POLITECNICO DI MILANO

Facoltà di Ingegneria dell'Informazione

Corso di laurea in Ingegneria Informatica

FILE SYSTEM A TAG: L'EVOLUZIONE ARRIVA DA INTERNET

Relatore: Prof. Fabio Alberto Schreiber

Correlatore: Ing. Davide Eynard

Elaborato di Laurea di:

Daniele Zanni Matr. n° 668432

Davide Zoni Matr. n° 670473

Anno accademico 2006 – 2007

Prefazione

Ben venga il caos, perché l'ordine non ha funzionato.
(Karl Kraus)

In ogni caos c'è un cosmo, in ogni disordine un ordine segreto.
(Carl Gustav Jung)

Realizzare una elaborato di tesi è certamente un'esperienza preziosa e significativa per lo studente che termina un ciclo di studi. Scelta importante è l'argomento che si vuole trattare, ma forse ancora più importanti sono le motivazioni che portano lo studente a sceglierlo. Anche nel nostro caso, si è fatta molta attenzione prima di decidere che il progetto che è stato realizzato, di cui questa prefazione è solo l'ultimo atto, fosse realmente quello adatto alle nostre esigenze e corrispondesse ai nostri interessi.

La necessità di affrontare alcuni corsi di progetto per portare a termine i primi tre anni della carriera universitaria ci ha fatto incontrare il concetto di "file system a tag" che ci ha subito affascinati. La nostra passione per le operazioni che costituiscono il cuore del compilatore e quella per le nuove frontiere del Web si sono potute così incontrare e hanno dato vita, con alti e bassi, con momenti di fervente attività e momenti di calma piatta, a un risultato a nostro avviso soddisfacente, oltre che per il prodotto in sè, per le conoscenze acquisite. Soddisfacente anche e soprattutto per averci dato la possibilità di imparare meglio a lavorare in coppia, dividendoci di volta in volta il lavoro, condividendo scelte, imparando a giungere talvolta a compromessi che seguissero le esigenze di entrambi.

Vanno a questo punto ringraziate tutte le persone che in qualche modo hanno contribuito alla realizzazione di questa tesi, dai consigli tecnici ai supporti morali. Un

ringraziamento all'Ing. Eynard per i preziosi consigli, il tempo concessoci, anche negli ultimi giorni, e soprattutto la pazienza dimostrata. Un grazie al Prof. Schreiber per aver acconsentito a portarci in tesi come relatore, facendo anche in modo che questa non subisse ritardi, nonostante l'iter burrascoso che ci ha portati a questo giorno. Grazie anche alla dott.ssa Petricca per l'attenzione e l'interesse mostrato alla nostra situazione.

Un ringraziamento inoltre a parenti e amici per averci sostenuto nei momenti più difficili, incoraggiati a ogni barlume di speranza e per aver gioito con noi nei momenti più felici.

Daniele Zanni

Davide Zoni

Indice generale

Prefazione....2
Introduzione....5
1 Perché un file system virtuale?....7
1.1 Sempre più informazioni....7
1.2 File e File System....7
1.3 I limiti dei file system di tipo gerarchico....9
1.4 Un File System Virtuale....10
2 Web e Folksonomie....14
2.1 Tassonomia vs Folksonomia....14
2.2 Il WWW e i tag....16
2.3 Un File System a Tag....20
2.4 I file system a tag: un primo approccio....24
3 Implementazione astratta di un file system a tag....26
3.1 Le componenti del modello teorico....26
3.2 Le operazioni di un file system a tag....27
3.3 Mappare le operazioni di un'interfaccia per file system gerarchici su un file system a tag...28
4 Progetto e implementazione....30
4.1 La nascita di TaggyFS....30
4.2 Virtual file system, SO, linguaggio di programmazione: Fuse detta legge....30
4.2.1 Come opera Fuse....32
4.3 Hibernate e il nuovo approccio per la memorizzazione dei metadati....34
4.3.1 Come opera Hibernate....36
4.4 File system gerarchico nascosto: lui c'è!....37
4.5 La fase embrionale di TaggyFS....37
4.5.1 Il SO non sa che è cambiata la semantica delle sue chiamate....38
4.5.2 L'integrazione con le interfacce esistenti....38
4.6 Evoluzione del progetto in itinere....39
4.7 Il volto di TaggyFS....40
4.7.1 L'idea delle operazioni di TaggyFS....42
4.7.2 Caratteristiche Avanzate....44
4.8 I modelli: uno sguardo al cuore di TaggyFS....46
4.8.1 I Package....46
4.8.2 Le classi....48
5 Test....50
5.1 Log e screenshot....50
5.1.1 Creazione di una directory (creazione di un tag)....50
5.1.2 Rinomina....51
5.1.3 Eliminazione di una directory (eliminazione di un tag)....51
5.1.4 Copia file dall'esterno....52
5.1.5 Cancellazione di un file....53
5.1.6 Copia di un file interno a TaggyFS (aggiunta di un tag)....54
5.1.7 Eliminazione di un tag foto tramite il cestino (eliminazione di un tag da un file)......56
5.1.8 Rinomina di un file....57
6 Conclusioni....58
Bibliografia....60

Introduzione

La struttura del file system rappresenta, in un sistema operativo, il livello di base dell'organizzazione. Quasi tutti i modi in cui un sistema operativo interagisce con gli utenti, le applicazioni e i modelli di sicurezza dipendono dal modo in cui il sistema memorizza i suoi file su un dispositivo di memorizzazione. Per svariate ragioni è importante che gli utenti, così come i programmi, possano fare riferimento a una linea guida comune per sapere dove leggere e scrivere i file.

TaggyFS è un File System a Tag. Esso si prefigge l'obiettivo di superare quelli che sono i limiti riscontrati nel tradizionale file system di tipo gerarchico, utilizzato dai più moderni sistemi operativi. Per la sua realizzazione si è preso spunto dall'enorme successo nel web di applicazioni che basano la classificazione dei propri contenuti sui tag. Esempi lampanti sono oggi i notissimi youTube e Flickr.

Obiettivo primario è stato quello di creare un File System che non disorienti l'utente al primo approccio con esso. Per questo motivo TaggyFS si presenta del tutto simile a un file system gerarchico, disponendo degli stessi comandi e fornendo una user experience pratica e innovativa.

Il progetto è partito con uno studio concettuale sui file system di tipo gerarchico e sui loro limiti. Quanto raccolto verrà illustrato nel Capitolo 1 di questo documento. Si è passati poi allo studio dei sistemi a tag, partendo da un quadro generale e dopo aver effettuato un'analisi dello stato dell'arte e di come si sia giunti allo stato attuale delle cose, studiando alcune applicazioni che si basano sul paradigma del tagging. Questa fase e i suoi risultati verranno descritti accuratamente nel Capitolo 2. È succeduta quindi una fase di modellizzazione, avente lo scopo di creare un primo esemplare descrittivo-comportamentale di un possibile file system a tag. Essa è descritta nel Capitolo 3. La fase successiva ha visto invece una progettazione più specifica che ha comportato la scelta degli strumenti necessari per la realizzazione dell'applicazione. Il Capitolo 4 descrive nel dettaglio questa fase. A seguito dell'implementazione di quanto progettato sono stati effettuati alcuni test per controllare che l'applicazione si comportasse effettivamente come prefissato. I log e gli screenshot relativi a questa fase verranno presentati nel Capitolo 5. Il documento termina con le nostre conclusioni e una descrizione dei possibili sviluppi futuri affrontati nel Capitolo 6

1 Perché un file system virtuale?

1.1 Sempre più informazioni

Lo sviluppo tecnologico ha negli ultimi anni aumentato esponenzialmente la quantità (e la qualità) delle informazioni che è possibile digitalizzare. Una interessante ricerca della School of Information Management and Systems alla University of California di Berkeley [1] ha mostrato che “la produzione mondiale annuale di immagini, film e altri contenuti ottici e magnetici richiederebbe circa 1,5 miliardi di gigabyte, ovvero l'equivalente di circa 250 MB per ciascun uomo, donna o bambino sulla Terra”. Questa stima è destinata a crescere.

Tutto ciò si riflette anche nella vita quotidiana dell'utente medio, che si trova così a dover organizzare all'interno del proprio sistema digitale sempre nuove informazioni. Oggi egli ha una naturale familiarità con il file system del proprio sistema operativo che solitamente è di tipo gerarchico e cerca di sfruttarne al meglio le caratteristiche, creando una complessa struttura di cartelle annidiate per ottimizzare la ricerca successiva dei file.

Alcune ricerche, tra cui la più notevole di D. Barreau e B. A. Nardi [2], hanno mostrato però come questo tentativo porti a risultati insoddisfacenti: i limiti di un file system di tipo gerarchico sono molti e i ricercatori hanno evidenziato come “ogni tentativo di elaborare schemi complessi per archiviare le informazioni sia destinato a fallire”.

1.2 File e File System

Nei sistemi moderni, il “file” è l'oggetto più utilizzato: un pathname unico identifica ogni file all'interno di un sistema. Ogni file si comporta come ogni altro file nel modo in cui viene utilizzato e modificato: le stesse chiamate di sistema e gli stessi comandi funzionano con qualsiasi file. Questo succede indipendentemente dal supporto fisico che contiene l'informazione e dal modo in cui l'informazione è organizzata sullo stesso.

La maggior parte dei sistemi operativi, attualmente disponibili sul mercato, utilizza file system di tipo gerarchico. Ciò significa che l'informazione viene organizzata secondo precisi schemi creati dall'utente. La struttura che meglio si adatta a rappresentare un file system di questo tipo è un albero: ogni directory può esser rappresentata da un nodo, mentre i file corrispondono alle foglie.

In un file system di tipo gerarchico si possono quindi definire dei livelli, così come accade per gli alberi.

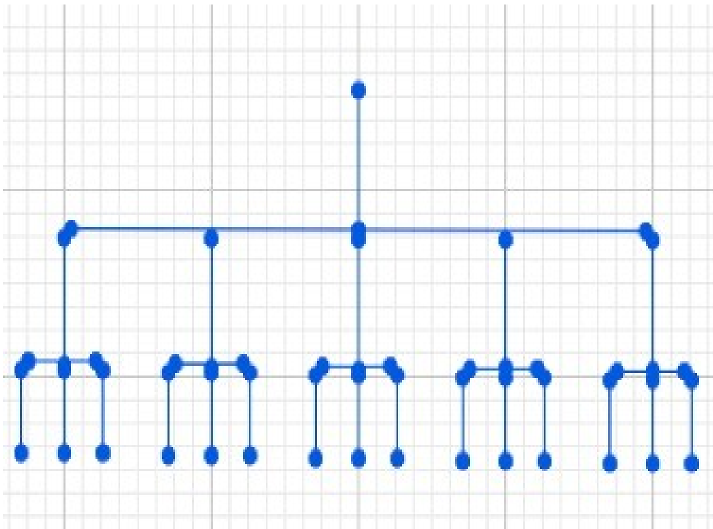

Figura 1.1: Un esempio di struttura ad albero

Come caso di studio si è scelto di utilizzare il modello Unix, adottato da Linux e dai più moderni sistemi operativi. Unix adotta il concetto di super-blocco, inode, directory. L'albero dei file che viene visto in un determinato momento dipende da come le differenti parti vengono assemblate; ogni "parte" in questo caso è rappresentata da una partizione di disco rigido o da un altro dispositivo che viene "montato" nel sistema.

- Il "super-blocco" deve il suo nome alle sue origini storiche, quando la meta-informazione riguardo al disco (o alla partizione) era immagazzinata nel primo settore del disco stesso. Al giorno d'oggi il super-blocco non corrisponde più ad un blocco di dati del disco, ma è ancora la struttura dati che contiene le informazioni riguardo al file system di cui fa parte.

- Un “inode” (index-node) è associato ad ogni file. L'inode contiene tutte le informazioni riguardanti un file tranne che il suo nome, ovvero il proprietario, il gruppo, i permessi e la dimensione dei dati contenuti nel file, come pure il numero di “link” relativi a tale file e altre informazioni. L'idea di separare le informazioni dal nome del file e dai suoi dati è quella che permette l'implementazione degli “hard link”, come pure permette di usare le notazioni “punto” e “punto-punto” per le directory, senza il bisogno di trattare tali nomi come speciali.
- La “directory” è un file che associa i nomi dei file agli inode. Il kernel non usa nessuna struttura dati particolare per rappresentare una directory, che viene trattata come un file normale nella maggior parte delle situazioni. Una directory viene letta e modificata tramite funzioni specifiche a ciascun file system, indipendentemente da come l'informazione è immagazzinata sul disco.
- Il “file” è un oggetto associato ad un inode. Di solito i file sono aree dati, ma possono anche essere directory, dispositivi, strutture quali pile, code, oppure socket. Un “file aperto” è rappresentato nel kernel da una struttura struct file: tale struttura contiene un puntatore all'inode che identifica il file. Le strutture file vengono create dalle chiamate di sistema come open(), pipe() e socket(), esse vengono condivise tra processo padre e figlio attraverso la chiamata fork().

1.3 I limiti dei file system di tipo gerarchico

Come già è stato accennato, l'albero è il modello di riferimento per un file sytem di tipo gerarchico. In un file system di questo tipo, si possono quindi definire dei livelli, così come accade per gli alberi. La rigidità di codesta struttura, mostra facilmente i limiti di questo tipo di organizzazione:

- Per ricercare un file, ad esempio, l'utente è obbligato a scorrere tutto l'albero, dalla radice alle foglie. Il file dunque, pur potendo avere dei collegamenti semantici con i livelli superiori, non è visibile da essi. Ad esempio il file x, raggiungibile al path /a/b/c/x non è visibile né in /a né in /a/b.
- Inoltre una struttura ad albero costringe l'utente a scegliere una modalità di classificazione statica: dato il file precedente /a/b/c/x, non ha alcun senso ricercarlo come /c/b/a/x o qualsiasi altra permutazione dell'ordine delle directory. Infatti non esiste proprietà commutativa tra le cartelle.
- Non esiste la possibilità di un raffinamento della ricerca: da una cartella è possibile vedere solo le sottocartelle esplicitamente posizionate dall'utente all'interno della stessa.
- Infine per ciascun file è possibile una e una sola allocazione, a meno che esso non venga duplicato.

1.4 Un File System Virtuale

TaggyFS è in realtà un file system virtuale. Un file system virtuale (VFS) è un layer astratto che lavora al di sopra di un file system reale. Lo scopo di un VFS è permettere alle applicazioni client di avere accesso al file system reale secondo le modalità definite da esso in modo uniforme. Un VFS può, ad esempio, essere usato per accedere a periferiche locali o di rete in maniera trasparente senza che l'applicazione client se ne renda conto. Può essere usato per superare le differenze tra i sistemi operativi sottostanti, in modo tale che l'applicazione non si renda conto del tipo di sistema a cui sta accedendo. O può, semplicemente, fornire una indicizzazione, classificazione, modalità di accesso differente rispetto a quella che effettivamente l'utente avrebbe, accedendo al file system reale. Un VFS specifica un'interfaccia tra il kernel e il file system reale. È dunque semplice creare nuovi file system virtuali: basta compilare questa interfaccia, aggiungendovi nuove regole.

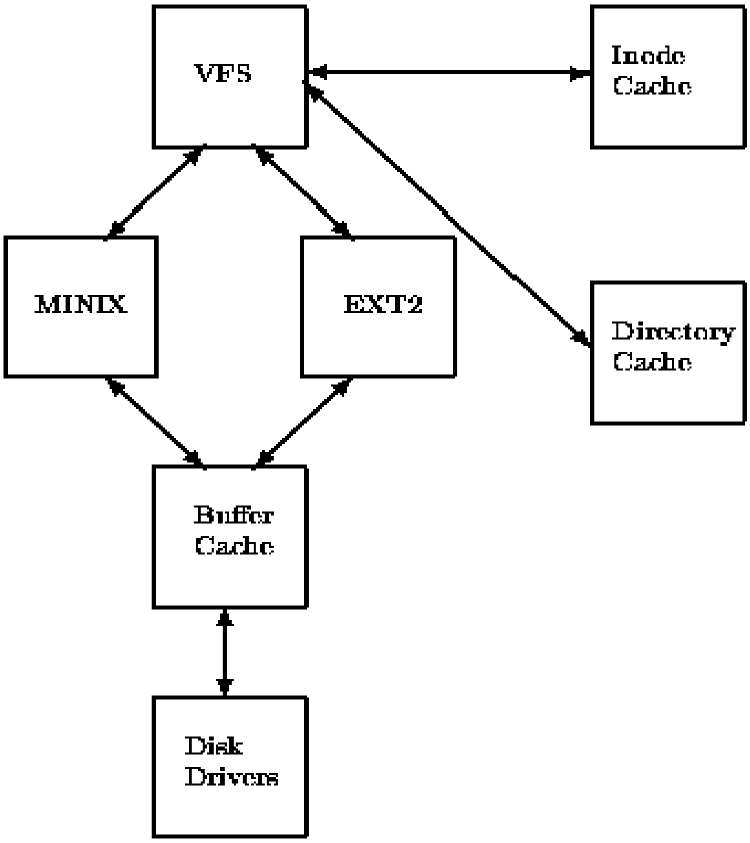

Figura 1.2: Schema a blocchi di un sistema operativo contenente un VFS

La figura di cui sopra, mostra la relazione tra un Virtual File System e il file system reale in un sistema Linux. Il Virtual File System deve organizzare gli eventuali differenti file system che sono montati sotto di esso. Per permettere ciò, vengono mantenute strutture dati le quali descrivono l'intero (virtual) file system e il file system reale.

Un VFS descrive i file del sistema nei termini di super-blocco e inode allo stesso modo del file system reale sottostante, mentre gli inode di un VFS descrivono i file e le directory all'interno del sistema, il contenuto e la topologia del Virtual File System.

Quando un file system è montato, il VFS deve leggerne il super-blocco corrispondente. Ogni routine di lettura del super-blocco di ciascun file system deve elaborare la topologia del file system e mappare quell'informazione sulla struttura dati del super-blocco del VFS. Ognuno di questi super-blocchi contiene informazioni e puntatori alle routine che eseguono particolari funzioni. Così, ad esempio, il super-blocco rappresentante un file system EXT2 contiene un puntatore alla routine di lettura degli inode EXT2. Questa routine di lettura degli inode EXT2, come tutte le specifiche routine di lettura degli inode di un qualsiasi file system, riempie i campi dell'inode del VFS. Inoltre ciascun VFS super-blocco contiene un puntatore al primo VFS inode del file system.

Ad esempio, per richiedere una *ls* su una directory o una *cat* per un file, occorre che il Virtual File System cerchi al suo interno gli inode che rappresentano il file system reale. Poiché ciascun file e ciascuna directory nel sistema sono rappresentati da un inode del VFS, ne consegue che alcuni inode saranno ripetutamente acceduti. Questi inode verranno mantenuti nella inode cache,la quale rende l'accesso ad essi più veloce. Se un inode non è nell'inode cache, allora una specifica routine del file system deve essere chiamata per leggere l'inode appropriato. L'azione di lettura dell'inode fa sì che esso sia copiato nella inode cache e i successivi accessi all'inode lo mantengano nella cache. Naturalmente i VFS inode meno utilizzati verranno rimossi dalla cache.

