UNIVERSITÀ DI PADOVA **FACOLTÀ DI INGEGNERIA**

TESI DI LAUREA

PARIMULO: OTTIMIZZAZIONE DEL FILESHARING

Laureando: Stefano Pelizzaro

Relatore: Prof. Enoch Peserico Stecchini Negri De Salvi
Correlatore: Ing. Paolo Bertasi

Corso di Laurea Triennale in Ingegneria Informatica

Data Laurea

Anno Accademico 2011/2012

Alla mia famiglia

Sommario

Dall'avvento delle prime reti, le peculiarità del *filesharing* hanno contribuito ad un drastico abbattimento delle barriere geografiche relative alla connettività globale, permettendo lo scambio di file tra persone poste a chilometri e chilometri di distanza. Ognuno con un proprio proposito, esite una grande varietà di client software che si interfacciano a queste reti. In una realtà in cui solo i software più diffusi riescono a sopravvivere lo sviluppo di un client deve presentare delle caratteristiche innovative o ottimizzate rispetto agli altri software in circolazione al fine di favorire la sua diffusione. In questa tesi viene trattato lo sviluppo di una serie di ottimizzazioni sul funzionamento di *Mulo*, un plugin di *PariPari* che implementa un client *eMule/eDonkey*.

Inizialmente viene affrontata l'implementazione di un sistema di *hashing parallelo*, che permette una maggiore efficienza nel procedimento di condivisione di un file nella rete *eMule/eDonkey*, al fine di diminuire il tempo impiegato per la condivisione di un file.

Successivamente viene affrontato il lavoro svolto per il supporto della compressione in *Mulo* implementando la gestione dei pacchetti compressi utilizzati nella rete *eMule*. Il supporto alla compressione dei dati permette a *Mulo* di scambiare dati con gli altri client più rapidamente.

Infine vengono presentate le basi di quello che in futuro sarà la prima implementazione funzionante di un client di filesharing multi-protocollo. Nella tesi viene illustrata l'ideazione e il funzionamento di *SuperFilesharing* per poi spiegare le problematiche e i successi incontrati negli sviluppi iniziali di questo progetto.

Indice

Capitolo 1

Introduzione

In questo capitolo viene presentata una panoramica dei concetti che ricorrono lungo il proseguimento della tesi. Viene inizialmente spiegato il progetto *PariPari* elencando le principali caratteristiche e i suoi obiettivi, proseguendo con una breve trattazione delle reti *eDonkey/eMule* per infine introdurre il plugin *Mulo*, sul quale è stato svolto il lavoro di questa tesi.

1.1 Il progetto PariPari

PariPari è una rete *peer-to-peer*[1] (P2P) *serverless*[2] multifunzionale il cui software è sviluppato in linguaggio *Java*. La rete è progettata per garantire l'anonimato ai propri *peer* e la sua natura *serverless* la rende una rete robusta e scalabile.

PariPari nasce con lo scopo di unificare tutta quella serie di servizi legati alla *connettività* dove ai giorni nostri vige una profonda classificazione, soprattutto dal punto di vista software. Il settore software oggigiorno offre una grande varietà di applicazioni incentrate sulla *connettività*, tuttavia ogni programma è per lo più specializzato nell'offrire solo un determinato tipo di servizio spingendo l'utente a utilizzare più applicazioni per soddisfare le sue necessità. In questo modo l'utilizzo di più software, se non gestiti opportunamente, possono condurre a un calo prestazionale o addirittura al decadimento dei servizi offerti da questi. Basti pensare alle interazioni che vengono a crearsi tra un client P2P e un client VoIP (ad esempio *Skype*[3]) utilizzati nello stesso momento sulla stessa macchina: la qualità del servizio Voip cala drasticamente quando il client P2P inizia a scaricare grosse quantità di dati, soprattutto nel caso si abbia a disposizione una connessione molto lenta. Con *PariPari* questo tende a non accadere perché il software cerca di distribuire in modo adeguato le risorse disponibili per garantire la fruibilità dei servizi utilizzati.

[1] Con *peer-to-peer* si intende una rete di computer o qualsiasi rete informatica che non possiede nodi gerarchizzati come client o server fissi, ma un numero di nodi equivalenti (peer) che fungono sia da cliente che da servente verso altri nodi della rete.

