione si arresta e viene rinviata. Nel frattempo si spera che i blocchi mancanti vengano ricostruiti, altrimenti la rigenerazione non andrà a buon fine.

4.3 Implementazione

La realizzazione vera e propria all'interno del progetto avviene tramite alcuni *Thread* con lo scopo di implementare l'algoritmo illustrato precedentemente. Il *Thread RegeneratorStarter* ha lo scopo di inizializzare la rigenerazione. Al momento dell'avvio del *client*, *ReneratorStarter* controlla quali blocchi sono custoditi dal nodo e interrogando la *DHT* controlla la presenza dei nodi ad essi collegati.

Quest'ultimi si trovano attraverso la matrice delle adiacenze associata al file di cui analizziamo i blocchi. I vicini dei blocchi

$$S_i \ e \ C_i \qquad \forall i < numberOfSourceblocks$$

si ottengono andando alla colonna i-esima e scandendo le sue righe. Ogni volta che si trova un 1, significa che il blocco $C_{j+numberOfSourceblocks}$, dove j è il numero di riga, è un vicino del blocco. Per questi blocchi si considera un vicino anche la replica, perciò C_i è vicino di S_i e viceversa.

Per i blocchi

$$C_i \qquad \forall i \geq numberOfSourceblocks$$

si accede invece alla riga $(i - numberOfSourceblocks)$-esima e scandendola si aggiunge al gruppo dei vicini S_j e C_j, dove j è l'indice di colonna, ogni volta che troviamo il valore 1.

Nel caso in cui un blocco non è reperibile viene lanciato un *Thread RegeneratorManager* con lo scopo di gestire la rigenerazione. È questo *Thread* che sceglie in modo casuale il nodo *regenerator*, il custode finale del blocco e nodo che effettuerà gli *XOR* per la riparazione.

Per svolgere il ruolo di *regenerator*, all'avvio del *plugin*, oltre che al *Thread RegeneratorStarter* viene lanciato anche il *Thread Regenerator*. Questo fa da *listener* per l'eventuale messaggio di richiesta di essere il nodo *regenerator* nel processo di rigenerazione che può arrivare ad ogni *client*. Tramite la presenza di un *Buffer* poi, *Regenerator* attende i blocchi da processare dagli altri nodi e man mano che i dati gli arrivano si eseguono gli *XOR* fino a ricostruzione avvenuta. Solo a questo punto *Regenerator* pubblica una *nota particolare* relativa al nuovo blocco sulla *DHT*.

Ritornando al *Thread RegeneratorManager*, oltre che a scegliere il nodo *regenerator* si deve occupare di gestire la ricerca dei blocchi necessari per la rigenerazione, con la condizione di mantenere minimo il grado di riparazione necessario. La ricerca viene eseguita con l'algoritmo di ricerca presentato per il recupero di un file (Capitolo 3), organizzando una lista di liste di blocchi, ovvero una matrice con stesso numero di colonne, ma numero di righe variabili. Le liste dei possibili nodi ricostruttori del blocco perso vengono memorizzate in ordine crescente di colonna: ad indici di colonna maggiori corrisponderanno liste di lunghezza maggiore. Per i *source blocks* o i *check* che ne rappresentano la replica la prima lista conterrà solo l'elemento replica e salendo di posizione si troveranno i blocchi con grado di riparazione crescente. Per i restanti *check blocks* sarà presente un'unica lista contenente i *source blocks* a cui fa riferimento. A questo punto *RegeneratorManager* itera sulle liste di blocchi e, quando scopre una lista con tutti i nodi custodi dei blocchi online, li invita tramite dei messaggi a spedire i blocchi al nodo *regenerator*. Il passo finale della rigenerazione, come già detto, viene svolto da *Regenerator*.