Tutti i file system utilizzano una buffer cache comune per memorizzare i dati dai dispositivi fisici sottostanti, al fine di incrementare la velocità di accesso. La buffer cache è indipendente dai file system ed è integrata nei meccanismi che il sistema operativo utilizza per allocare, leggere e scrivere i data buffers. Esso ha il vantaggio di rendere i file system indipendenti dai dispositivi sottostanti. Nel momento in cui un file system reale legge i dati dal disco fisico sottostante, avvengono richieste ai driver dei dispositivi di lettura dei blocchi fisici dal dispositivo che questi controllano. Quando i blocchi vengono letti dai file system sono salvati nella buffer cache globale condivisa da tutti i file system. I buffer all'interno della buffer cache sono identificati dal proprio numero di blocco e da un id univoco per il dispositivo che lo ha letto. In tal modo se gli stessi dati sono richiesti più volte, verranno rintracciati nella buffer cache piuttosto che letti dal disco, operazione che richiederebbe più tempo.

Il VFS tiene anche una cache delle directory più visitate per garantire un eventuale accesso futuro più veloce.

2 Web e Folksonomie

2.1 Tassonomia vs Folksonomia

Il modello teorico a cui risponde un file system gerarchico è quello di tassonomia, ossia una classificazione gerarchica e rigida di concetti. Volendone dare una descrizione più formale, una tassonomia è vista come struttura ad albero di istanze (o categorie) appartenenti ad un dato gruppo di concetti. A capo della struttura c'è un'istanza singola, il nodo radice, le cui proprietà si applicano a tutte le altre istanze della gerarchia (sotto-categorie). I nodi sottostanti a questa radice costituiscono categorie più specifiche le cui proprietà caratterizzano il sotto-gruppo del totale degli oggetti classificati nell'intera tassonomia.

Nel capitolo precedente sono già stati evidenziati i limiti di questo approccio, che viene ora messo a confronto con una nuova idea, la quale cerca di superarne in parte i difetti.

La **folksonomia** viene definita per la prima volta dall'unione delle parole *folk (*o folks) e *sonomy* (contrazione di tassonomia) che è stata attribuita a Thomas Vander Wal, come metodo atto alla categorizzazione collaborativa di informazioni mediante l'utilizzo di parole chiave (o tag) scelte liberamente. Sostanzialmente quest'idea copre diversi rami di conoscenza ed esperienza, che tra loro hanno significative differenze dividendosi dunque in:

- narrow folksonomy
- broad folksonomy

Nel primo caso le risorse possono essere etichettate da un solo utente a favore di tutti gli altri; nel secondo caso, invece, più utenti possono assegnare il medesimo marcatore allo stesso oggetto, generando quindi della meta informazione aggiuntiva. Per maggior chiarezza, si può asserire che la nuova informazione, consiste nella possibilità di conoscere il numero di etichette uguali associate ad ogni risorsa dai diversi utenti, permettendo di effettuare un ordinamento in base a tale proprietà. In questo modo ogni visitatore verrà a conoscenza dei vari concetti, con relativa probabilità, che quel concetto sia preponderante rispetto a tutti gli altri associabili attribuibili all'oggetto stesso, secondo i criteri cognitivi della comunità.

Diversamente dalle tassonomie, per le quali risulta necessario un lavoro di progettazione da parte di esperti, la folksonomia si espande più in fretta a causa principalmente della sua semplicità d'approccio, tramite la quale ogni utente, anche non in possesso di una specifica formazione sulla risorsa che manipola, può ugualmente cercare di categorizzarla attraverso categorie a lui familiari. Altra motivazione della veloce diffusione di questa idea riguarda la similarità con cui si avvicina alla mente umana rendendo possibile visualizzare ogni risorsa attraverso diverse angolazioni.

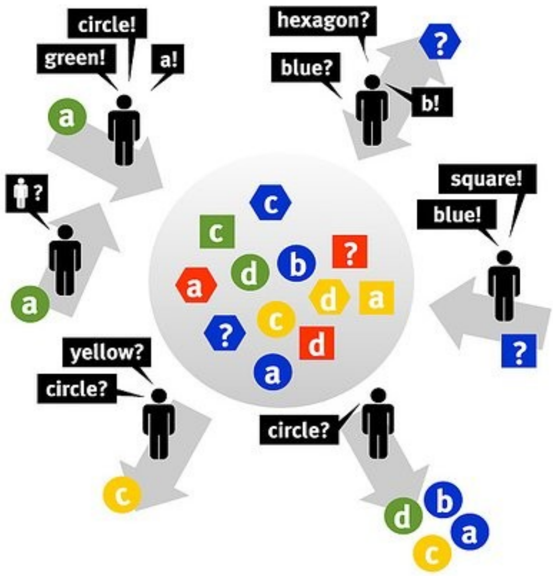

Figura 2.1: Esempio di folksonomia

Nella pratica attuale, come si avrà modo di capire nel corso della lettura, la folksonomia viene spesso associata all'idea del tagging**.** Questa inesattezza deriva dall'utilizzo che oggi viene fatto dei tag, quale principale strumento atto ad implementare l'idea stessa di folksonomia, soprattutto nella rete.

In questo elaborato, viene analizzato dunque il concetto di tagging come ramo particolare, ma largamente utilizzato della folksonomia, approdando allo studio di un file system a tag.

Va comunque ricordato che la folksonomia risulta molto più ampia del semplice tagging a scopo personale, in quanto il suo potere espressivo è dato in larga misura dalle meta informazioni generate conseguenti dell'aggregazione dell'informazione. Senza un contesto distribuito sottostante i tag diverrebbero semplici parole piatte, utili solamente all'utente che le ha associate all'oggetto.

I vantaggi di un simile approccio all'organizzazione delle informazioni spaziano dai bassi costi di manutenzione e gestione all'aggiornamento, al raffinamento delle associazioni effettuato automaticamente secondo i criteri utilizzati dalla comunità, oppure dalla facilità di scaling conseguente all'operazione di aggiunta, modifica o rimozione di un tag (operazione sempre possibile). Verranno trattati in seguito nello specifico riferendosi al file system a tag in seguito.

2.2 Il WWW e i tag

Il web è un universo di informazioni in continua e vertiginosa espansione. A differenza dei media tradizionali, il web consente a chiunque (o quasi) di produrre e pubblicare contenuti virtualmente senza limiti. Come catalogare questi dati per renderli facilmente reperibili agli utenti?

Intorno alla metà degli anni 90 ha luogo con Yahoo! [3] il primo tentativo di mettere ordine nel web. Degli esperti di catalogazione organizzano i contenuti in categorie raggruppate gerarchicamente. Il modello concettuale applicato ricorda lo schema del catalogo di una biblioteca, dove i libri sono suddivisi per argomento e ordinati in scaffali. La gerarchia delle categorie, invece, ricorda il modello del file system di un computer, dove i file sono organizzati in cartelle e sotto-cartelle, e in una cartella è possibile creare dei collegamenti a risorse presenti in altre cartelle. Sembra naturale poter applicare al web un modello concettuale che funziona bene per le biblioteche e per i personal computer, ma, come aveva mostrato Tim Berners Lee (creatore del World Wide Web) nei primi anni novanta, il web è costituito da una rete di collegamenti. Non è un gigantesco file system gerarchico, né presenta i vincoli fisici degli scaffali di una biblioteca.

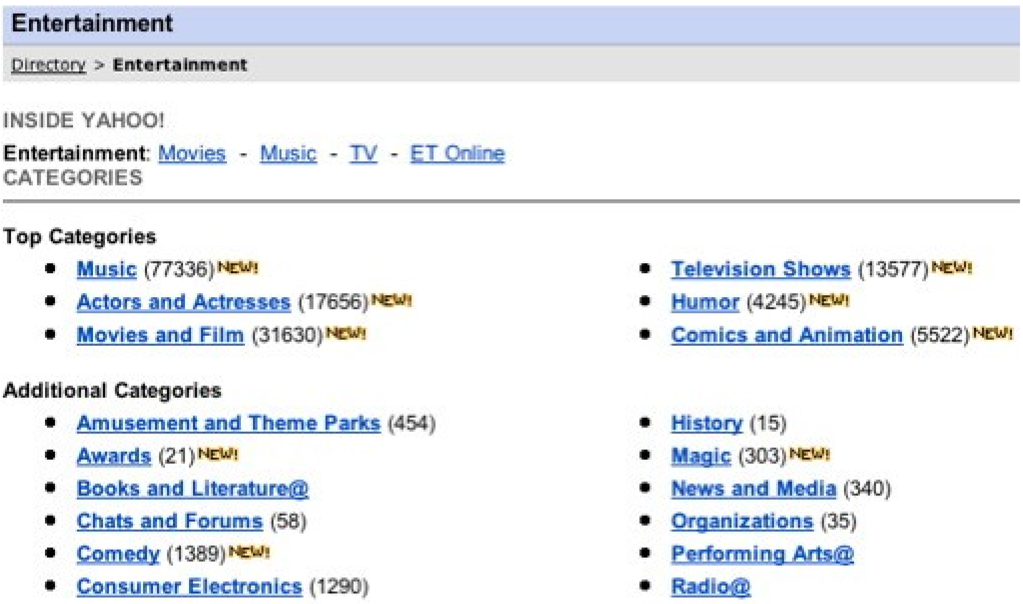

Figura 2.2: Screenshot delle web-directory di Yahoo!

Con l'aumentare delle informazioni, la catalogazione si fa sempre più difficile così come la navigazione degli utenti nei vari livelli di categorie. Nel frattempo si stanno diffondendo i motori di ricerca. Ci si rende conto che nel web non ci sono scaffali, ma soprattutto si capisce che la catalogazione fatta da un ristretto pool di esperti non funziona bene per il web: in esso, infatti, vengono continuamente prodotte nuove informazioni che non appartengono a categorie formali, né esistono enti o autorità in grado di controllare e coordinare i flussi di informazione e gli utenti che li producono. Non è possibile prevedere in anticipo gli schemi cognitivi che le persone utilizzeranno per categorizzare le nuove informazioni prodotte, né è possibile prevedere la stabilità delle categorie nel tempo. Il “paradigma della ricerca” presto soppianta il "paradigma della navigazione" all'interno delle directory. Il nuovo paradigma stabilisce che nessuno può anticipare cosa l'utente cercherà e come effettuerà la sua ricerca. Solo quando l'utente pone la domanda (con parole chiave liberamente scelte), il sistema fa del suo meglio per reperire le informazioni attinenti, scandagliando la struttura non gerarchica della rete.

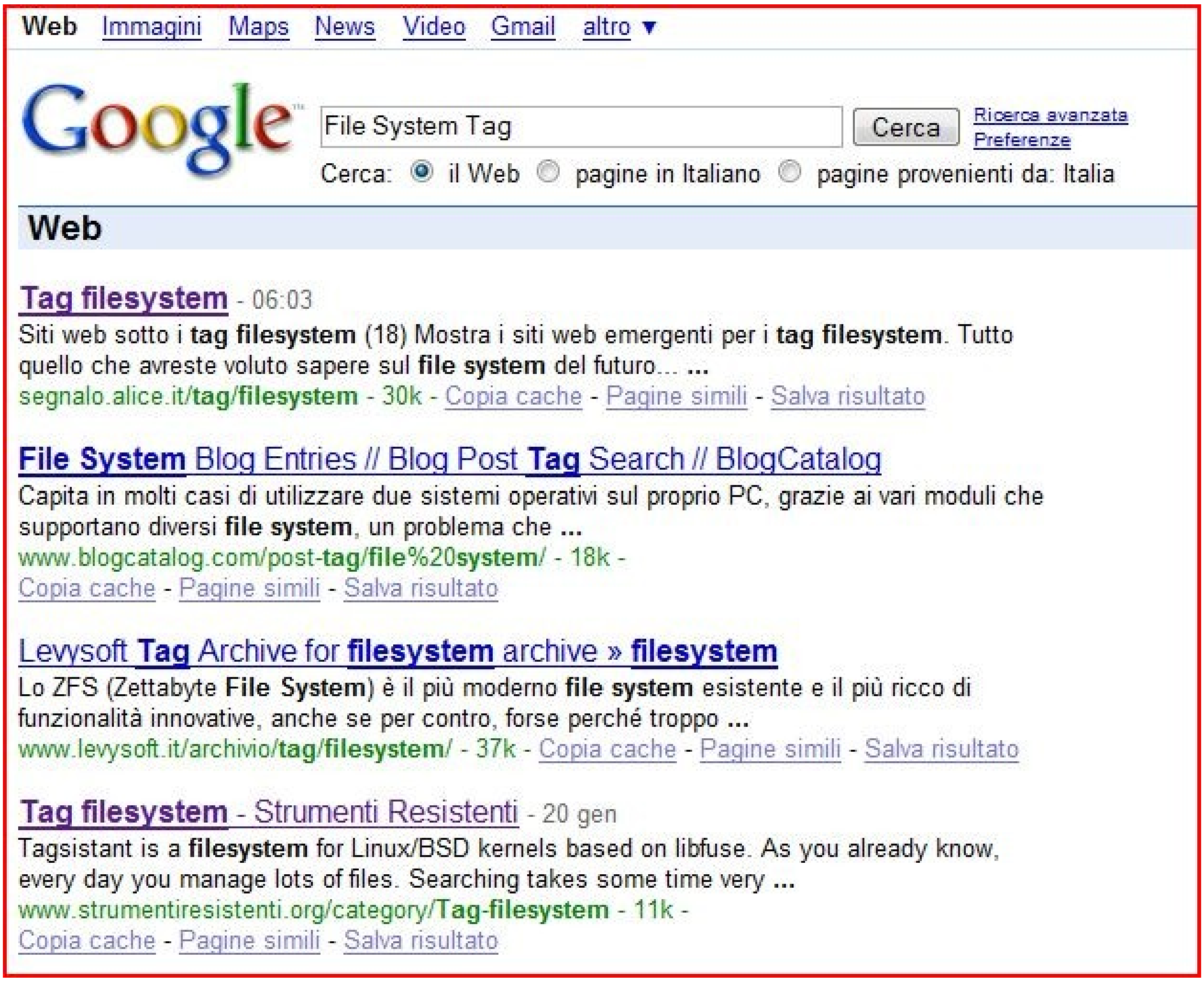

Figura 2.3: Uno screenshot del risultato di una ricerca su Google.

Nel 1998 nasce Google[4] e nel giro di pochi anni diventa lo strumento più utilizzato per cercare informazioni sul web. La novità di Google rispetto ai precedenti motori di ricerca sta nell'ipotesi che un sistema in grado di analizzare le relazioni fra le pagine web darà risultati migliori rispetto alla semplice analisi della frequenza di un parola nelle singole pagine. Benché la ricerca per parole chiave sia un notevole progresso rispetto alla navigazione nelle directory, occorre una certa esperienza e molta pazienza per filtrare i risultati qualitativamente validi dalla massa di informazioni inutili che ci vengono restituite. Gli algoritmi di ricerca sono sofisticati, ma sono ben lontani dall'emulare la raffinatezza dei processi cognitivi dell'uomo nel valutare la qualità e la pertinenza dei contenuti.

Un passo successivo in questo senso è però avvenuto con la nascita di Del.icio.us [5], il servizio di social bookmarking ideato da Joshua Shachter (e ora parte di Yahoo!).

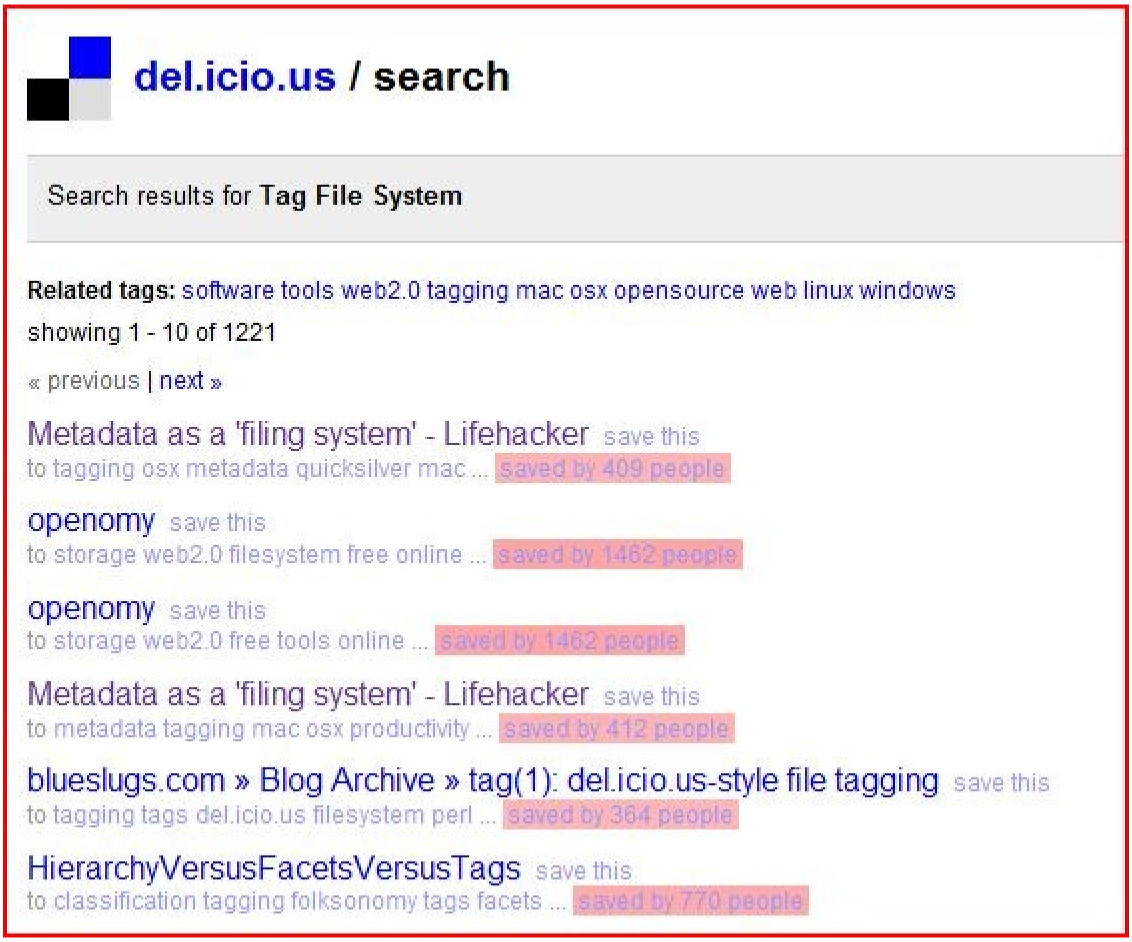

Figura 2.4: Uno screenshot di una didascalia su Del.icio.us

L'idea di base è molto semplice: l'utente trova un articolo ben fatto su un argomento che gli interessa, lo salva nei preferiti su Del.icio.us, aggiunge ad esso una breve descrizione, e assegna delle etichette (che sceglie liberamente) per ricordare l'argomento o gli argomenti a cui si riferisce, condividendo questo bookmark con gli altri utenti del sito.

La catalogazione fatta da ciascun utente è un lavoro imperfetto e di qualità mediocre se paragonato a ciò che potrebbe fare un esperto. Ciascuno cataloga le informazioni a modo suo, ma esistono schemi di categorizzazione condivisi da gruppi di utenti che hanno in comune un campo di interesse. La somma del lavoro di catalogazione svolto da un gran numero di utenti dà un risultato decisamente superiore a quello di un gruppo di esperti di settore, e ad un costo nettamente più basso. Da quanto sopra enunciato, viene meno il problema di prevedere gli schemi cognitivi degli utenti, essendo loro gli autori della catalogazione ed essendo essi stessi gli artefici della scelta dei tag. Non occorre nemmeno preoccuparsi della stabilità nel tempo delle categorie: il tagging (processo di libera assegnazione delle etichette ai bookmark) è un processo continuo e i tag evolvono con il contributo degli utenti.