[2] Con il termine serverless ("senza server") si intende un network la cui gestione non viene incentrata su dei server.

[3] http://www.skype.it/

Figura 1.1: Logo Ufficiale di PariPari.

Dal punto di vista software *PariPari* si presenta come un'applicazione Java per la rete. È caratterizzata da una forte **modularità** ed è costituita da un modulo centrale, denominato *Core*, e da un insieme di "plugin". Ogni plugin estende le funzionalità del Core permettendo di usufruire di vari servizi sulla rete Internet. L'utente, col solo utilizzo del software *PariPari*, ha la possibilità di accedere ai contenuti offerti dalle reti *BitTorrent*, *eDonkey* e *Kad* [4]; è in grado di conversare liberamente con *VoIP*[5]; è libero di utilizzare canali *IRC* [6] e chat *IM*[7]; gli è consentito usufruire di servizi come *DBMS*[8], *DHT*[9] e *WebServer*[10], il tutto caricando dal Core i relativi plugin.

I plugin, per essere caricati nel modulo principale, devono implementare delle *API*[11] ben definite. In questa maniera viene a crearsi un livello di astrazione a *black box*[12] tra i plugin ed il Core in cui le modifiche interne ad un singolo plugin risultano trasparenti al Core e quindi a tutti gli altri plugin rendendo così più semplice l'organizzazione del progetto. In *PariPari* i plugin sono classificati in due grandi categorie (Figura 1.2): i plugin della *cerchia interna*, che hanno lo scopo di fornire le funzionalità di base agli altri plugin, e quelli della *cerchia esterna* che forniscono le varie funzionalità all'utente finale. I plugin della cerchia interna sono principalmente due:

- il plugin *Storage*, che si occupa di gestire gli accessi alla memoria di massa, permet-

[4] *BitTorrent*, *eDonkey* e *Kad* sono protocolli finalizzati alla distribuzione e condivisione di file nella rete.

[5] Voice over IP è una tecnologia che rende possibile effettuare una conversazione telefonica sfruttando una connessione Internet.

[6] Internet Relay Chat.

[7] Istant Messaging.

[8] Database Management System.

[9] Distributed Hash Table.

[10] WebServer è un servizio he si occupa di fornire, tramite software dedicato e su richiesta dell'utente, file di qualsiasi tipo, tra cui pagine web.

[11] Application Programming Interface API - Interfaccia di Programmazione di un'Applicazione.

[12] Black box è un livello di astrazione che permette di vedere un sistema similmente ad una scatola nera, ovvero descrivibile solo per come reagisce (output) ad una determinata sollecitazione (input), ma i cui ingranaggi non sono visibili.

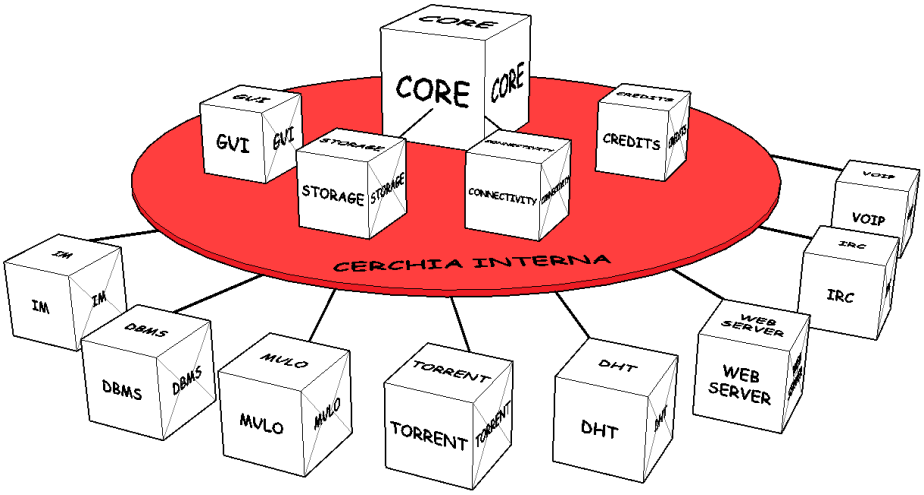


Figura 1.2: Struttura del progetto PariPari.

tendo creazione,lettura e scrittura dei file.