Capitolo 5

Conclusioni e sviluppi futuri

In questo elaborato è stato presentata l'implementazione di un sistema di salvataggio di file distribuito in una rete *P2P*. Di fondamentale importanza per il sistema è garantire una probabilità di recupero di un file che sia il più possibile prossima a 1. La replicazione dell'intero file non è implementabile per file di dimensioni medio-grandi, in quanto genererebbe all'interno della rete un traffico dati troppo elevato. Per ovviare a questo problema si fa uso di codici a correzione d'errore e si è implementato un processo di rigenerazione. Un generico *erasure code* riduce drasticamente la quantità di dati immessa nella rete rispetto ad un processo di replicazione, ma introduce un nuovo parametro cruciale al sistema: il grado di riparazione. Nell'alternativa di dividere un file in blocchi e di applicare la replicazione a questi, il grado di riparazione diventa ottimo, ma la probabilità di recupero rispetto ad un *erasure code* peggiora e per aumentarla si devono aumentare le replicazioni all'interno della rete, occupando grandi moli di spazio su disco nei *client* connessi. I codici gerarchici rappresentano un *trade-off* tra queste due metodologie di codici a correzione d'errore. Tramite i parametri che vengono utilizzati per la costruzione del codice si può variare la probabilità di recupero, il numero di blocchi salvati nella rete e il grado di riparazione. La probabilità di recupero aumenta con l'aumentare dei blocchi nella codifica e con l'aumentare del grado di riparazione massimo, ma in presenza di un buon processo di rigenerazione può essere considerata molto vicina a 1 anche con codifiche gerarchiche più deboli. Il vero vantaggio dei codici gerarchici sta nella riduzione del grado di riparazione che

essi introducono rispetto agli *erasure codes*. Questa codifica salva generalmente più dati rispetto agli *erasure*, ma porta con sè il vantaggio di un recupero dei file e di una rigenerazione più veloci e mirati. Soprattutto per quanto riguarda il recupero, che è il comando "attivo" dell'utente, si deve garantire una buona velocità. Da come è stata implementata la codifica, rispetto agli *erasure codes* è più probabile che la ricostruzione utilizzi un grado di riparazione molto più piccolo rispetto a quello necessario per un *erasure code*, e quindi si ha una maggior velocità di *retrieve*. Ciò è garantito anche dal processo di rigenerazione, più snello per la maggior parte dei blocchi rispetto agli *erasure*. Per alcuni blocchi però la rigenerazione richiede un grado di riparazione molto elevato, e sono i blocchi appartenenti ai gruppi con grado gerarchico maggiore. Tuttavia attraverso i parametri della codifica il numero massimo di livelli gerarchici può essere modificato a piacere. Inoltre la rigenerazione è un procedimento "passivo" invisibile per l'utente. Essa può causare dei lievi picchi di traffico di rete da parte dei nodi in cui si rigenera un blocco con grado di riparazione elevato, ma sono poco duraturi, in quanto una volta ricreato il blocco il traffico cessa e questi nodi non sono in genere necessari per la ricostruzione di un file o di un altro blocco, in quanto sono scelti come ultime possibilità.

Dato il numero dei parametri che entrano in gioco e le diverse conseguenze in termini di probabilità di recupero, di grado di riparazione e di quantità di dati da salvare in rete, sarà necessario uno studio su come debbano essere scelti all'interno della rete *PariPari*. Ad incidere sarà soprattutto il numero medio dei *client* connessi alla rete e la grandezza dei file. Dall'analisi fatta in questo elaborato si può già dedurre che il numero di frammenti iniziali non deve essere troppo grande, in quanto deteriora la probabilità del recupero e non conviene aggiungere troppi livelli gerarchici alla codifica, in quanto penalizzerebbe la rigenerazione dei blocchi di grado elevato.

Per quanto riguarda il *retrieve*, questo può essere migliorato introducendo il download simultaneo di più blocchi. Nella realizzazione attuale infatti tra un blocco e il successivo nell'ordine da scaricare c'è l'attesa che il dowload del primo termini. Per *client* con una elevata banda di download questo può rappresentare una limitazione e l'introduzione di scaricamenti multipli potrebbe migliorare le prestazioni.

Nella rigenerazione invece bisogna migliorare la gestione del controllo dei

blocchi da rigenerare. Infatti, i *client* che custodiscono blocchi di grado gerarchico elevato hanno il dovere di garantire la presenza in rete di un numero di blocchi maggiore rispetto ai *client* che custodiscono blocchi di gradi minori. Questo potrebbe rappresentare un problema nelle codifiche in cui vi sono molti livelli gerarchici, mentre per codifiche più semplici ciò non costituisce un grave problema.

Bibliografia

[1] *Hierarchical codes: A flexible trade-off for erasure codes in peer-to-peer storage systems*, Alessandro Duminuco - Ernst W. Biersack, Published online: 22 April 2009

[2] Nevio Benveuto, Michele Zorzi, *Principles of Communications Networks and Systems*, Wiley, 2011

[3] Andrea Mambrini, *Erasure codes for distributed storage systems*, Thesis, A.Y.2008/2009

[4] Robert G.Gallager, *Low-Density Parity-Check Codes*, 1963

[5] Michael T.Goodrich, Roberto Tamassia, *Data Structures and Algorithms in Java*, Wiley, 4th Edition.