Non esistono tag sbagliati. Possiamo usare qualsiasi etichetta riteniamo sia utile a descrivere il contenuto di una pagina web. Il sistema, però, può suggerirci quali tag sono stati scelti dalla maggior parte delle persone che prima di noi hanno aggiunto quella stessa pagina ai loro preferiti. Un tag non è altro che una parola-chiave (descrittore) atta a specificare un attributo, una caratteristica dell'oggetto a cui esso è applicato. È in questo modo possibile ricercare tutti gli oggetti aventi una determinata caratteristica o un particolare set di questa.

2.3 Un File System a Tag

I limiti precedentemente riscontrati in un file system di tipo gerarchico vengono attenuati, se non in alcuni casi addirittura eliminati, in un file system basato sui tag. Il concetto base è semplice: sfruttare gli enormi vantaggi riscontrati nell'utilizzo di tag nel web in locale, utilizzando come oggetti da etichettare non più le pagine web (o i video o le immagini), ma gli stessi file.

Come mostrato dalla ricerca “A cognitive analysis of tagging” [6], alla base dei limiti di un file system gerarchico vi è il seguente assunto: un file presenta intrinsecamente una serie di concetti ad esso legato. L'inserimento di esso all'interno di una struttura ad albero obbliga l'utente a sceglierne uno tra i tanti, quello mediante il quale avviene la categorizzazione:

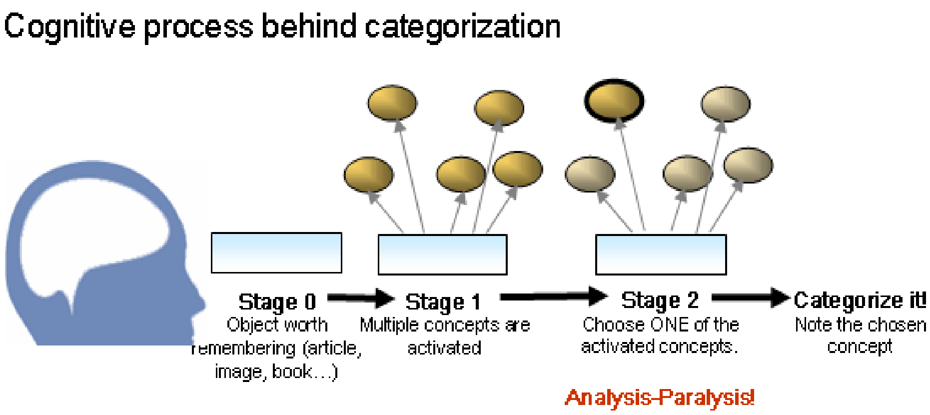

Figura 2.5: Processi cognitivi alla base della categorizzazione per un sistema gerarchico

Il processo di tagging invece non necessita di alcuna categorizzazione, poiché ogni concetto legato al file è direttamente associato ad esso tramite una etichetta:

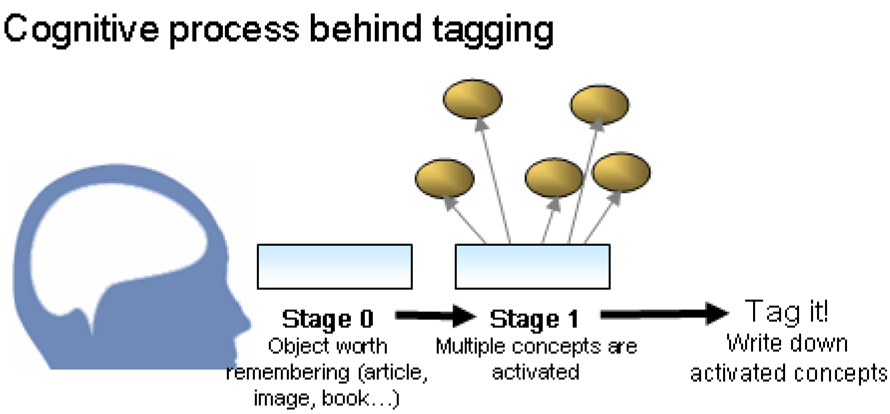

Figura 2.6: Processi cognitivi alla base del "tagging"

In questo modo:

- Un file è visibile accedendo a uno qualunque dei concetti ad esso legato.

Non è necessario ricordare tutti i tag ad esso associati. Un file x, a cui sono associati i tag a, b e c, sarà comunque visibile anche all'accesso nella sola a, b o c o a qualunque altra combinazione di essi.

- Come già accennato, non esiste una classificazione predefinita. Non esiste una gerarchia tra tag e cercare i file taggati con “a e b” equivale a cercare i file taggati con “b e a”.
- Esiste la possibilità di affinare la ricerca utilizzando ulteriori condizioni al fine di formulare una query che si avvicini il più possibile alla richiesta che l'utente intende fare al sistema.
- Ad ogni file possono essere associate molte etichette senza che sia necessaria alcuna duplicazione di esso con conseguente economia dal punto di vista dello spazio utilizzato sul supporto fisico.

L'utilizzo di tag inoltre porta numerose innovazioni. Tra esse spicca la possibilità di eseguire query avanzate: è possibile creare vere e proprie proposizioni logiche, con gli operatori AND, OR e NOT. Possiamo così generare un comando che restituisca “tutti i file aventi certi tag oppure certi altri e non altri ancora”.

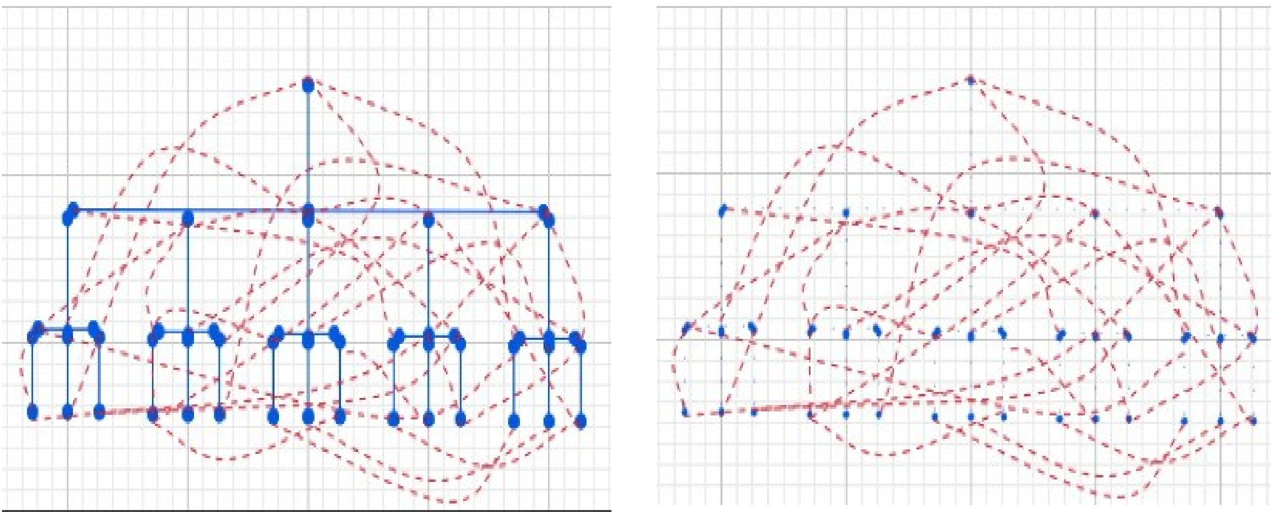

Figura 2.7: La creazione dei vincoli semantici e l'eliminazione dei vincoli strutturali

Tra i file verranno dunque a crearsi dei collegamenti impliciti, dovuti alle relazioni tra i vari tag, mentre la struttura a albero cesserà di esistere.

La struttura che meglio rappresenta la nuova situazione diventa l'insieme matematico. I singoli elementi saranno i file, contenuti negli insiemi rappresentanti i tag, all'interno di un sistema universo costituito dal file system. Avranno dunque senso le operazioni di intersezione, unione e sottrazione tra tag che potranno eventualmente dare risultato nullo. Saranno inoltre evidenti le relazioni tra i file: ciascun file x è in relazione con tutti quelli contenuti in almeno uno degli insiemi di cui x fa parte.

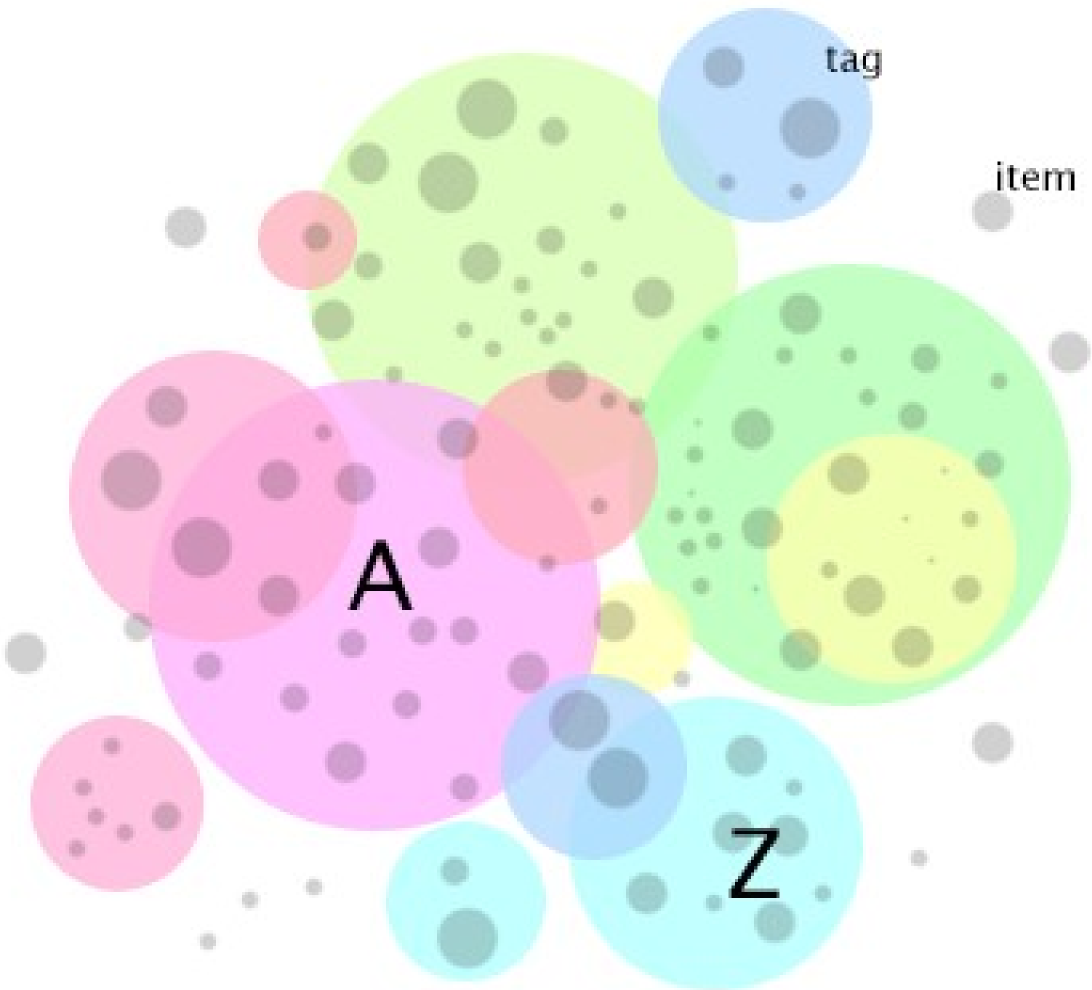

Figura 2.8: Visione insiemistica dei file in un file system a tag

2.4 I file system a tag: un primo approccio

Attribuire etichette ai file diventa il modo più comodo per creare relazioni tra gli stessi. I primi passi in questo campo seguono le orme dei siti web che per primi hanno utilizzato i tag. Vengono create applicazioni stand-alone che permettono una catalogazione dei file del proprio hard disk per una più rapida ricerca. Esempio in questo campo sono GLS[3] [7] e TAGS[8].

Il primo si presenta come un pratico file manager in grado di associare etichette ai file, estrarne i metadati associati e permettere ricerche all'interno degli stessi.

Il secondo è invece una applicazione che integra lo strumento di ricerca del sistema OS X con delle comode etichette da applicare ai vari file.

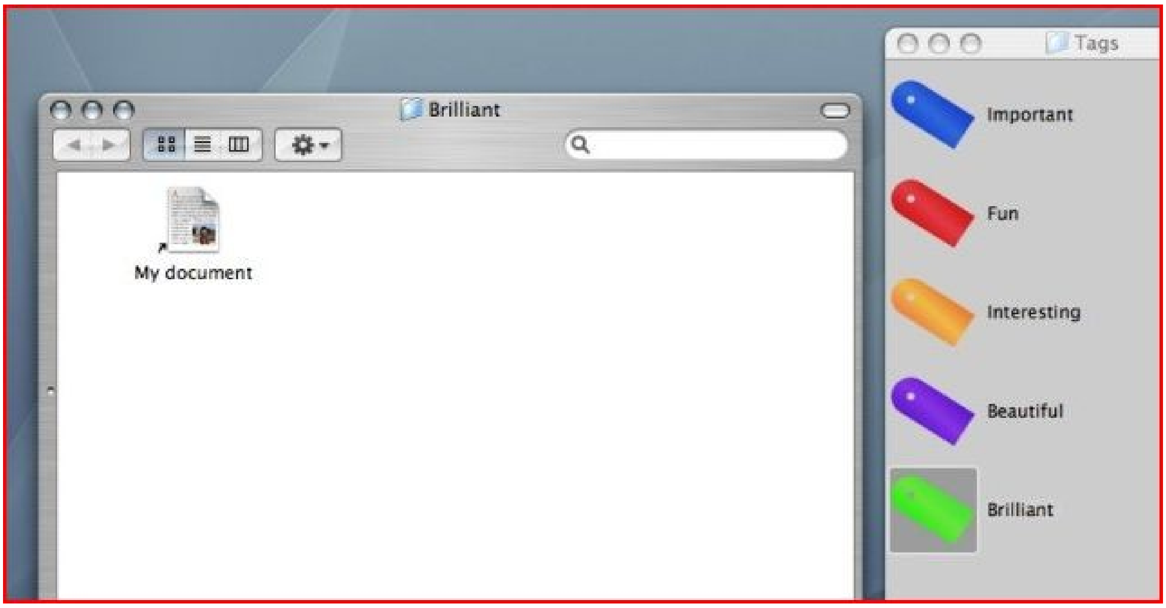

Figura 2.9: Uno screenshot di Tags

Entrambe le applicazioni hanno pregi e difetti, dovuti essenzialmente alla loro natura di applicazioni ad alto livello: una semplicità di installazione, un utilizzo intuitivo e pratico sono affiancati a un'interfaccia colorata e ricca di opzioni. Tra i difetti c'è però il fatto che questo tipo di applicazioni sono in realtà dei semplici “tool” di organizzazione e ricerca, che non si integrano però con gli altri programmi presenti nel calcolatore. È perciò possibile effettuare ricerche a parte e poi utilizzare i risultati di queste all'interno delle varie applicazioni.

Contemporaneamente alla creazione delle prime applicazioni autonome nasce l'idea di creare qualcosa che collabori direttamente col sistema operativo. L'idea di un layer virtuale che faccia da middleware tra l'utente e il file system gerarchico esistente.

È così che nascono numerosi progetti in questa direzione: RelFS[9], SemFS[10], e Tagsistant[11] sono gli esempi più riusciti in questa direzione; navigando nella rete Internet è possibile scovare molti progetti cominciati (e alcuni mai finiti) al riguardo. I tre progetti citati, seppur nati in contesti differenti, mostrano una linea comune: sono dei file system virtuali che conservano in un DB o in una RDF repository i metadati (tag) associati dall'utente ai file. Le normali chiamate di sistema vengono mappate e ridefinite: aggiungere una cartella con *mkdir* diventa nel nuovo sistema un'operazione di: “crea un nuovo tag”, con le relative necessarie modifiche al database o alla repository.

Inoltre gli ultimi due progetti hanno inserito un piccolo manager che permette di creare relazioni tra tag. È così possibile dire che il tag jazz è collegato al tag musica, che il tag metal è equivalente al tag heavy-metal, che il tag chiaro è opposto al tag scuro.

I progetti fin qui citati sono stati un ottimo spunto per la nascita di TaggyFS. Il loro utilizzo ha consentito di conoscere meglio questo nuovo modo di organizzare le informazioni evidenziandone i punti di forza ed i punti più deboli. Ciò che si è cercato di ottenere è un sistema innovativo, che non disorientasse l'utente al primo approccio, conservando intatti tutti i comandi a cui egli è abituato, cambiandone solamente la semantica così da adattarsi al nuovo sistema.

3 Implementazione astratta di un file system a tag

3.1 Le componenti del modello teorico

Come accennato nei capitoli precedenti, TaggyFS vuole essere un file system a tag. Il suo scopo è quindi quello di raccogliere i dati presenti nell'unità (solitamente un hard disk, ma non necessariamente) collegarli ai metadati presenti in un apposito spazio e offrire all'utente quelle operazioni che qualunque file system offre, come l'apertura, lo spostamento, la rinomina, la modifica, e così via.

A prescindere dall'implementazione pratica, esisteranno delle componenti imprescindibili, di cui il file system a tag dovrà servirsi per poter funzionare.

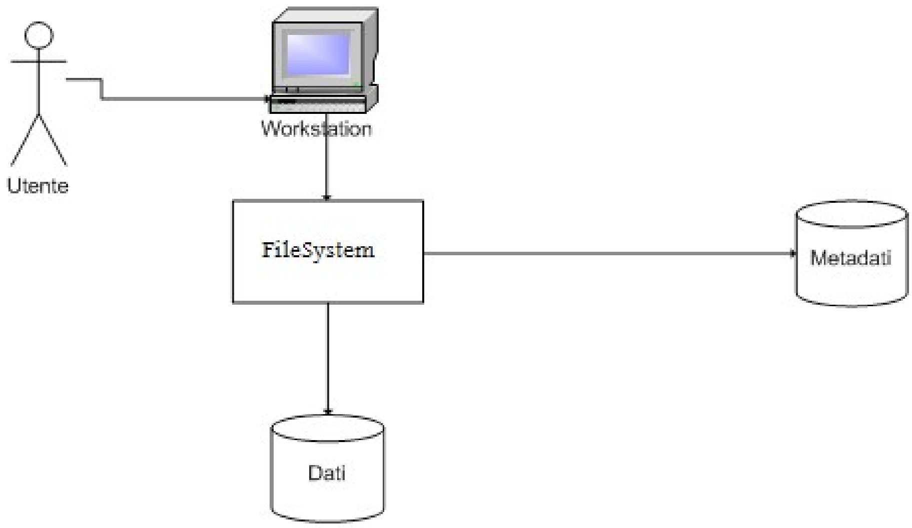

Figura 3.1: Modello a blocchi delle componenti principali in un file system a tag

- I **dati** rappresentano l'elemento di congiunzione tra i file system gerarchici e i file system a tag. Essi sono la fonte dell'informazione, la destinazione a cui l'utente vuole accedere. Dovrà perciò esistere una struttura fisica atta a contenerli (un hard disk, un cd rom, un floppy disk, un nastro magnetico o altro) e una apposita struttura dati che ne faciliti il ritrovamento.