- il plugin *Connectivity*, che si occupa di instaurare e gestire le connessioni di rete.

In secondo piano a questi troviamo il plugin *Credits* che si occupa di organizzare e gestire tutto il sistema di Crediti di *PariPari*. Come detto in precedenza, *PariPari* cerca di distribuire in modo opportuno le risorse disponibili ai plugin; a questo scopo viene eretto attraverso Credits un sistema di crediti in cui avviene una "monetizzazione" delle risorse permettendo così al Core di decidere se un certo plugin è abilitato o meno ad accedere alla risorsa richiesta.

Altro plugin degno di nota è *GUI* la cui mansione principale è fornire un iterfaccia utente "user friendly" così da garantire all'utente facilità di utilizzo e un'esperienza positiva nell'uso di *PariPari*. Attualmente GUI si orienta su un'interaccia di tipo Web (una scelta adottata anche da molti software del mercato) a cui è possibile accedere tramite browser. Tra i plugin della cerchia esterna troviamo invece tutti quei plugin che forniscono all'utente l'accesso ai vari servizi precedentemente descritti:

Mulo Plugin per l'interfacciamento alle reti di filesharing eDonkey/Kad.

Torrent Plugin per l'interfacciamento alla rete BitTorrent.

IRC* e *IM Plugin per le funzioni di messaggistica istantanea.

Voip Plugin per effettuare conferenze o videoconferenze multiutente.

DBMS

DHT

WebServer

Dal lato organizzativo, lo sviluppo del progetto è coordinato sfruttando l'indipendenza di Core e plugin. Ogni modulo viene seguito ed esteso da un team di sviluppo dedicato costituito di un numero variabile di sviluppatori e *tester*[13]. Il processo di sviluppo tenta di seguire le linee guida dell'*Extreme Programming*[14], alternando brevi fasi implementative a fasi di testing in modo da produrre codice con minima presenza di errori e soprattutto cercando di evitare eventuali ripercussioni degli errori in tutto il codice. In questo frangente ogni sviluppatore assolve anche al ruolo di tester, testando il codice prodotto dai colleghi.

In *PariPari* sono implicati più di 100 persone tra sostenitori e studenti delle facoltà dei rami di Ingegneria dell'Informazione, suddivisi in un totale di 14 gruppi(uno per ogni plugin, più i gruppi per Core e testing). Ogni gruppo viene regolato da un *team leader*. Al vertice di tutto è posto il *project manager*, nonché ideatore e sostenitore del progetto. Il progetto è distribuito sotto licenza GPL[15], e mette a disposizione sorgenti e documentazione del codice, seguendo la filosofia *"open source"*[16].

1.2 eDonkey, eMule e Kad

Attorno al termine *eDonkey* ruotano tre concetti principali: quello di *rete*, di *client* e di *protocollo di rete*. Un *protocollo* è definito come un insieme di regole formali che due macchine tra loro connesse devono rispettare per poter instaurare un canale di comunicazione. L'ideazione di protocolli è quindi necessaria per poter far comunicare elaboratori interconnessi tra loro. Tuttavia solo macchine che rispettano lo stesso protocollo possono comunicare costruttivamente tra loro, da questo aspetto sorge l'evidente analogia tra Protocollo e Linguaggio. Alla base dei fatti possiamo generalizzare il concetto di Linguaggio come un insieme di regole formali atte a garantire la comunicazione tra un individuo ed un'altro, osservando così che il linguaggio non è altro che una sorta di protocollo anch'esso. Come esistono diversi tipi di linguaggi possono coesistere differenti tipi di protocolli, ognuno ideato per uno scopo preciso.

Un *client*, in ambito software, non è altro che un'applicazione che permette ad una macchina interconnessa di percepire e comprendere un determinato protocollo di rete, per il quale il client è stato creato. Il client è quindi un programma e può essere sviluppato in differenti linguaggi. È possibile che più client "affini" siano implementati per il medesimo protocollo e nulla vieta che un client implementi il supporto di uno o più protocolli.

Una *rete* viene invece concepita come un insieme di client "affini" interconnessi tra loro (*nodi*) che comunicano utilizzando un protocollo ben definito. I servizi offerti dalla rete dipendono quindi direttamente dalle potenzialità offerte dal protocollo di base utilizzato, come da esso dipendono sicurezza, stabilità e struttura della rete. Da questo punto di vi-

[13]Un tester ha il compito di effettuare il collaudo del software.

[14]L'Extreme Programming è una metodologia agile e un approccio all'ingegneria del software formulato da Kent Beck, Ward Cunningham e Ron Jeffries.

[15]La GNU General Public License è una licenza per software libero.

[16]La filosofia open source prevede software i cui autori ne permettono il libero studio e l'apporto di modifiche da parte di altri programmatori indipendenti.

sta sussistono due tipologie di reti: *client-server*, che si basano sull'idea che un particolare nodo(server) offre le sue risorse a tutti gli altri; o *peer-to-peer* in cui ogni nodo contemporaneamente offre le proprie risorse e beneficia delle risorse altrui. Tra queste due tipologie convivono poi tutta una serie di sfumature che cercano di trarre i vantaggi dalle due diverse configurazioni; è il caso delle reti "ibride".

eDonkey, inteso come rete, è una rete *ibrida* (principalmente P2P) che permette ai suoi utenti la condivisione dei file (*filesharing*). La rete distingue due tipologie di nodi:

- I **peer** *eDonkey* (i client *eDonkey2000*) che permettono all'utente di condividere i propri file con gli altri utenti della rete, e contemporaneamente di richiedere i file condivisi dagli altri utenti.

- I **server** *eDonkey* che hanno lo scopo di indicizzare i contenuti messi a disposizione della rete dai vari utenti. I server si occupano di fornire all'utente, che vuole scaricare un determinato file contenuto nella rete, un elenco di utenti della rete che possiedono il file in questione da i quali ottenerlo.

Il protocollo *eDonkey* prevede che ogni peer si connetti alla rete scegliendo un server *eDonkey* di riferimento. A tale server il peer comunica quindi i file che intende condividere con la rete. Il server provvede così ad indicizzare i nuovi contenuti condivisi dal peer rendendoli disponibili in tutta la rete. Qualora il peer voglia scaricare un file *target.txt*, procede a interpellare il server di riferimento che restituirà una lista degli altri peer della rete in possesso totale o parziale del file (*fonti*). A questo punto il peer contatta le farie fonti richiedendo le varie parti del file in loro possesso, al fine di ottenerlo completamente. Il protocollo prevede inoltre tutta una serie di *messaggi*[17] utilizzati per le varie interazioni peer-peer e peer-server, per un analisi più approfondita delle specifiche del protocollo *eDonkey* si faccia riferimento a [eDonkeyProtocol].

La rete *eDonkey* e il client originale *eDonkey2000* vengono realizzati dall'azienda americana MetaMachine nel 2000. Fin dalla nascita l'utilizzo di *eDonkey* nel mondo cresce enormemente, affermandosi fin da subito come il software P2P di condivisione file più diffuso. Lo sviluppo del progetto subisce però un brusco colpo nel settembre del 2006, quando MetaMachine viene citata in giudizio sotto accusa della *RIAA*, venendosi costretta a cessare il mantenimento della rete. Nel frattempo però, il protocollo *eDonkey* cresceva portando con se nuove estensioni e nuovi client.

eMule[18] è un client *eDonkey* sviluppato in linguaggio *C++*. Attualmente è il client per filesharing più diffuso nel mondo e si basa pienamente sul protocollo *eDonkey* apportando una serie di estensioni del protocollo per migliorare le prestazioni della rete, prime fra tutte la *compressione dei pacchetti*, il *sistema AICH*[19] e l'*estensione a 64 bit*. *eMule* tuttavia eredita i punti deboli del suo predecessore: come su *eDonkey* infatti la rete *eMule* posa la sua stabilità sui server. Il loro funzionamento è di vitale importanza per la sopravvivenza della rete. Per questo modivo il client *eMule* implementa in se il supporto ad un protocollo