- I **metadati** sono la novità inserita dai file system a tag. Essi rappresentano le informazioni aggiuntive riguardo al file, non reperibili da esso. In un file system a tag essi corrispondono ai tag stessi. Anche in questo caso deve esistere una struttura fisica di memorizzazione e una struttura dati atta alla ricerca.
- Il **file system** è il cuore del sistema operativo. Deve essere in grado di gestire i file, i metadati e le relative associazioni, eseguendo operazioni sia sui primi che sui secondi. Sarà dunque necessario che possegga metodi di basso livello per la loro gestione sulle strutture fisiche e che fornisca all'interfaccia metodi di più alto livello, comprensibili per l'utente finale.
- L'**interfaccia** è lo strumento di cui l'utente dispone per l'interazione con la macchina. Essa deve essere il più possibile semplice, intuitiva e adatta a ogni tipo di utenza. Deve offrire a chi la utilizza la possibilità di interagire con il file system, fornendogli la possibilità di accedere ai metodi esposti da esso in modo diretto o indiretto. Per uno stesso file system può esistere anche più di una interfaccia, ciascuna adatta a specifici utilizzi e/o utenti.

3.2 Le operazioni di un file system a tag

Le operazioni offerte da un tale file system, devono fornire all'utente la possibilità di interagire con file e tag. Sono state dunque identificate le operazioni base che saranno necessarie per il corretto funzionamento di TaggyFS (e di qualunque altro file system a tag) che ricalcano da vicino almeno quelle di un file system gerarchico:

- creare un tag
- eliminare un tag
- aggiungere un tag a un file

- eliminare un tag da un file
- rinominare un tag
- rinominare un file

3.3 Mappare le operazioni di un'interfaccia per file system gerarchici su un file system a tag

La prima problematica affrontata nel momento in cui si è deciso di costruire un file system a tag è stata: che significato dare a tutte quelle operazioni di cui conosciamo bene il significato nei file system tradizionali?

Di seguito verrà proposto un quadro generale dei comandi ad alto livello e il modo in cui si è deciso di ridefinirli nel nuovo file system. È fondamentale comprendere che queste associazioni sono state frutto di scelte: un file system che interpretasse in modo differente le varie chiamate di sistema sarebbe potuto essere ugualmente utilizzabile. Si è cercato di valutare di volta in volta le possibilità e si sono scelte quelle ritenute migliori, prendendo spesso spunto dall'esperienza maturata utilizzando applicazioni simili.

- **Nuova Cartella**: in un file system a tag, questi sono l'equivalente delle vecchie directory. Allo stesso modo, ogni tag ha associati uno o più file. In questo senso, creare una nuova cartella (vuota) equivale a creare un nuovo tag, pronto ad essere associato a uno o più file.
- **Rinomina Cartella**: la corrispondenza tra tag e directory, ha suggerito di convertire questo comando con il semplice "rinomina tag", rinominandolo per tutti quei file cui esso è associato.
- **Apri cartella**: l'apertura di una cartella nei sistemi tradizionali corrisponde alla visualizzazione di tutti i file contenuti in essa. È evidente quindi la grossa

differenza rispetto a un file system a tag. In TaggyFS si è scelto di mostrare all'utente tutti i file aventi come tag quello scelto. Se il path dalla radice alla cartella selezionata è composto da più di un tag (es. /a/b/c) il file system effettua di default l'AND logico tra i tag, mostrando i file aventi come tag tutti quelli presenti nel percorso (a AND b AND c).

- **Elimina Cartella**: l'eliminazione di una cartella corrisponde all'eliminazione di un tag. Esso viene eliminato da tutti i file. Se un file è taggato con esso, viene eliminato dal sistema operativo.
- **Copia File**: la copia di un file in un nuovo percorso corrisponde all'aggiunta dei tag facenti parte del nuovo percorso a quel file.
- **Sposta File**: lo spostamento di un file corrisponde all'aggiunta dei tag facenti parte del nuovo percorso e all'eliminazione dei tag facenti parte del vecchio.
- **Rinomina File**: l'operazione di rinomina di un file è inalterata rispetto a quella dei file system classici
- **Elimina File**: l'operazione di eliminazione di un file è inalterata rispetto a quella dei file system classici.

Operazione nuova è invece quella di eliminare un tag da un file per il quale non è stata trovata alcuna corrispondenza nei file system tradizionali.

4 Progetto e implementazione

4.1 La nascita di TaggyFS

Fino ad ora sono stati presi in considerazione, sebbene con un certo livello di approfondimento, solamente aspetti teorici, che hanno permesso, nella sezione precedente, di arrivare a formulare il modello teorico di un file system a tag.

In questa sezione sarà presentato invece il progetto pratico che effettivamente implementa le funzionalità teoriche descritte.

Ricordando che TaggyFS realizza un VFS ospitato su un architettura già esistente, la realizzazione ha portato a costruire qualcosa che funzionasse e si adattasse ai vincoli intrinseci del sistema sottostante, dovendo quindi effettuare scelte implementative riguardanti il sistema operativo da utilizzare, il metodo di realizzare un file system virtuale sul SO scelto ed infine l'archiviazione dei metadati e dei file fisici; la necessità quindi di trasformare i blocchi del progetto teorico in software in grado di offrire pari funzionalità.

4.2 Virtual file system, SO, linguaggio di programmazione: Fuse detta legge

Relativamente a questa problematica sono possibili due soluzioni:

1) sviluppare un'applicazione stand-alone in grado di funzionare come un visualizzatore di file secondo un paradigma tag-based;
2) arricchire il SO ospite, offrendo all'utente la possibilità di montare partizioni (sottoalberi) utilizzando il paradigma del file system a tag.

Chiaramente lo sviluppo di un prodotto integrato rispetto al sistema operativo, può sembrare a prima vista più complesso; si sta però ultimamente facendo strada un nuovo sistema di gestione dei file system per utenti che ha permesso di propendere per questa soluzione.

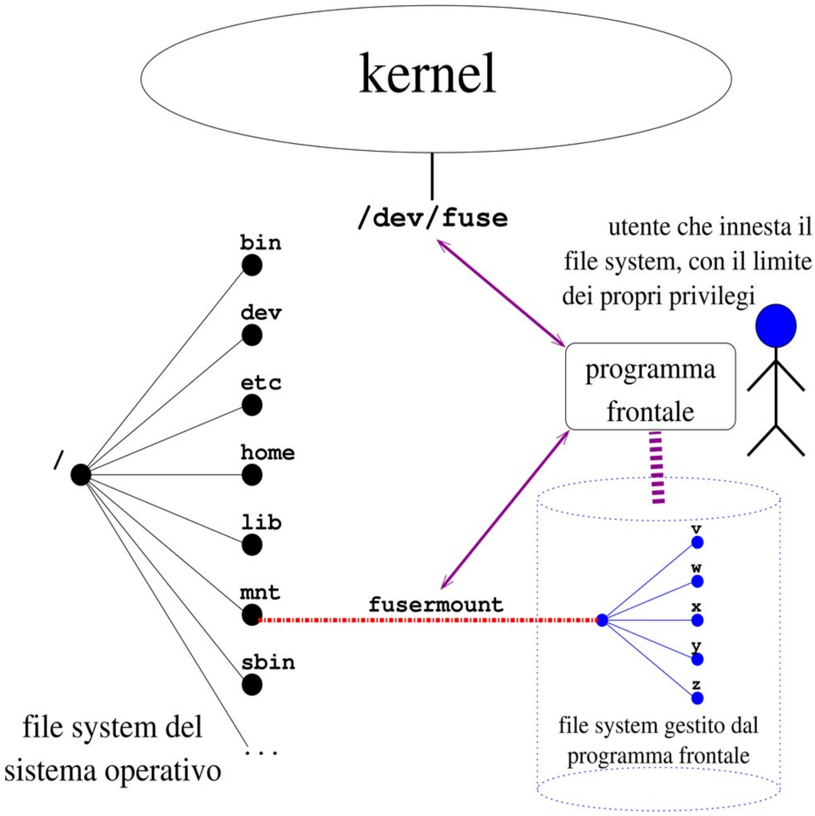

Figura 4.1: Modello di un sistema con il modulo Fuse attivato

Filesystem in USErspace (FUSE)[12] è un progetto Open Source, rilasciato sotto la licenza GPL, volto alla realizzazione di un modulo per il kernel Linux che permetta agli utenti non privilegiati di un sistema di creare un proprio file system senza la necessità di scrivere codice a livello kernel.

FUSE è particolarmente utile per scrivere file system virtuali. Risulta essere un sistema molto potente, in quanto virtualmente ogni risorsa disponibile ad essere implementata sfruttando FUSE può divenire un file system virtuale. Sono infatti stati realizzati, ad esempio, file system quali SSHFS, che permette di montare un sistema remoto in connessione SSH come un file system, o addirittura GmailFS, che utilizza lo spazio fornito da Google Inc. per le caselle e-mail del proprio sistema Gmail.com per ricavarne un file system virtuale.

Grazie a questo pacchetto si è quindi in grado di riscrivere le system call del sistema sottostante, e quindi le nuove regole semantiche che verranno utilizzate solamente nel sottoalbero montato con Fuse.

Questo fa comprendere da subito come la scelta di integrare, utilizzando Fuse, pur con tutte le difficoltà di implementazione, potrebbe portare a sviluppare un oggetto totalmente indipendente dai dispositivi fisici sottostanti, siano essi hard disk locali, network file system o altro.

Per quanto riguarda la scelta del sistema operativo adatto, la ricerca si è quindi ristretta ai soli che supportano il pacchetto Fuse. In questo modo la preferenza del sistema ospite è ricaduta sull'ambiente Linux in quanto è il sistema operativo su cui è stato possibile lavorare maggiormente durante il corso di studi.

Si è poi passati alla valutazione del linguaggio di binding per FUSE: in questo senso la scelta sarebbe potuta essere vastissima, essendo cospicuo l'insieme dei linguaggi di programmazione supportati (tra cui ricordiamo C, C#, C++, Pyton, Perl, Java e molti altri). La scelta del linguaggio è ricaduta su Java, poiché utilizza il paradigma ad oggetti, il quale tornerà molto utile durante lo sviluppo, poichè faciliterà l'interazione con un altro framework utilizzato illustrato in seguito.

4.2.1 Come opera Fuse

Dall'analisi di Fuse, emergono tre macro-blocchi su cui si basa la logica che lo rende in grado di fornire la potenza teorica di un virtual file system:

1. **Interfaccia per la ridefinizione delle system call.** L'interfaccia offerta, chiaramente in un diverso linguaggio di programmazione a seconda della versione scelta di FUSE, impone al programmatore di riscrivere il corpo di ogni system call del file system: è questo il momento in cui la nuova semantica

delle operazioni viene definita.

2. **Modulo kernel.** Esiste un modulo a livello kernel, il quale dialoga direttamente con il sistema operativo e intercetta da esso le chiamate relative alla porzione di file system montata sotto il controllo di Fuse. Queste richieste vengono trasferite al modulo user space ed è quindi questo il momento in cui la richiesta, passando di livello, può essere elaborata da un software a livello user.

3. **Modulo user space.** Il modulo user space conoscendo la richiesta, invoca lo stesso metodo che invocherebbe il kernel normalmente. In realtà, il metodo realmente chiamato è quello contenente il codice scritto dal programmatore. Tutto questo è reso possibile, poiché i nomi delle chiamate di sistema sono gli stessi che compaiono nel file d'interfaccia.

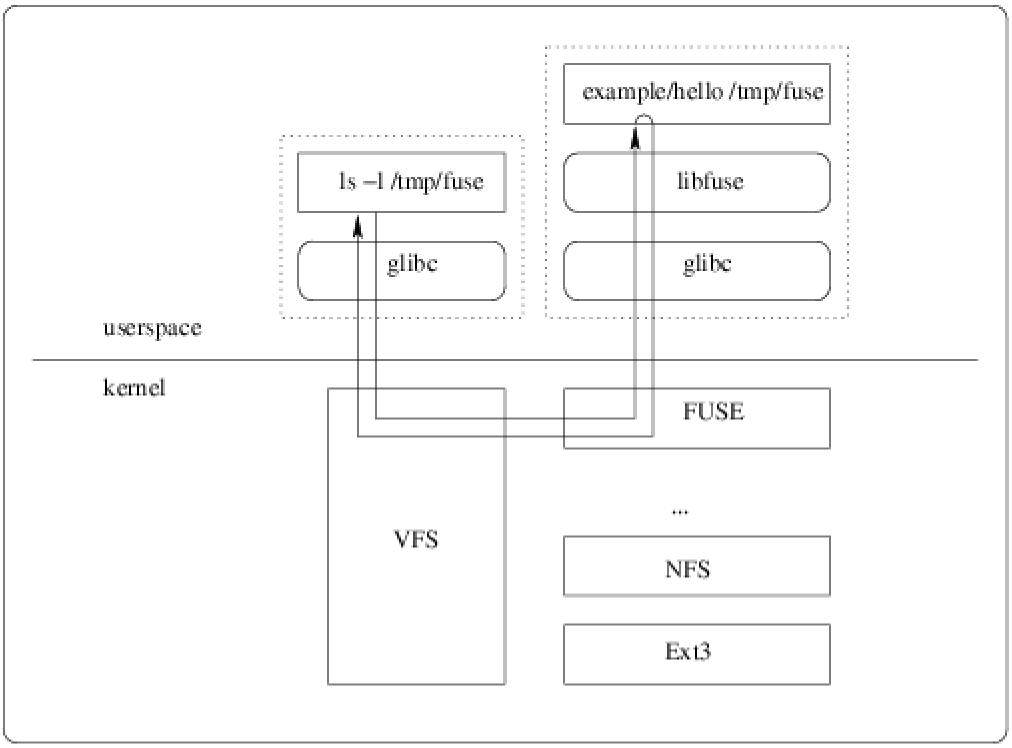

Figura 4.2: Schema di funzionamento di Fuse

I risultati prodotti a livello utente, vengono formattati opportunamente per essere interpretati dal file system sottostante reale, e quindi trasferiti al modulo kernel, che a sua volta li consegna al sistema operativo che sarà in grado di leggerli.

Un'ultima precisazione riguarda la formattazione dei dati che avviene in user space, poiché in kernel space non è possibile per l'applicazione compiere alcuna operazione. Fuse permette quindi di estrarre i comandi d'interesse per le directory montate. In tal modo può elaborarli a livello utente, riconsegnandoli in una forma interpretabile dal sistema operativo che a sua volta li restituirà all'utente destinatario preoccupandosi della visualizzazione.

4.3 Hibernate e il nuovo approccio per la memorizzazione dei metadati.

La memorizzazione dei metadati sarà affidata ad un DBMS, poiché il progetto nasceva con l'espresso intento di utilizzare una base di dati per tale compito.

Rispettando questo vincolo imposto nella consegna e scegliendo quindi MySql come DBMS non solo a causa della sua popolarità, ma anche poiché era sostanzialmente l'unico su cui abbiamo avuto possibilità di lavorare, si è deciso di arricchire il progetto introducendo uno strumento che implementa un'idea oggi molto di moda e sulla quale era già stato possibile lavorare in passato: Hibernate.

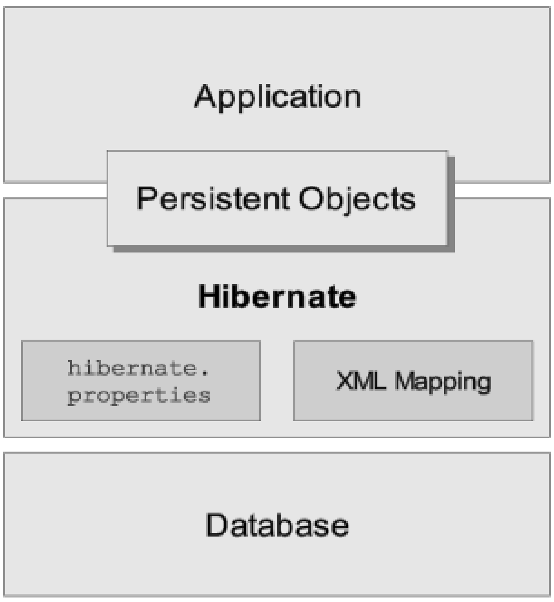

Figura 4.3: Hibernate come middleware tra l'applicazione e il database

Hibernate[13] è uno dei più noti ed utilizzati framework Java per l'Object Relational Mapping. È uno strumento che semplifica ed automatizza la gestione della corrispondenza tra il modello ad oggetti delle nostre applicazioni e quello relazionale dei database server ai quali queste ultime comunemente si interfacciano. Uno o più file di configurazione stabiliscono la corrispondenza tra gli oggetti della generica applicazione (e le relazioni che sussistono tra essi) e le tabelle del database cui il software è interessato ad accedere. Il motore di persistenza di Hibernate può occuparsi così di tutto il lavoro necessario per estrarre, inserire o modificare i dati che costituiscono lo stato degli oggetti del nostro dominio applicativo, generando autonomamente ed in tutta trasparenza i comandi e le interrogazioni SQL necessari.

Hibernate rende persistente praticamente ogni classe Java che lo richieda, senza eccessivi vincoli. Ogni POJO (*Plain Old Java Object*), disegnato in fase di modellazione del dominio applicativo, per poter essere gestito da Hibernate deve semplicemente aderire alle fondamentali regole dei più pratici *javabeans*: metodi pubblici *getXXX* e *setXXX* per le proprietà persistenti, esistenza di un costruttore nullo.

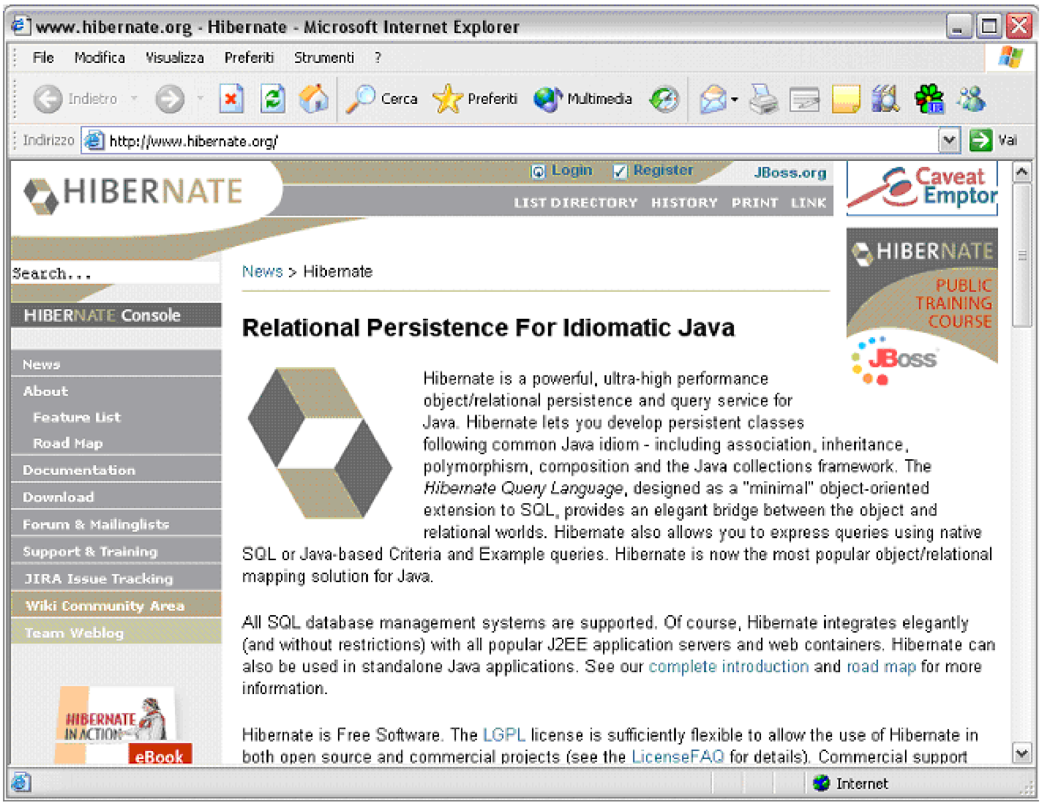

Figura 4.4: La hompage di Hibernate

Hibernate lavora quindi generando un livello di middleware che lascia al programmatore completa libertà di sviluppo, poiché non dovrà occuparsi della gestione del database o dello specifico dialetto utilizzato dal DBMS sottostante in quanto la comunicazione avviene con Hibernate e soprattutto tramite oggetti con cui il programmatore è abituato a trattare.