[17] Nel protocollo *eDonkey* i messaggi consistono in pacchetti TCP e UDP appositamente strutturati.
[18] http://www.emule-project.net/
[19] Advanced Intelligent Corruption Handling è un sistema di controllo dell'errore ideato da *eMule*.

alternativo: *Kad*. *Kad* è la soluzione che *eMule* utilizza per *decentralizzare* la rete *eDonkey*. Con *Kad* viene rimosso il concetto di server, assegnando il compito di indicizzare i contenuti in maniera distribuita ai vari nodi della rete *eMule*. *Kad* implementa questo con l'utilizzo una *DHT* adattando opportunamente il protocollo *Kademlia*[20], da cui prende il nome. In questo modo la rete *eMule* può riuscire a sopravvivere indipendentemente dalla sopravvivenza dei server.

1.3 Mulo

Mulo è uno dei plugin di *PariPari* (Sezione 1.1) che ha il compito di implementare un client *eDonkey/eMule* per permettere a *PariPari* di interfacciarsi con queste reti. Il suo scopo non è però solamente quello di effettuare un *porting* del codice di eMule, come già propongono software quali *aMule*[21], *JMule*[22], *Lphant*[23] ed altri. *Mulo* vuole invece proporre delle soluzioni aggiornate e *migliorate* rispetto alla sua controparte più famosa. *Mulo* vede il codice di *eMule* come qualcosa da cui ispirarsi e soprattutto come traguardo da superare.

Figura 1.3: Logo Ufficiale del plugin Mulo.

Alla stesura di questa tesi il plugin *Mulo* conta più di *52* file sorgenti per un totale di *35484* righe di codice, alle quali vanno aggiunte un'estesa documentazione ed un gran numero di classi per il *testing* del codice. La grande difficoltà nello sviluppo del client *Mulo* è data dalla mancanza di specifiche del protocollo *eMule/eDonkey*. Gran parte dei documenti a riguardo soffrono infatti di mancanze o imprecisioni, inoltre per questo motivo ogni tipo di client *eDonkey* interpreta in maniera personale il protocollo rendendone più complesso l'apprendimento. Principalmente *Mulo* si affida al codice di *eMule* (distribuito sotto licenza GPL) per reperire eventuali informazioni sul protocollo *eMule/eDonkey*; non

[20]Kademlia è un protocollo di rete peer-to-peer per un network di computer decentralizzato.
[21]http://www.amule.org/
[22]http://jmule.org/
[23]http://www.lphant.com/

tanto perché *eMule* lo implementi correttamente, ma in quanto client più diffuso. *Mulo* supporta al momento la quasi totalità del protocollo *eDonkey* ed è in continuo sviluppo per supportare le varie estensioni di protocollo *eMule*. La carenza più critica di *Mulo* è la mancanza di supporto all'*estensione a 64 bit*, la quale permette la condivisione dei file di dimensione superiore ai *4 GB*. Il protocollo *eDonkey* prevede che la dimensione di ogni file debba essere esprimibile con al massimo 32 bit, da qui il nome di *protocollo a 32 bit*. Questo apporta un limite alla dimensione massima di un file condivisibile nella rete *eDonkey* di 2^{32} *byte* (B), ovvero *4 GB*.

$$2^{32} = 4\,294\,967\,296$$

Tale limite limita molto l'utilizzo della rete, per questo *eMule* apporta un estensione al protocollo di *eDonkey*: l'*estensione a 64 bit*. Con questa estensione, per la dimensione dei file, sono a disposizione ben 64 bit. Consentendo la condivisione di file con una grandezza fino a 2^{64} B, ovvero:

$$2^{64} = 18\,446\,744\,073\,709\,551\,616$$

Il limite è studiato per essere esaustivo e non apportare limitazioni alla condivisione dei file nella rete.

Mulo si interfaccia alla rete *eMule/eDonkey* attraverso connessioni *TCP*[24] e, secondariamente, a connessioni *UDP*[25]. I pacchetti scambiati sono quelli standard definiti dal protocollo *eDonkey* e sono caratterizzati da un *header* comune a tutti i pacchetti.

Campo	Size [B]	Descrizione
`Protocol`	1	Descrive il tipo di protocollo a cui il pacchetto fa riferimento. Ha tipicamente 3 possibili valori: **0xE3** se il pacchetto è del protocollo *eDonkey*, **0xC5** se si riferisce alle estensioni *eMule*, **0xD4** se si tratta di un pacchetto del protocollo compresso.
`Size`	4	Indica la lunghezza totale del pacchetto escludendone i primi 6 byte corrispondenti all'header.
`Type`	1	Identifica il tipo di pacchetto del protocollo e definisce univocamente la sua struttura: il tipo di informazioni che esso porta, il suo significato e la disposizione dei campi da cui è composto.

Tabella 1.1: Struttura header dei pacchetti *eMule/eDonkey* TCP

In Tabella 3.1 troviamo la struttura di un header per un pacchetto di tipo TCP, per quanto riguarda i pacchetti utilizzati nelle connessioni UDP, l'header si compone dei soli campi `Protocol` e `Type`. Il procollo *eMule/eDonkey* definisce molti pacchetti rivolti alle comunicazioni coi server e alle comunicazioni coi peer. Per funzionare *Mulo* si serve principalmente di queste connessioni e di una serie di file XML, sui quali mantiene informazioni riguardanti i dati relativi a server e file in dowload/upload, un file di log sul quale stampa

[24]Transmission Control Protocol.
[25]User Datagram Protocol

notifiche o errori avvenuti durante l'esecuzione ed un file contenente i parametri di configurazione. *Mulo* è infatti altamente configurabile e permette di impostare a piacimento le porte utilizzate per le connessioni (preimpostate ai valori standard: 4662 per TCP e 4672 per UDP), il *nickname*[26] da utilizzare nella rete, i limiti di download e upload, il massimo numero di *uploader*[27] e le directory in cui salvare i file scaricati. *Mulo* mette poi a disposizione una lista iniziale di server che può essere completamente gestita dall'utente. L'utente è in grado di aggiungere o rimuovere manualmente server alla lista, nonché *scaricare nuove liste server* fornendo gli opportuni link.

Mulo implementa un sistema di *ranking dei server* che permette l'*autologin* alla rete *eMule/eDonkey* in modo da connettersi utilizzando i server migliori presenti nella lista (si veda [DiPieri] per approfondimenti). Al momento della connessione *Mulo* è in grado di avvertire l'utente se è connesso alla rete *eMule/eDonkey* come peer *High ID* o *Low ID*. Il protocollo *eDonkey* prevede infatti che ogni peer della rete venga identificato univocamente da un *User ID*. Questo *User ID* è un numero calcolato alla connessione del peer alla rete. Il peer infatti contatta il proprio server di riferimento ed il server procede ad effettuare l'*handshake eDonkey*[28], per verificare se il peer è in grado di accettare connessioni TCP in entrata. Nel caso l'handshake vada a buon fine, il server assegna al peer un *User ID* calcolato a partire dal suo *indirizzo IP*. In questo caso, un utente con indirizzo IP *a.b.c.d* avrà *User ID* calcolato come:

$$a + b \cdot 2^8 + c \cdot 2^{16} + d \cdot 2^{24}$$

Nel caso invece il test fallisca il server provede ad assegnare un *User ID* prendendo casualmente un numero positivo strettamente inferiore a 2^{24}. È quindi intuibile che un peer venga classificato *Low ID* quando il suo *User ID* sia strettamente inferiore a 2^{24}, mentre venga classificato come *High ID* quando il suo *User ID* sia maggiore di tale valore. La classificazione *Low ID High ID* è molto importante in quanto definisce se un peer è o meno in grado di accettare connessioni TCP dagli altri peer della rete; questo significa che un peer classificato *High ID* è molto più avvantaggiato rispetto ad un client *Low ID*. Mentre con i peer *High ID* le connessioni vengono accettate direttamente, con i peer *Low ID* si deve invece ricorrere al meccanismo della *callBack*. Un peer *High ID* contatta il peer *Low ID* attraverso il server. Il server quindi comunica al peer *Low ID* che il peer *High ID* vuole contattarlo, in modo che sia lo stesso peer *Low ID* ad instaurare la connessione. Da questa spiegazione emerge subito l'impossibilità che due peer *Low ID* si contattino in quanto nessuno dei due è in grado di accettare connessioni in ingresso. Questo limita molto la disponibilità di fonti della rete ai peer *Low ID*, che possono solo utilizzare fonti *High ID*. Anche i peer *High ID* non sono però esenti da queste problematiche; è il caso di quei peer che pur potendo accettare connessioni in ingresso presentano un indirizzo IP del tipo *a.b.c.*0. In questo caso, per loro sfortuna, lo *User ID* calcolato risulta un valore