4.3.1 Come opera Hibernate

Risulta interessante comprendere in maniera più approfondita come questo framework di sviluppo riesca ad offrire tutte le funzionalità sopra descritte. Per raggiungere lo scopo è necessario capire quali siano e in che modo interagiscano tra essi i suoi componenti interni.

Fondamentalmente ogni applicazione Hibernate è costituita da un insieme di classi Java, tramite l'implementazione delle quali il tool è in grado di costruire una mappatura utile ad implementare il livello di astrazione richiesto. L'attenzione va quindi riposta nello sviluppo delle classi Java, le quali si dividono in tre gruppi:

- Java bean
- Java utils
- Java facade

Le **Java bean** sono classi Java che all'atto della compilazione verranno interpretate da Hibernate come tabelle del database, ossia ogni attributo presente in tali classi rappresenterà una colonna della tabella omonima. È interessante notare, inoltre, che le Java beans non implementano alcun metodo ad eccezione dei getter e setter per ogni attributo di classe.

Le classi **Java facade** sono invece costruite a partire dalle bean corrispondenti e rappresentano gli oggetti che implementano le operazioni e quindi anche le query, relative all'oggetto cui fanno riferimento. In questo modo Hibernate effettua una divisione tra l'informazione, ovvero le classi Java bean e gli strumenti di manipolazione, ossia le classi Java facade.

Come è stato già introdotto, Hibernate svolge il ruolo di middleware per l'applicazione rispetto al DBMS sottostante. In questo modo si rende però necessario un altro componente, ossia una classe **Java util** che, sebbene generata automaticamente dal sistema, deve essere configurata per connettersi allo specifico database. È quindi necessario intervenire a questo livello impostando la base di dati sulla quale si andrà ad operare oltre al tipo di DBMS sottostante.

4.4 File system gerarchico nascosto: lui c'è!

È stato necessario affrontare la problematica dell'effettiva memorizzazione dei file fisici contenenti le informazioni.

Ricordando che tutto il progetto poggia su un file system gerarchico, abbiamo scelto di memorizzare i dati reali proprio all'interno di una cartella nascosta del file sytem sottostante, organizzandolo in una struttura su base alfabetica fissa a 2 livelli.

4.5 La fase embrionale di TaggyFS

Dopo aver brevemente discusso delle scelte di carattere pratico relative al software già esistente utile allo sviluppo del progetto, vengono presentate le prime caratteristiche del neonato TaggyFS.

In questa fase del lavoro, è previsto che il software implementi le principali operazioni di un file system gerarchico, cercando di mantenere un'interfaccia il più possibile invariata rispetto a quest'ultimo, in modo da evitare un eccessivo disorientamento all'utente. Chiaramente ad ogni comando è stata associata una nuova semantica, grazie a Fuse, che ha permesso di ridefinire tutte le system call utilizzate dal sistema operativo per la gestione del file system. Ricordando ancora una volta che la ridefinizione di tali metodi resta valida solamente per le partizioni montate con Fuse, esistono altri aspetti fondamentali che meritano più attenzione.

4.5.1 Il SO non sa che è cambiata la semantica delle sue chiamate

Come già detto, Fuse permette di modificare il comportamento di ogni system call, ma non la sequenza e la frequenza con cui queste vengono richiamate a fronte di un comando, che viene decisa ancora in maniera esclusiva dal sistema operativo.

Tale sistema risulta inconsapevole della semantica delle chiamate, conosce solamente il mapping tra ogni comando che gli può pervenire e la sequenza di chiamate che deve attivare ordinatamente.

La difficoltà nel realizzare TaggyFS non risultava dunque la mera riscrittura delle system call, ma l'implementazione di codice in grado di svolgere operazioni semantiche di un file system a tag pur essendo chiamate in sequenza secondo i criteri di un file system gerarchico.

Per meglio esemplificare questo punto è stata aggiunta, nel capitolo successivo, una parte dedicata ai log in cui vengono mostrate le varie chiamate in successione scatenate dall'esecuzione di un comando.

4.5.2 L'integrazione con le interfacce esistenti

A fronte di un problema così grande da superare come quello appena descritto l'integrazione con il sistema operativo consente, oltre ad un doveroso studio approfondito su esso, anche di beneficiare del vantaggio di riutilizzare tutti i sistemi di output già presenti nel sistema ospite.

L'artefice di tutto questo è ancora una volta Fuse, che restituisce qualunque risultato dell'invocazione di un comando diretto verso di lui dal SO, il quale a sua volta, riconosciuta una formattazione appropriata dei risultati, si preoccuperà di visualizzarli nel modo corretto all'utente destinatario.

In questo modo, una volta scritto codice funzionante, TaggyFS sfrutta la grafica di sistema e il terminale di sistema comportandosi a tutti gli effetti come se fosse parte integrante del sistema operativo su cui è in esecuzione.

4.6 Evoluzione del progetto in itinere

La scelta di componenti reali quali Hibernate e Fuse ha consentito di fornire una traccia implementativa del progetto TaggyFS favorendo la presentazione delle caratteristiche di base; tuttavia la scelta di un prodotto rispetto ad un altro porta quasi sempre con sé l'insorgere di nuove decisioni da prendere a causa dei vincoli che il tool stesso può introdurre.

Hibernate implementa al suo interno la possibilità di gestire le relazioni tra le tabelle del database, come del resto ogni altro vincolo di integrità referenziale, funzionalità fino ad ora di esclusiva competenza del DBMS. Durante il progetto l'alternativa di lasciare a MySql il controllo di tale funzione avrebbe sicuramente richiesto meno approfondimento da parte nostra, introducendo però il limite che alla rigenerazione del codice anche il database dipendente direttamente da hibernate sarebbe stato rigenerato perdendo quindi ogni proprietà impostata. In tal caso si sarebbe dovuti intervenire ad ogni compilazione sul database per reimpostare tali proprietà.

La scelta di far gestire i controllo di integrità referenziale ad Hibernate, centralizzando in questo modo le modifiche ai soli file di configurazione del tool di middleware ha prodotto due risultati positivi a fronte di uno sforzo considerato accettabile in termini di studio.

Per prima cosa rendendo tali proprietà persistenti anche alla rigenerazione del database esse non sarebbero andate perdute, eliminando virtualmente il rischio di errori durante il reinserimento; inoltre, ha concesso a TaggyFS totale indipendenza dal DBMS sottostante purchè Hibernate lo supportasse.

Ultimo problema riscontrato in fase di sviluppo riguarda l'identificazione di una tupla all'interno del database che normalmente avviene tramite la sua chiave primaria.

Relativamente al progetto TaggyFS le chiavi primarie sono costituite rispettivamente da *nomeFile* per la tabella *file*, *nomeTag* per la tabella *Tag* e *nomeUtente* per la tabella *Utente*(nella prima versione), a causa di tutte le considerazioni sull'unicità di nome all'interno di un file system a tag e sull'unicità di un utente all'interno di un sistema.

Sfortunatamente all'interno di un file system è possibile rinominare file e cartelle oltre a poter cambiare il nome utente; questo fatto si riflette sulla base di dati in una modifica delle rispettive chiavi primarie. Hibernate implementando la teoria secondo la quale un oggetto ha un unico identificativo e se questo cambia allora l'oggetto non è più lo stesso, non permette la modifica dei campi chiave primaria degli oggetti.

Poiché l'unica soluzione, conservando l'attuale modello di database, sarebbe stata quella di effettuare una modifica mascherata, ricorrendo ad una cancellazione e creazione successiva di un nuovo oggetto, con conseguente degrado prestazionale, è stato aggiunto un identificatore univoco incrementale che ha permesso una soluzione più elegante al problema.

4.7 Il volto di TaggyFS

Dopo aver ampiamente trattato riguardo le scelte progettuali e le conseguenze che avranno sul lavoro viene presentato un modello a blocchi, il quale vuole riassumere tramite la grafica lo schema comportamentale del progetto.

Come è possibile notare il blocco relativo al DBMS utilizzato riporta MySql in quanto è il sistema effettivamente utilizzato, tuttavia sottolinea la possibilità offerta da TaggyFS di appoggiarsi ad un qualunque DBMS supportato da Hibernate senza che l'utente ne conosca virtualmente nulla.

Lo schema a blocchi mostra come qualunque applicazione accede sempre al sistema operativo sottostante per svolgere le operazioni in kernel space a prescindere dal ricevente della sua richiesta. Nel caso le richieste dell'applicazione riguardino una partizione montata con Fuse, vengono reindirizzate ancora in kernel space al modulo del VFS.

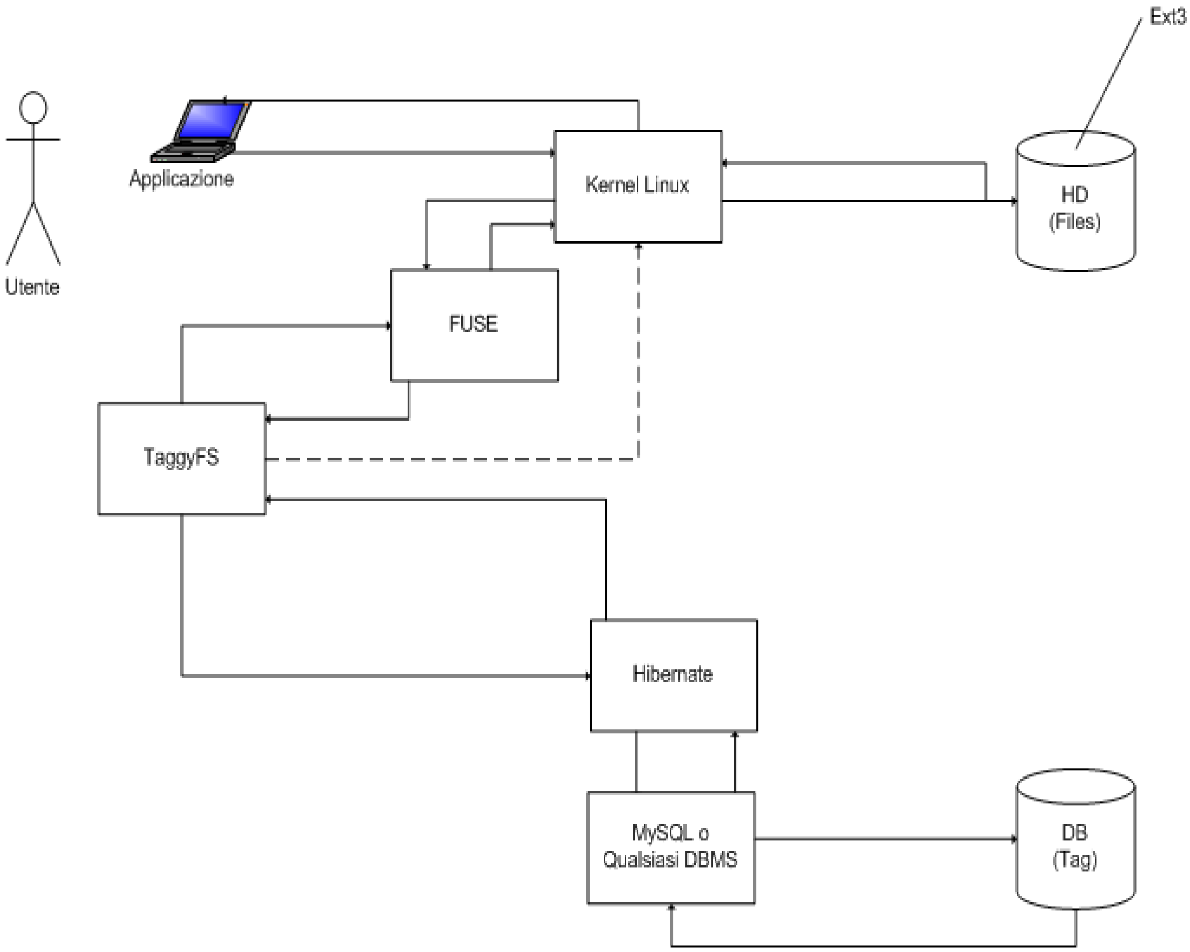

Figura 4.5: Diagramma a blocchi dell'ultima versione del progetto TaggyFS

Fuse kernel si occuperà di trasferire la chiamata a TaggyFS operante in userspace, il quale ospita l'interfaccia contenente le system call modificate.

Interagendo con Hibernate TaggyFS estrapola i dati richiesti per completare il comando li formatta in modo opportuno e li rimappa in kernel space.

Le query verranno composte, tradotte da Hibernate per lo specifico DBMS (MySql) e quindi inviate. La risposta, in caso di query ben formata, sarà costituita dall'oggetto o dalla lista di oggetti che rispondono alla richiesta effettuata, costituendo ciascuno una vera e propria copia della tupla corrispondente.

A questo punto il modulo Fuse lato kernel si occupa di restituire tali risultati al kernel Linux.

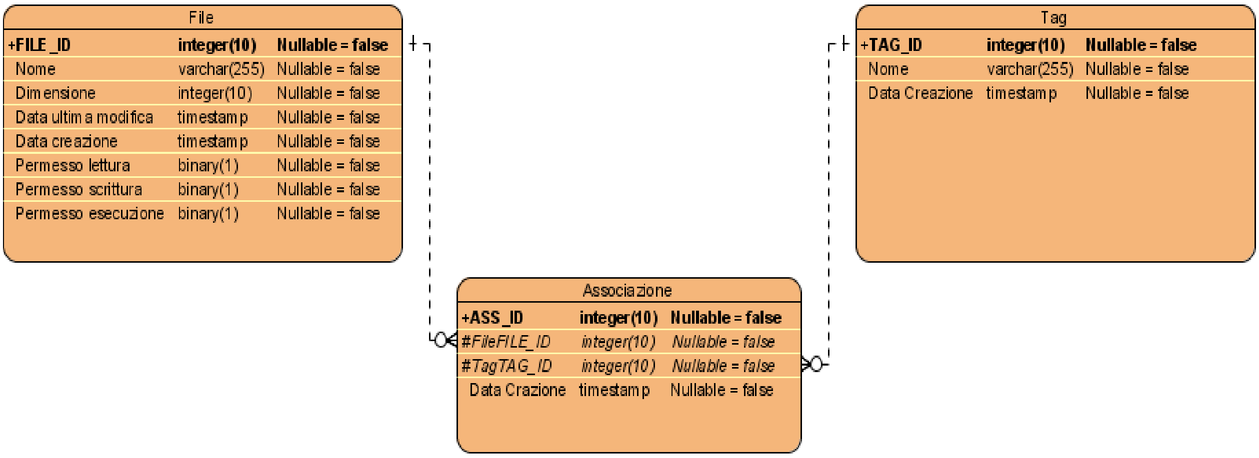

Figura 4.6: Diagramma E-R di TaggyFS

Una nota importante riguarda l'accesso ai dati veri e propri che viene effettuato in base alle informazioni reperite nel database, ma direttamente dal kernel Linux.

4.7.1 L'idea delle operazioni di TaggyFS

Viene ora presentata una breve descrizione del comportamento di TaggyFS a fronte dei principali comandi di gestione di un file system, riprendendo le stesse operazioni descritte nel capitolo precedente.

- **Nuova Cartella**: viene rispettato quanto proposto teoricamente: creare una nuova cartella (vuota) equivale a creare un nuovo tag, pronto a essere associato a uno o più file.
- **Rinomina Cartella**: anche in questo caso, nessun problema è stato riscontrato: la rinomina di una cartella è eseguita rinominando il tag corrispondente, rinominandolo per tutti quei file cui esso è associato.
- **Apri cartella**: all'apertura di una cartella si è scelto di mostrare all'utente tutti i file aventi come tag quelli presenti nel path compreso tra la root e la cartella corrente, come già visto. Si è inoltre deciso di mostrare all'utente tag

che possano permettere un raffinamento della ricerca, in quanto relazionati con quelli mostrati. Ogni tag mostrato è infatti associato a uno o a più di uno dei file visualizzati.

- **Elimina Cartella**: l'eliminazione di una cartella è stato uno dei primi punti problematici per la realizzazione di TaggyFS: poiché alla base vi è un file system gerarchico, il file system a tag genera dinamicamente (e incontrollabilmente) alberi, le cui foglie sono i file e i nodi sono gli altri tag relazionati con quelli presenti nel path. Al momento della cancellazione di un tag il file system prende l'albero dinamico attuale come statico e esegue operazioni di cancellazione ricorsive non controllabili.
- **Copia File**: la copia di un file risponde effettivamente a quanto è stato proposto, corrispondendo cioè all'aggiunta dei tag facenti parte del nuovo percorso.
- **Sposta File**: lo spostamento di un file risponde anch'esso a quanto proposto dal modello teorico, corrispondendo cioè all'aggiunta dei tag facenti parte del nuovo percorso e all'eliminazione dei tag facenti parte del vecchio.
- **Rinomina File**: l'operazione di rinomina di un file è effettivamente inalterata rispetto a quella dei file system classici.
- **Elimina File**: l'operazione di unlink per i sistemi unix è unica, mentre per un file system a tag sono necessarie due operazioni distinte, una per eliminare da un file tutti i tag cui è associato (corrispondente alla classica cancellazione del file), l'altra per eliminare l'associazione tra un tag e un file. Questo problema è dovuto al fatto che il sistema operativo individua l'unicità in base al path, mentre in un file system a tag essa è individuata dal solo nome del file. La soluzione al problema proposta nel progetto, consiste nell'introduzione di un tag univoco e fittizio detto cestino, utilizzato per effettuare l'operazione "Elimina Tag", mentre la cancellazione di un file è stata implementata allo stesso modo che in un file system gerarchico. Tramite questo tag, operante

come un buco nero, risulta possibile a livello pratico tramite un comando di move spostare un file da un tag reale al cestino stesso.

Per l'implementazione del file system sottostante tramite la *move* viene perso il riferimento al path sorgente, mentre restano conservati i dati relativi all'associazione al nuovo tag. In tal modo il file risulterà svincolato dal tag sorgente, ma poiché tutte le associazioni con il tag cestino vengono rimosse da TaggyFS, tale file non risulterà linkato nemmeno a tale tag.

4.7.2 Caratteristiche Avanzate

Dopo aver analizzato la struttura delle operazioni, che rendono TaggyFS almeno pari dal punto di vista della potenza espressiva rispetto al file system gerarchico, passiamo ad analizzare una caratteristica avanzata introdotta a seguito del restyling del progetto per fornire comunque una marcia in più a TaggyFS rispetto ai concorrenti che implementano diversi paradigmi.

Nella versione finale esso implementa un engine in grado di svolgere ricerche avanzate consentendo all'utente di formalizzare interrogazioni molto simili alle query utilizzando gli operatori logici AND,OR,NOT, [,] (da notare che TaggyFS utilizza le parentesi quadre e non quelle tonde).

Non volendo implementare un'interfaccia apposta per supportare questa funzionalità, si è cercato di integrarla come strumento offerto nell'ambito del sistema operativo stesso, in questo modo è stato necessario adattare la sintassi tipica di una query a quella del terminale di shell.

Esistono due modalità di attivazione di tale query:

1) comando di *change directory* seguito dal path composto dai tag e dagli operatori logici;

comando di *ls* sul path impostato in precedenza;

2) comando di *list* (*ls*) seguita dal path composto dai tag e dagli operatori logici.

Una query come:

"listami tutti gli mp3 di Battisti degli anni '80 che non appartengono all'album Don Giovanni"

verrà dunque tradotta con:

> ls /mp3/AND/battisti/AND/80s/AND/NOT/don giovanni

Dal punto di vista implementativo esiste un algoritmo di composizione della query SQL a partire dai dati di ingresso che viene poi indirizzata al modulo Hibernate: ad esso è lasciata l'esecuzione della query che in caso di errori sintattici genera una eccezione, che verrà reindirizzata al modulo Fuse e quindi come messaggio d'errore all'utente finale. Qualora la query risulti ben formata Hibernate ritornerà i risultati richiesti, che verranno processati e visualizzati.

Questo potente strumento è stato implementato per il solo utilizzo in modalità terminale; è tuttavia possibile portarlo in modalità grafica tramite l'utilizzo di alcuni accorgimenti quali ad esempio la visualizzazione di "cartelle-operatori" tramite cui comporre le operazioni. Malgrado questa soluzione possa apparire attraente a prima vista, durante lo sviluppo è stato però deciso di abbandonare questa alternativa, poiché avrebbe comportato soluzioni macchinose e poco fluide considerando che i desktop grafici esistenti sono espressamente sviluppati per gestire e interagire con un file system sottostante di tipo gerarchico.

Ultima considerazione di carattere pratico riguarda il metodo con cui l'utente deve considerare la composizione di una nuova query: essa viene sempre composta partendo dalla radice di TaggyFS, aggiungendo quindi eventuali parti extra nel caso non ci si trovi sulla radice. Ad esemepio:

Taggyroot~> cd foto/AND/musica

porterà in:

Taggyroot/foto/AND/musica~>

e quindi il comando ls visualizzerà tutti i file taggati con foto e musica

Taggyroot/video~>cd foto/AND/musica

porterà in:

Taggyroot/video/foto/AND/musica~>

e la successiva *ls* darà quindi errore, in quanto questa query mescola le due modalità (avanzata e non).

4.8 I modelli: uno sguardo al cuore di TaggyFS

Viene ora presentato un dettaglio dell'organizzazione reale del codice che, pur astraendo del codice stesso, fornisce una visione di come le varie parti dell'applicazione interagiscano tra loro per ottenere un risultato comune.

4.8.1 I Package

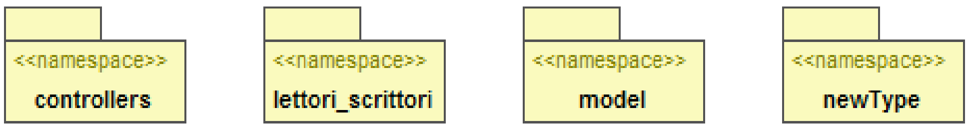

Figura 4.7: I package di TaggyFS

Questo diagramma mostra semplicemente i package in cui è diviso il progetto:

- **controllers:** contiene il controllore del database, che esegue le chiamate alle facade degli HibernateBeans e genera degli oggetti di interscambio.
- **new_tipe**: di questo package fanno parte le classi di interscambio utilizzati

da TaggyFS.

- **model**: contiene gli oggetti che modellizzano il database. Essi costituiscono gli HibernateBeans, uno per ogni tabella del database. Ad ogni HibernateBeans è associata una HibernateFacade, che contiene i metodi per invocare le query principali sul relativo oggetto.
- **lettori_scrittori**: qui si trovano gli oggetti necessari per la scrittura e la lettura dei file, necessari per la comunicazione di TaggyFS con eventuali altri file system o per altre eventuali operazioni.

Una visione di insieme è data dal prossimo diagramma: esso mostra le dipendenze dei vari package e mostra come il package *controllers* sia il cuore dell'applicazione.

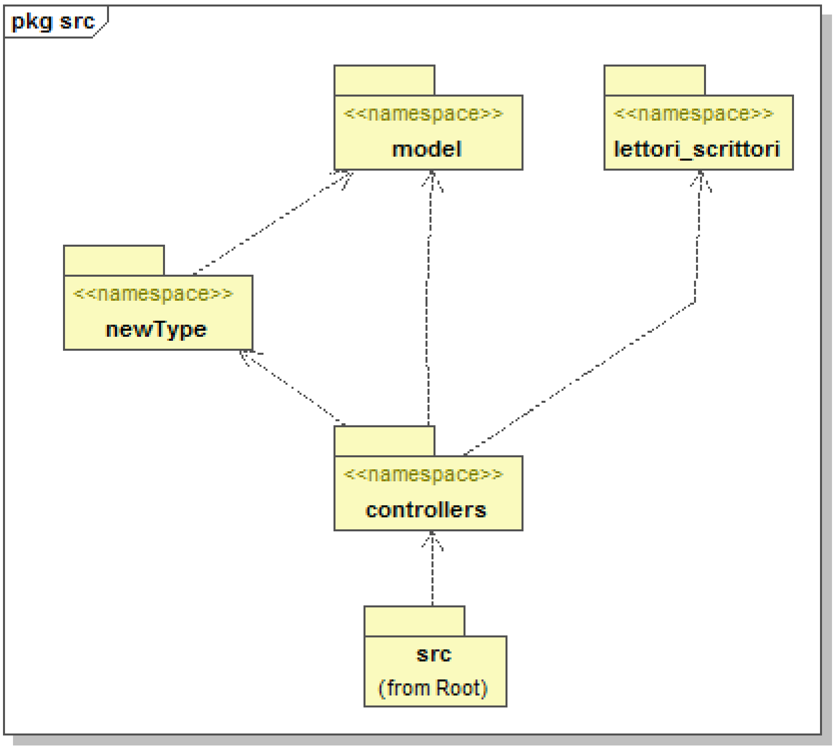

Figura 4.8: Schema delle dipendenze dei package di TaggyFS

4.8.2 Le classi

Verranno ora inseriti i grafici UML della classe *DBController* e della classe *TaggyFS.*

La classe DBColtroller si occupa della comunicazione tra Fuse e il database, esponendo quei metodi necessari per effettuare le macro-operazioni come *addTag(), delTag, writeFile(), readFile(),* ecc... Essa istanzia anche le facade necessarie per eseguire i metodi di ricerca, salvataggio e update nel database.

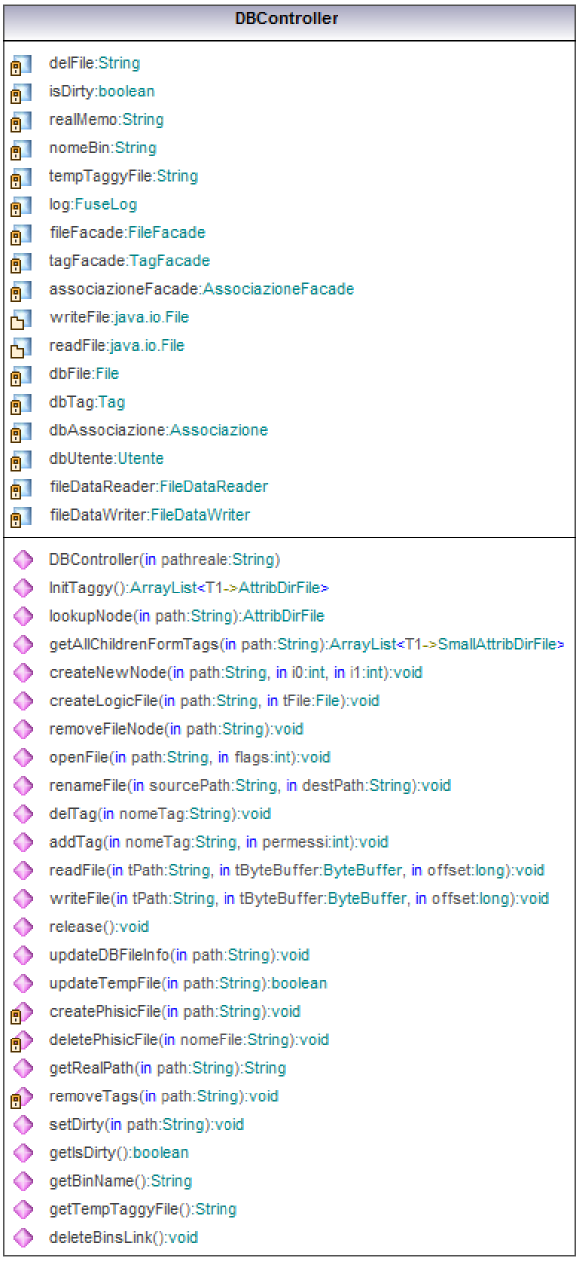

Figura 4.9: UML diagram di DBController

La classe Taggy implementa i metodi dell'interfaccia *filesystem1* di Fuse-J. Essi verranno fatti corrispondere a quelli dei moduli Fuse del kernel e verranno chiamati per eseguire le varie operazioni.

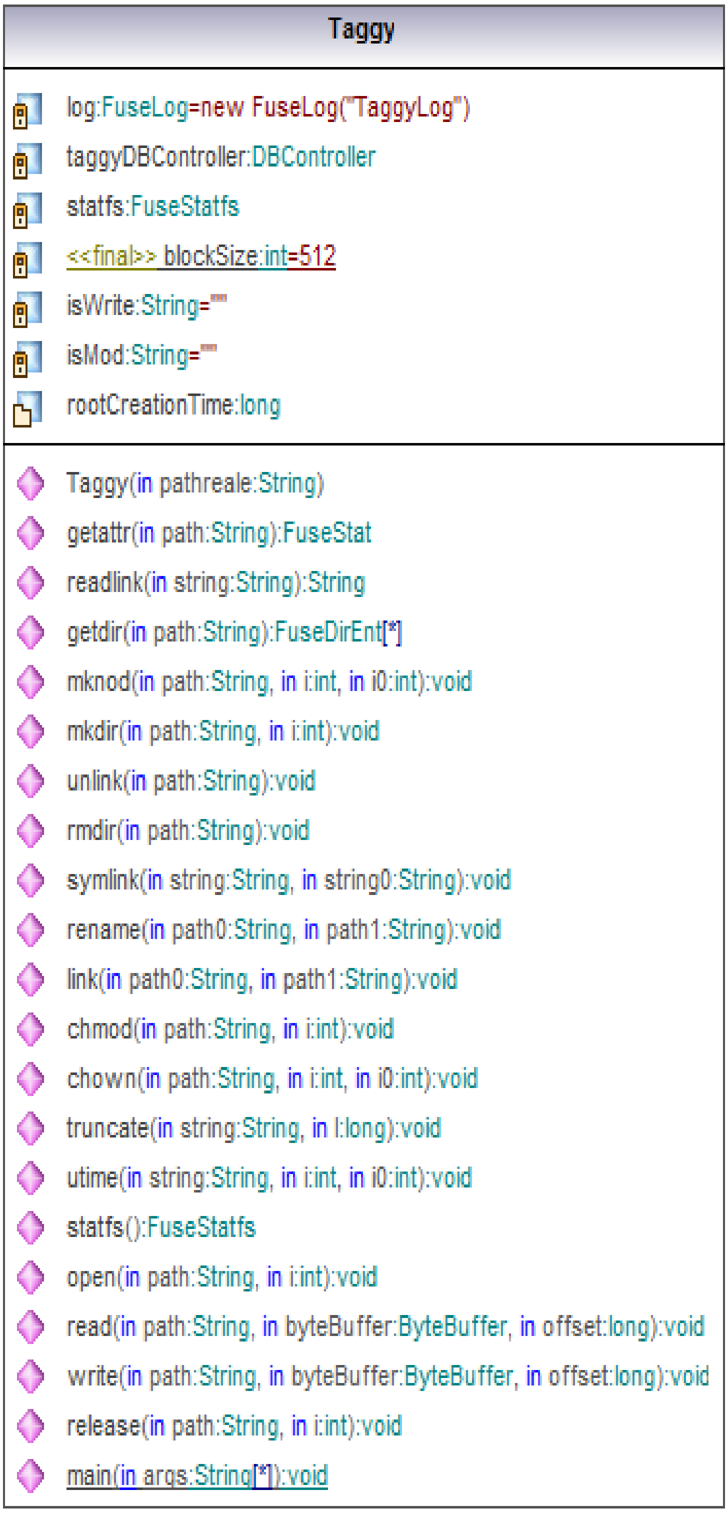

Figura 4.10: Diagramma UML della classe Taggy

5 Test

5.1 Log e screenshot

In questa sezione verranno inseriti i log delle chiamate di sistema per le operazioni principali eseguite con TaggyFS. In questo modo è data la possibilità di valutare le chiamate che un file system gerarchico effettua; questo dovrebbe risultare utile a scopo di studio. In tal modo è stato possibile effettuare un confronto tra ciò che ci aspettavamo accadesse e ciò che realmente accadeva. Vengono anche riportati alcuni screenshot riguardanti le operazioni più significative.

5.1.1 Creazione di una directory (creazione di un tag)

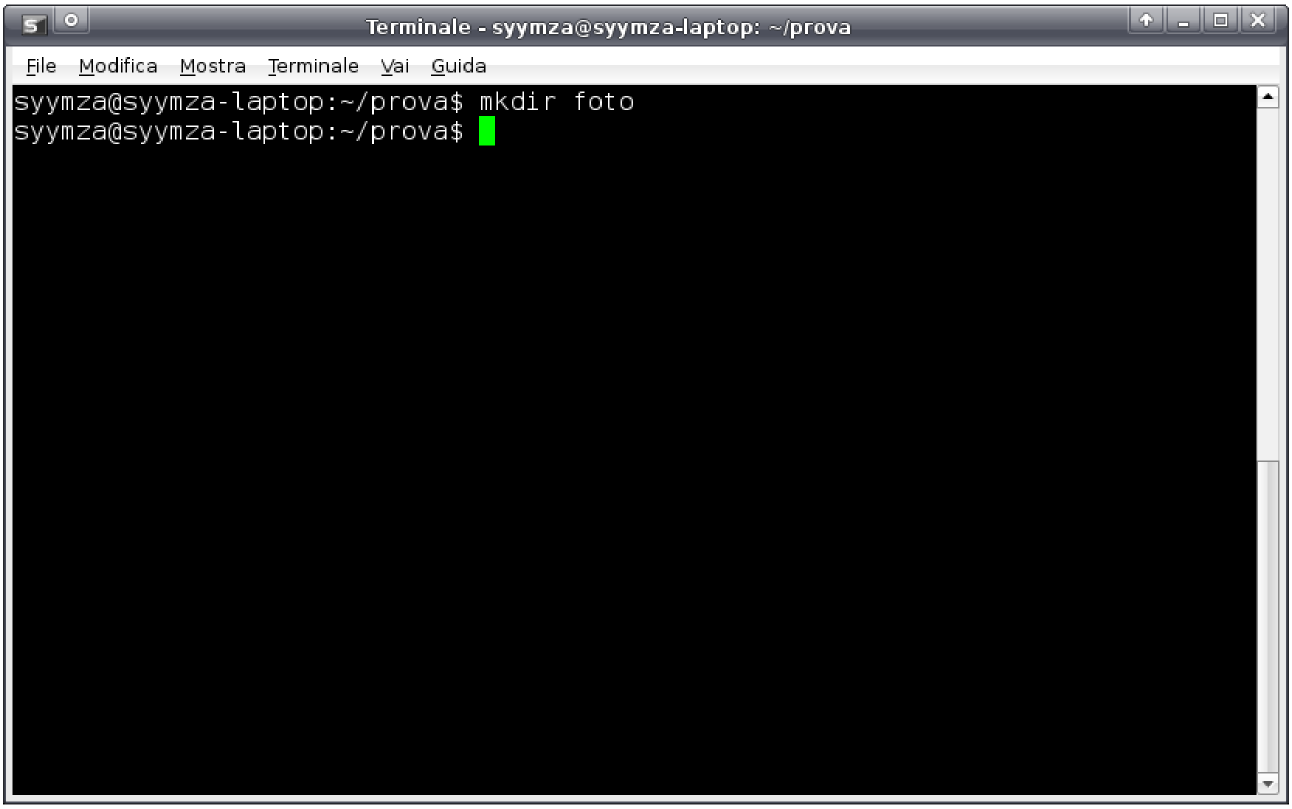

Figura 5.1: Creazione di una directory via terminale

```
02:59:18.472    main  INFO [TaggyLog]: getattr /foto
02:59:18.494    main  INFO [TaggyLog]: mkdir /foto 493
02:59:18.502    main  INFO [TaggyLog]: getattr /foto
```

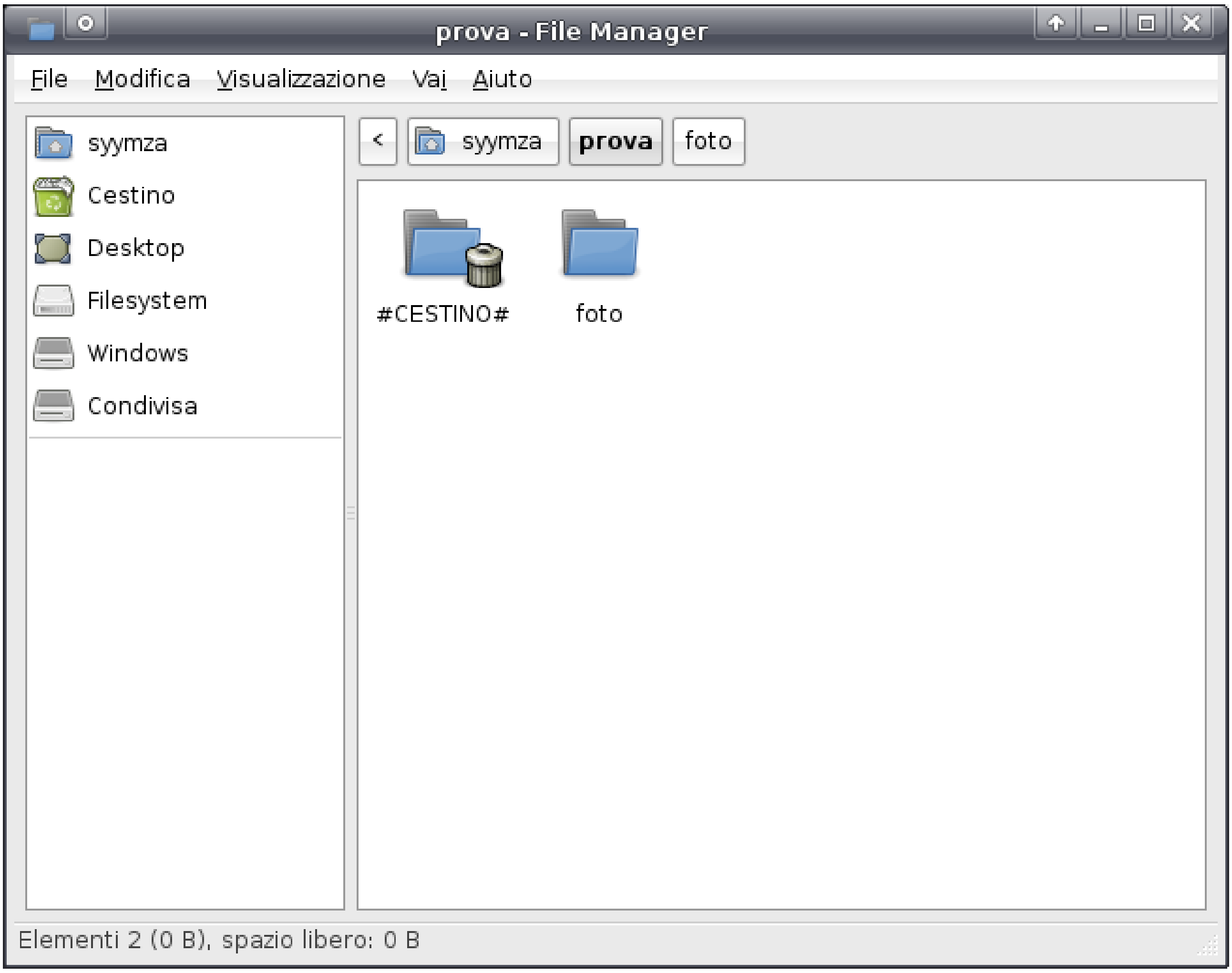

Figura 5.2: Il risultato della creazione di una directory

5.1.2 Rinomina

03:00:11.271 main INFO [TaggyLog]: getattr /foto1
03:00:11.305 main INFO [TaggyLog]: getattr /foto
03:00:11.308 main INFO [TaggyLog]: getattr /foto1
03:00:11.312 main INFO [TaggyLog]: rename /foto /foto1

5.1.3 Eliminazione di una directory (eliminazione di un tag)

Nota: per come vengono effettuate le chiamate risulta impossibile predisporre un metodo che cancelli dir solo se vuota perché il Sistema Operativo cancella prima i file al suo interno poi directory alla fine

```
03:00:57.619    main  INFO [TaggyLog]: getattr /foto1

03:00:57.622    main  INFO [TaggyLog]: getattr /
03:00:57.622    main  INFO [TaggyLog]: getattr /.Trash
03:00:57.625    main  INFO [TaggyLog]: getattr /.Trash-1000
03:00:57.628    main  INFO [TaggyLog]: getattr /.Trash-1000
03:00:57.630    main  INFO [TaggyLog]: mkdir /.Trash-1000 448
03:00:57.638    main  INFO [TaggyLog]: getattr /.Trash-1000
03:00:57.642    main  INFO [TaggyLog]: rmdir /.Trash-1000
03:00:57.667    main  INFO [TaggyLog]: getdir /foto1
03:00:57.689    main  INFO [TaggyLog]: getattr /foto1
03:00:57.692    main  INFO [TaggyLog]: getattr /
03:00:57.693    main  INFO [TaggyLog]: getattr /.Trash
03:00:57.696    main  INFO [TaggyLog]: getattr /.Trash-1000
03:00:57.698    main  INFO [TaggyLog]: getattr /.Trash-1000
03:00:57.701    main  INFO [TaggyLog]: mkdir /.Trash-1000 448
03:00:57.705    main  INFO [TaggyLog]: getattr /.Trash-1000
03:00:57.708    main  INFO [TaggyLog]: rmdir /.Trash-1000
03:00:57.714    main  INFO [TaggyLog]: rmdir /foto1
```

5.1.4 Copia file dall'esterno

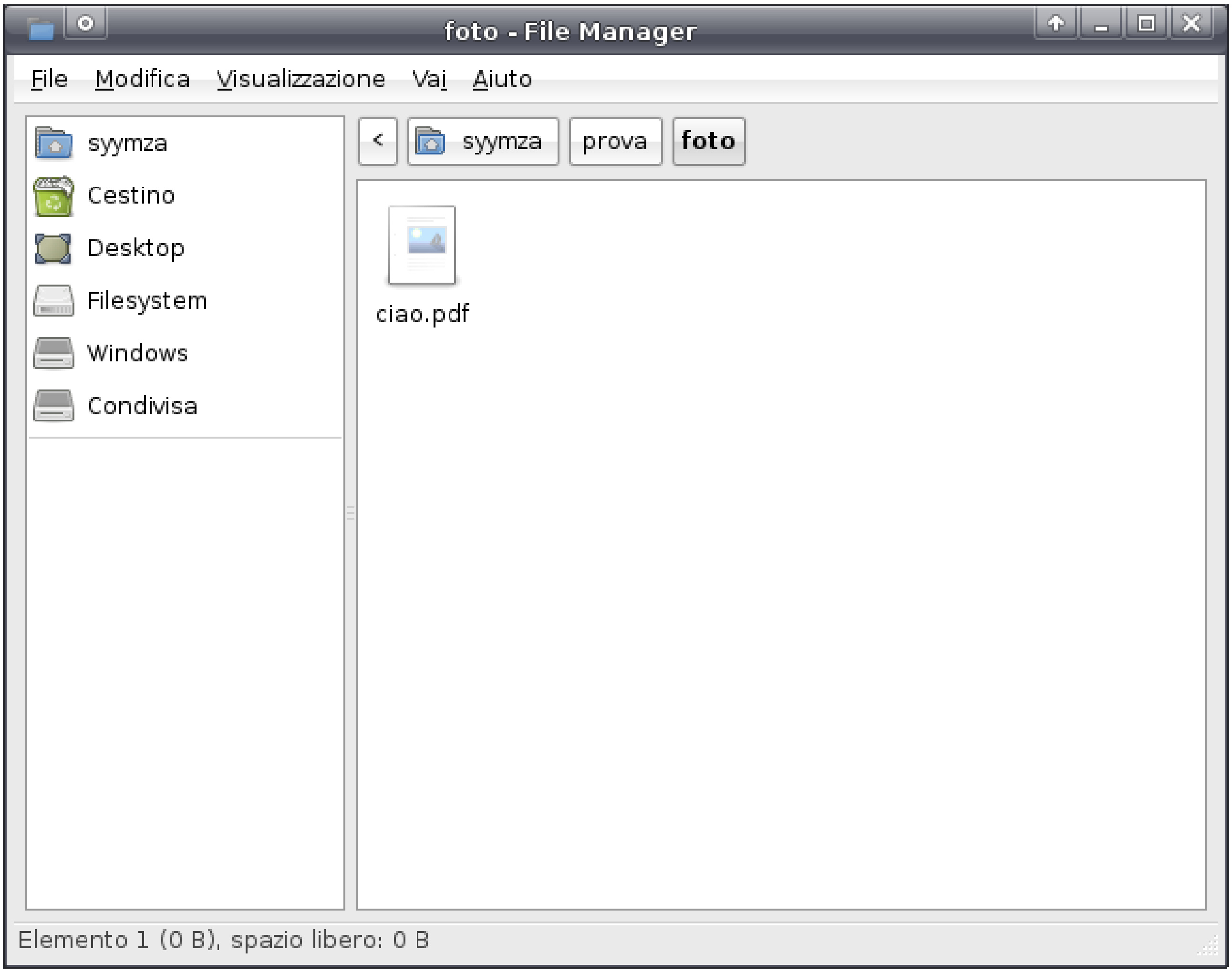

Figura 5.3: Il contenuto di una cartella dopo la copia di un file

03:04:06.770 main INFO [TaggyLog]: getattr /foto
03:04:06.774 main INFO [TaggyLog]: getattr /foto/Ciao.pdf
03:04:06.778 main INFO [TaggyLog]: mknod /foto/Ciao.pdf
03:04:06.791 main INFO [TaggyLog]: getattr /foto/Ciao.pdf
03:04:06.811 main INFO [TaggyLog]: open /foto/Ciao.pdf 32769
//varie write dipendono dalla lunghezza del file
03:04:06.815 main INFO [TaggyLog]: write /foto/Ciao.pdf java.nio.DirectByteBuffer[pos=0 lim=4096 cap=4096] 0
03:04:06.817 main INFO [TaggyLog]: write /foto/Ciao.pdf java.nio.DirectByteBuffer[pos=0 lim=4096 cap=4096] 4096
[...]
03:04:06.916 main INFO [TaggyLog]: write /foto/Ciao.pdf java.nio.DirectByteBuffer[pos=0 lim=4096 cap=4096] 442368
03:04:06.916 main INFO [TaggyLog]: write /foto/Ciao.pdf java.nio.DirectByteBuffer[pos=0 lim=2320 cap=2320] 446464
03:04:06.921 main INFO [TaggyLog]: getattr /foto/Ciao.pdf
03:04:06.929 main INFO [TaggyLog]: release /foto/Ciao.pdf 32769

5.1.5 Cancellazione di un file

03:06:32.071 main INFO [TaggyLog]: getattr /foto
03:06:32.073 main INFO [TaggyLog]: getattr /foto/ciao.pdf
03:06:32.080 main INFO [TaggyLog]: getattr /
03:06:32.080 main INFO [TaggyLog]: getattr /.Trash
03:06:32.082 main INFO [TaggyLog]: getattr /.Trash-1000
03:06:32.084 main INFO [TaggyLog]: getattr /.Trash-1000
03:06:32.086 main INFO [TaggyLog]: mkdir /.Trash-1000 448
03:06:32.093 main INFO [TaggyLog]: rmdir /.Trash-1000
03:06:32.097 main INFO [TaggyLog]: getattr /
03:06:32.098 main INFO [TaggyLog]: getattr /.Trash
03:06:32.100 main INFO [TaggyLog]: getattr /.Trash-1000
03:06:32.102 main INFO [TaggyLog]: getattr /.Trash-1000
03:06:32.104 main INFO [TaggyLog]: mkdir /.Trash-1000 448
03:06:32.106 main INFO [TaggyLog]: getattr /.Trash-1000
03:06:32.109 main INFO [TaggyLog]: rmdir /.Trash-1000
03:06:32.113 main INFO [TaggyLog]: open /foto/ciao.pdf 32768
03:06:32.124 main INFO [TaggyLog]: getattr /
03:06:32.124 main INFO [TaggyLog]: read /foto/ciao.pdf java.nio.DirectByteBuffer[pos=0 lim=131072 cap=131072] 0
03:06:32.125 main INFO [TaggyLog]: read /foto/ciao.pdf java.nio.DirectByteBuffer[pos=0 lim=65536 cap=65536] 131072
03:06:32.126 main INFO [TaggyLog]: read /foto/ciao.pdf java.nio.DirectByteBuffer[pos=0 lim=65536 cap=65536] 196608
03:06:32.127 main INFO [TaggyLog]: read /foto/ciao.pdf java.nio.DirectByteBuffer[pos=0 lim=20480 cap=20480] 262144
03:06:32.127 main INFO [TaggyLog]: release /foto/ciao.pdf 32768
03:06:32.131 main INFO [TaggyLog]: ulink/foto/ciao.pdf
03:06:33.238 main INFO [TaggyLog]: getattr /foto

5.1.6 Copia di un file interno a TaggyFS (aggiunta di un tag)

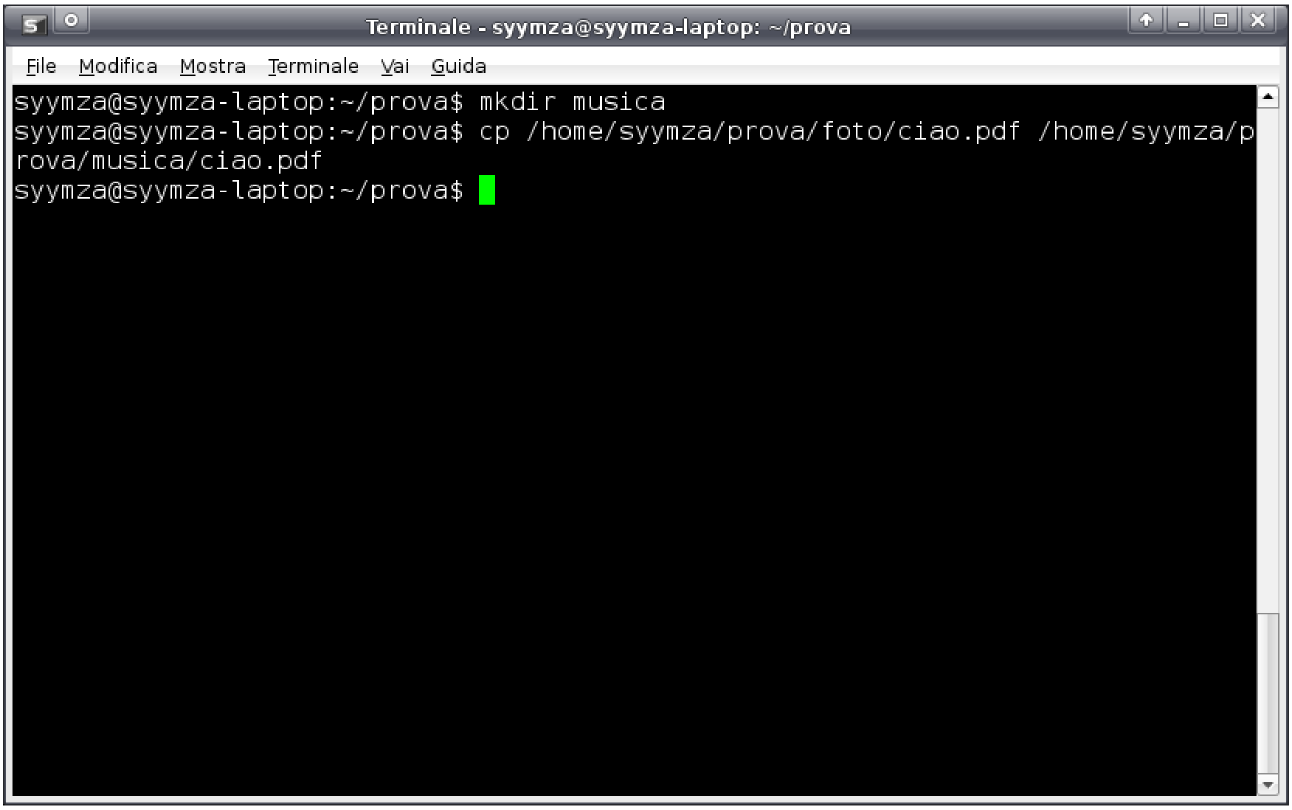

Figura 5.4: Copia di un file interno a TaggyFS

```
03:12:22.407    main  INFO [TaggyLog]: getattr /foto
03:12:22.408    main  INFO [TaggyLog]: getattr /foto/ciao.pdf
03:12:22.411    main  INFO [TaggyLog]: open /foto/ciao.pdf 32768
03:12:22.413    main  INFO [TaggyLog]: read /foto/ciao.pdf java.nio.DirectByteBuffer[pos=0 lim=16384 cap=16384] 0
03:12:22.419    main  INFO [TaggyLog]: release /foto/ciao.pdf 32768
03:12:23.437    main  INFO [TaggyLog]: getattr /foto
03:12:23.440    main  INFO [TaggyLog]: getattr /foto/ciao.pdf
03:12:23.445    main  INFO [TaggyLog]: open /foto/ciao.pdf 32768
03:12:23.450    main  INFO [TaggyLog]: getattr /
03:12:23.451    main  INFO [TaggyLog]: getattr /musica
03:12:23.452    main  INFO [TaggyLog]: getattr /musica/ciao.pdf
03:12:23.455    main  INFO [TaggyLog]: mknod /musica/ciao.pdf
03:12:23.470    main  INFO [TaggyLog]: getattr /musica/ciao.pdf
03:12:23.473    main  INFO [TaggyLog]: open /musica/ciao.pdf 32769
03:12:23.476    main  INFO [TaggyLog]: read /foto/ciao.pdf java.nio.DirectByteBuffer[pos=0 lim=77824 cap=77824] 0
03:12:23.476    main  INFO [TaggyLog]: write /musica/ciao.pdf java.nio.DirectByteBuffer[pos=0 lim=4096 cap=4096] 0
03:12:23.476    main  INFO [TaggyLog]: write /musica/ciao.pdf java.nio.DirectByteBuffer[pos=0 lim=4096 cap=4096] 4096
03:12:23.476    main  INFO [TaggyLog]: write /musica/ciao.pdf java.nio.DirectByteBuffer[pos=0 lim=4096 cap=4096] 8192
03:12:23.477    main  INFO [TaggyLog]: write /musica/ciao.pdf java.nio.DirectByteBuffer[pos=0 lim=4096 cap=4096] 12288
03:12:23.477    main  INFO [TaggyLog]: write /musica/ciao.pdf java.nio.DirectByteBuffer[pos=0 lim=4096 cap=4096] 16384
03:12:23.477    main  INFO [TaggyLog]: write /musica/ciao.pdf java.nio.DirectByteBuffer[pos=0 lim=4096 cap=4096] 20480
03:12:23.477    main  INFO [TaggyLog]: write /musica/ciao.pdf java.nio.DirectByteBuffer[pos=0 lim=4096 cap=4096] 24576
03:12:23.477    main  INFO [TaggyLog]: write /musica/ciao.pdf java.nio.DirectByteBuffer[pos=0 lim=4096 cap=4096] 28672
03:12:23.477    main  INFO [TaggyLog]: write /musica/ciao.pdf java.nio.DirectByteBuffer[pos=0 lim=4096 cap=4096] 32768
03:12:23.478    main  INFO [TaggyLog]: write /musica/ciao.pdf java.nio.DirectByteBuffer[pos=0 lim=4096 cap=4096] 36864
03:12:23.478    main  INFO [TaggyLog]: write /musica/ciao.pdf java.nio.DirectByteBuffer[pos=0 lim=4096 cap=4096] 40960
03:12:23.478    main  INFO [TaggyLog]: write /musica/ciao.pdf java.nio.DirectByteBuffer[pos=0 lim=4096 cap=4096] 45056
03:12:23.478    main  INFO [TaggyLog]: write /musica/ciao.pdf java.nio.DirectByteBuffer[pos=0 lim=4096 cap=4096] 49152
03:12:23.478    main  INFO [TaggyLog]: write /musica/ciao.pdf java.nio.DirectByteBuffer[pos=0 lim=4096 cap=4096] 53248
03:12:23.478    main  INFO [TaggyLog]: write /musica/ciao.pdf java.nio.DirectByteBuffer[pos=0 lim=4096 cap=4096] 57344
03:12:23.479    main  INFO [TaggyLog]: write /musica/ciao.pdf java.nio.DirectByteBuffer[pos=0 lim=4096 cap=4096] 61440
```

```
03:12:23.479    main  INFO [TaggyLog]: write /musica/ciao.pdf java.nio.DirectByteBuffer[pos=0 lim=4096 cap=4096] 65536
03:12:23.479    main  INFO [TaggyLog]: write /musica/ciao.pdf java.nio.DirectByteBuffer[pos=0 lim=4096 cap=4096] 69632
03:12:23.479    main  INFO [TaggyLog]: write /musica/ciao.pdf java.nio.DirectByteBuffer[pos=0 lim=3816 cap=3816] 73728
03:12:23.479    main  INFO [TaggyLog]: getattr /musica/ciao.pdf
03:12:23.482    main  INFO [TaggyLog]: release /musica/ciao.pdf 32769
03:12:23.485    main  INFO [TaggyLog]: release /foto/ciao.pdf 32768
03:12:23.501    main  INFO [TaggyLog]: open /musica/ciao.pdf 32768
03:12:23.503    main  INFO [TaggyLog]: read /musica/ciao.pdf java.nio.DirectByteBuffer[pos=0 lim=16384 cap=16384] 0
03:12:23.507    main  INFO [TaggyLog]: release /musica/ciao.pdf 32768
03:12:24.799    main  INFO [TaggyLog]: getattr /musica
03:12:24.801    main  INFO [TaggyLog]: getattr /musica/ciao.pdf
03:12:24.805    main  INFO [TaggyLog]: open /musica/ciao.pdf 32768
03:12:24.807    main  INFO [TaggyLog]: read /musica/ciao.pdf java.nio.DirectByteBuffer[pos=0 lim=16384 cap=16384] 0
03:12:24.808    main  INFO [TaggyLog]: release /musica/ciao.pdf 32768
03:12:24.810    main  INFO [TaggyLog]: getattr /musica/ciao.pdf
03:12:24.813    main  INFO [TaggyLog]: open /musica/ciao.pdf 32768
03:12:24.815    main  INFO [TaggyLog]: read /musica/ciao.pdf java.nio.DirectByteBuffer[pos=0 lim=16384 cap=16384] 0
03:12:24.819    main  INFO [TaggyLog]: release /musica/ciao.pdf 32768
```

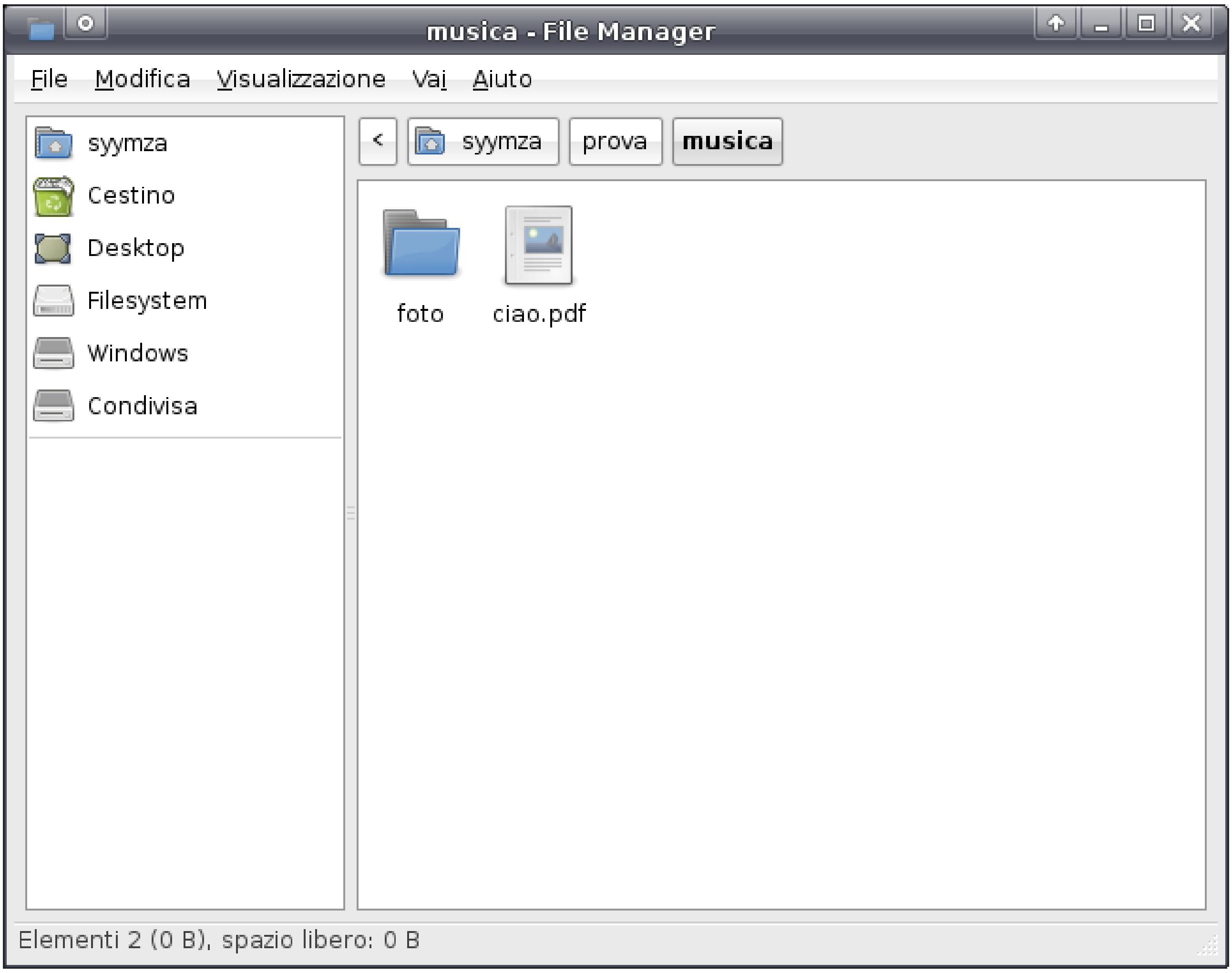

Figura 5.5: Risultato dell'aggiunta di un tag

5.1.7 Eliminazione di un tag foto tramite il cestino (eliminazione di un tag da un file)

```
03:17:05.959    main  INFO [TaggyLog]: getattr /foto
03:17:05.960    main  INFO [TaggyLog]: getattr /foto/Ciao.pdf
03:17:05.962    main  INFO [TaggyLog]: open /foto/Ciao.pdf 32768
03:17:05.965    main  INFO [TaggyLog]: read /foto/Ciao.pdf java.nio.DirectByteBuffer[pos=0 lim=16384 cap=16384] 0
03:17:05.968    main  INFO [TaggyLog]: release /foto/Ciao.pdf 32768
03:17:09.299    main  INFO [TaggyLog]: getattr /foto
03:17:09.302    main  INFO [TaggyLog]: getattr /foto/Ciao.pdf
03:17:09.305    main  INFO [TaggyLog]: open /foto/Ciao.pdf 32768
03:17:09.308    main  INFO [TaggyLog]: read /foto/Ciao.pdf java.nio.DirectByteBuffer[pos=0 lim=16384 cap=16384] 0
03:17:09.308    main  INFO [TaggyLog]: release /foto/Ciao.pdf 32768
03:17:18.553    main  INFO [TaggyLog]: getattr /#CESTINO#
03:17:18.555    main  INFO [TaggyLog]: getattr /foto
03:17:18.556    main  INFO [TaggyLog]: getattr /#CESTINO#/Ciao.pdf
03:17:18.561    main  INFO [TaggyLog]: getattr /foto/Ciao.pdf
03:17:18.567    main  INFO [TaggyLog]: getattr /#CESTINO#/Ciao.pdf
03:17:18.581    main  INFO [TaggyLog]: rename /foto/Ciao.pdf /#CESTINO#/Ciao.pdf
```

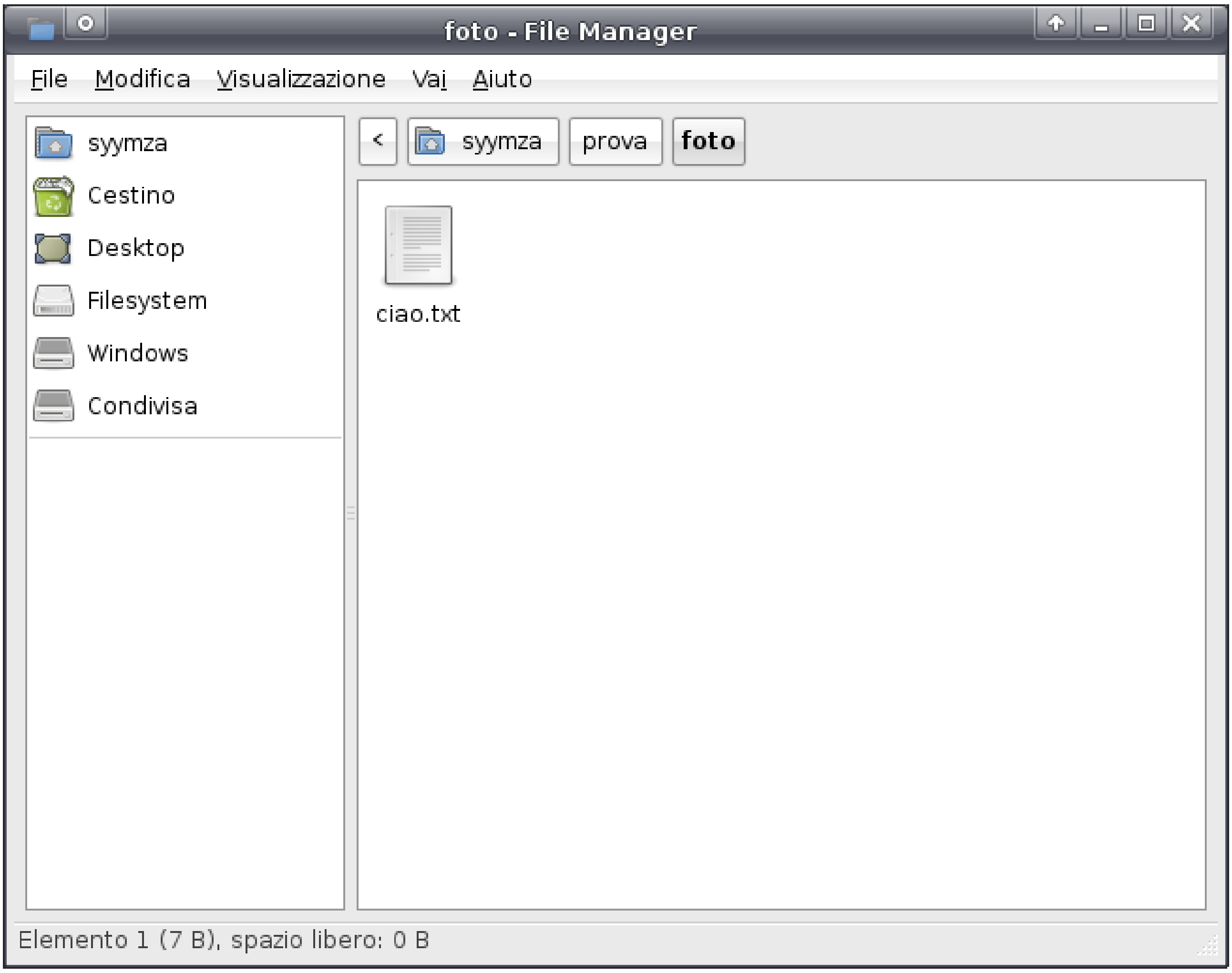

Figura 5.6: Risultato dell'eliminazione di un tag da un file

5.1.8 Rinomina di un file

```
03:42:13.455 main INFO [TaggyLog]: getattr /musica/ciao.pdf
03:42:13.458 main INFO [TaggyLog]: getattr /musica/Ciao.pdf
03:42:13.461 main INFO [TaggyLog]: getattr /musica/ciao.pdf
03:42:13.463 main INFO [TaggyLog]: rename /musica/Ciao.pdf /musica/ciao.pdf
03:42:13.471 main INFO [TaggyLog]: open /musica/ciao.pdf 32768
03:42:13.473 main INFO [TaggyLog]: read /musica/ciao.pdf java.nio.DirectByteBuffer[pos=0 lim=16384 cap=16384] 0
03:42:13.473 main INFO [TaggyLog]: release /musica/ciao.pdf 32768
03:42:13.479 main INFO [TaggyLog]: open /musica/ciao.pdf 32768
03:42:13.482 main INFO [TaggyLog]: getattr /musica/ciao.pdf
03:42:13.486 main INFO [TaggyLog]: read /musica/ciao.pdf java.nio.DirectByteBuffer[pos=0 lim=16384 cap=16384] 0
03:42:13.486 main INFO [TaggyLog]: release /musica/ciao.pdf 32768
03:42:13.489 main INFO [TaggyLog]: open /musica/ciao.pdf 32768
03:42:13.492 main INFO [TaggyLog]: getattr /musica/ciao.pdf
03:42:13.496 main INFO [TaggyLog]: read /musica/ciao.pdf java.nio.DirectByteBuffer[pos=0 lim=16384 cap=16384] 0
03:42:13.515 main INFO [TaggyLog]: release /musica/ciao.pdf 32768
03:42:13.519 main INFO [TaggyLog]: getattr /musica/ciao.pdf
03:42:13.532 main INFO [TaggyLog]: open /musica/ciao.pdf 32768
03:42:13.534 main INFO [TaggyLog]: read /musica/ciao.pdf java.nio.DirectByteBuffer[pos=0 lim=16384 cap=16384] 0
03:42:13.534 main INFO [TaggyLog]: getattr /musica/ciao.pdf
03:42:13.537 main INFO [TaggyLog]: read /musica/ciao.pdf java.nio.DirectByteBuffer[pos=0 lim=4096 cap=4096] 446464
03:42:13.538 main INFO [TaggyLog]: getattr /musica/ciao.pdf
03:42:13.541 main INFO [TaggyLog]: read /musica/ciao.pdf java.nio.DirectByteBuffer[pos=0 lim=4096 cap=4096] 442368
03:42:13.542 main INFO [TaggyLog]: read /musica/ciao.pdf java.nio.DirectByteBuffer[pos=0 lim=4096 cap=4096] 430080
03:42:13.543 main INFO [TaggyLog]: read /musica/ciao.pdf java.nio.DirectByteBuffer[pos=0 lim=4096 cap=4096] 401408
03:42:13.544 main INFO [TaggyLog]: read /musica/ciao.pdf java.nio.DirectByteBuffer[pos=0 lim=4096 cap=4096] 364544
03:42:13.545 main INFO [TaggyLog]: read /musica/ciao.pdf java.nio.DirectByteBuffer[pos=0 lim=4096 cap=4096] 413696
03:42:13.545 main INFO [TaggyLog]: read /musica/ciao.pdf java.nio.DirectByteBuffer[pos=0 lim=4096 cap=4096] 409600
03:42:13.546 main INFO [TaggyLog]: read /musica/ciao.pdf java.nio.DirectByteBuffer[pos=0 lim=12288 cap=12288] 417792
03:42:13.549 main INFO [TaggyLog]: read /musica/ciao.pdf java.nio.DirectByteBuffer[pos=0 lim=8192 cap=8192] 434176
03:42:13.551 main INFO [TaggyLog]: read /musica/ciao.pdf java.nio.DirectByteBuffer[pos=0 lim=16384 cap=16384] 368640
03:42:13.552 main INFO [TaggyLog]: read /musica/ciao.pdf java.nio.DirectByteBuffer[pos=0 lim=16384 cap=16384] 385024
03:42:13.553 main INFO [TaggyLog]: read /musica/ciao.pdf java.nio.DirectByteBuffer[pos=0 lim=4096 cap=4096] 405504
03:42:13.574 main INFO [TaggyLog]: release /musica/ciao.pdf 32768
03:42:13.583 main INFO [TaggyLog]: open /musica/ciao.pdf 32768
03:42:13.586 main INFO [TaggyLog]: getattr /musica/ciao.pdf
03:42:13.590 main INFO [TaggyLog]: read /musica/ciao.pdf java.nio.DirectByteBuffer[pos=0 lim=16384 cap=16384] 0
03:42:13.590 main INFO [TaggyLog]: release /musica/ciao.pdf 32768
03:42:14.787 main INFO [TaggyLog]: getattr /musica
03:42:14.789 main INFO [TaggyLog]: getattr /musica/ciao.pdf
03:42:14.792 main INFO [TaggyLog]: open /musica/ciao.pdf 32768
03:42:14.794 main INFO [TaggyLog]: read /musica/ciao.pdf java.nio.DirectByteBuffer[pos=0 lim=16384 cap=16384] 0
03:42:14.795 main INFO [TaggyLog]: release /musica/ciao.pdf 32768
```

6 Conclusioni

Il prodotto ottenuto risulta stabile nelle parti sviluppate, presentandosi con potere espressivo almeno pari rispetto ad un file system tradizionale, consentendo una organizzazione nuova dei file, basata sui tag; ogni file è quindi in relazione con altri file proprio grazie a questi tag.

In realtà, il progetto implementa già alcune caratteristiche avanzate che mettono in luce le potenzialità di un siffatto sistema di gestione delle informazioni.

Tuttavia esistono ancora diverse funzionalità che potrebbero essere sviluppate in futuro quali ad esempio:

- Gestione degli utenti multipli, con creazione di tag privati e di tag pubblici.
- Sistema di reasoning per creare relazioni semantiche ancora più forti tra i file, creando ad esempio cluster di tag o relazioni fra i tag.
- Recupero di tag intrinseci dai file, ad esempio le proprietà Idv3 dagli mp3.
- Sviluppo di un'integrazione con il sistema operativo che consenta l'espressione di ricerche avanzate in modalità grafica.

Viene ribadito comunque che questo rimane semplicemente un'introduzione a cui abbiamo avuto il piacere di lavorare data la novità del campo, poiché questi sistemi sono in rapida diffusione soprattutto nel web.

La nostra speranza rimane quindi quella che lo sviluppo di TaggyFS non termini, in virtù delle molteplici possibilità di espansione e dell'innovatività; ci piacerebbe quindi immaginare questo lavoro come piattaforma di lancio per altri che vogliano approfondire le loro conoscenze in quest'ambito, ben consci della scarsità e caoticità del materiale a disposizione sulla rete Internet.

Un'ultima nota importante, invita chiunque a riflettere sull'idea che il paradigma dei file system a tag non sia la soluzione a tutti i problemi di organizzazione di dati, ma semplicemente un'idea innovativa e ricca dal punto di vista teorico, che in alcuni contesti, quali ad esempio i gruppi di lavoro, consente effettivamente una gestione più versatile ed efficiente. Diversamente, se usato in modo improprio, potrebbe risultare addirittura più scadente dei tradizionali paradigmi su cui si basano gli attuali file system.

Bibliografia

[1] Peter Lyman and Hal R. Varian. How much information, 2003.
http://www.sims.berkeley.edu/how-much-info-2003, 2003.

[2] Deborah Barreau and Bonnie A. Nardi. Finding and reminding: file organization from the desktop. SIGCHI Bull., 1995.

[3] Yahoo!, http://www.yahoo.it

[4] Google, http://www.google.com

[5] Del.icio.us, http://del.icio.us

[6] Rashmi Sinha, A cognitive analysis of tagging, 2005,
http://www.rashmisinha.com/archives/05_09/tagging-cognitive.html

[7] Home page del progetto GLS^3, http://www.glscube.org/

[8] Home page del progetto TAGS, http://mac.softpedia.com/get/Utilities/Tags.shtml

[9] Home page del progetto RelFS, http://relfs.sourceforge.net/

[10] Home page del progetto SemFS, http://semweb4j.org/site/semfs/

[11] Home page del progetto Tagsistant, http://www.tagsistant.net/

[12] Home page del progetto Fuse, http://fuse.sourceforge.net/

[13] Home page del progetto Hibernate, http://www.hibernate.org/

Indice delle immagini

Figura 1.1: Un esempio di struttura ad albero....8
Figura 1.2: Schema a blocchi di un sistema operativo contenente un VFS....11
Figura 2.1: Esempio di folksonomia....15
Figura 2.2: Screenshot delle web-directory di Yahoo!....17
Figura 2.3: Uno screenshot del risultato di una ricerca su Google....18
Figura 2.4: Uno screenshot di una didascalia su Del.icio.us....19
Figura 2.5: Processi cognitivi alla base della categorizzazione per un sistema gerarchico....21
Figura 2.6: Processi cognitivi alla base del "tagging"....21
Figura 2.7: La creazione dei vincoli semantici e l'eliminazione dei vincoli strutturali....22
Figura 2.8: Visione insiemistica dei file in un file system a tag....23
Figura 2.9: Uno screenshot di Tags....24
Figura 3.1: Modello a blocchi delle componenti principali in un file system a tag....26
Figura 4.1: Modello di un sistema con il modulo Fuse attivato....31
Figura 4.2: Schema di funzionamento di Fuse....33
Figura 4.3: Hibernate come middleware tra l'applicazione e il database....34
Figura 4.4: La hompage di Hibernate....35
Figura 4.5: Diagramma a blocchi dell'ultima versione del progetto TaggyFS....41
Figura 4.6: Diagramma E-R di TaggyFS....42
Figura 4.7: I package di TaggyFS....46
Figura 4.8: Schema delle dipendenze dei package di TaggyFS....47
Figura 4.9: UML diagram di DBController....48
Figura 4.10: Diagramma UML della classe Taggy....49
Figura 5.1: Creazione di una directory via terminale....50
Figura 5.2: Il risultato della creazione di una directory....51
Figura 5.3: Il contenuto di una cartella dopo la copia di un file....52
Figura 5.4: Copia di un file interno a TaggyFS....54
Figura 5.5: Risultato dell'aggiunta di un tag....55
Figura 5.6: Risultato dell'eliminazione di un tag da un file....56