[26] Un nickname è uno pseudonimo usato dagli utenti di Internet per identificarsi in una determinata comunità virtuale

[27] Un uploader è un peer della rete a cui stanno venendo inviati dei dati.

[28] Letteralmente "stretta di mano" l'handshake consiste in uno scambio di pacchetti tra peer e server che permette al peer di entrare a far parte della rete.

strettamente minore si 2^{24} risultando così dei *Low ID* agli occhi degli altri peer. *Mulo* individua quindi nel momento in cui si connette alla rete se si viene riconosciuti come *High ID* o *Low ID* e in quest'ultimo caso provvede ad elencare delle possibili cause; per lo più legate alla presenza di router o firewall.

Mulo offre all'utente ben tre modalità di ricerca sulla rete *eMule/eDonkey*. L'utente può utilizzare la ricerca standard prevista da *eDonkey*, collegandosi ad un server di riferimento e contattandolo per ricercare un particolare file nella rete. In aggiunta a questa modalità, *Mulo* permette anche di effettuare ricerche collegandosi alla rete *Kad* di *eMule* permettendo così una ricerca dei file senza la necessità di contattare alcun server. Oltre a queste mulo implementa una modalità di ricerca innovativa: la *ricerca globale*. Attraverso la ricerca globale *Mulo* effettua la ricerca dei file relativi alle *keyword*[29] date dall'utente contattando tutti i server disponibili ed effettuando in parallelo una ricerca nella rete *Kad* (si veda [Piccolo] per dettagli sulla sua implementazione), restituendo così molti più risultati che consentono all'utente una maggiore scelta. Tale tecnica è utilizzata anche nell'ambito della *ricerca delle fonti*; in questa maniera il numero di fonti reperite aumenta notevolmente rispetto agli standard *eMule*, traducendosi in un punto a favore di *Mulo*.

Riguardo alle procedure di download, *Mulo* si presenta con un motore in grado di gestire i download più efficacemente della controparte *eMule*. Il motore, in particolare, ottimizza le fasi iniziali e finali della procedura di download, scaricando inizialmente i frammenti di file più rari nella rete e al termine riassegnando a fonti più rapide gli ultimi frammenti da scaricare. Queste migliorie diminuiscono i tempi di download evitando i rallentamenti finali, tipici nei client come *eMule*.

Mulo possiede poi altre importanti caratteristiche di *eMule*: dal sistema AICH per la correzione degli errori ,col quale è possibile rimediare efficientemente allo scaricamento di frammenti di file corrotti (si veda [Muscarella] per una presentazione più dettagliata), al supporto del sistema di crediti e di *identificazione sicura*[30] in *eMule*, per il quale si rimanda a [Daberdaku], passando per il supporto dei pacchetti compressi, approfondito al Capitolo 3.

[29] Letteralmente "parole chiave" le keywork sono stringhe che forniscono dei parametri di ricerca.
[30] L'identificazione sicura è un sistema atto a evitare che peer malevoli si spaccino per altri peer.

Capitolo 2

Hashing Parallelo

In questo capitolo viene illustrato il lavoro svolto per l'implementazione del sistema di Hashing parallelo di Mulo. Viene prima fornita una panoramica sulle funzioni di hash e su come queste vengono utilizzate dalle reti *eMule/eDonkey* per il loro funzionamento, per poi spiegare i passi che hanno portato all'implementazione vera e propria del sistema.

2.1 Le funzioni di Hash in eMule

Prima di addentrarsi nei concetti degli *hash* legati contesto della rete *eMule/eDonkey* è necessario definire alcuni concetti primari.

Definizione 2.1 *Si definisce* ***funzione hash*** *come una funzione* ***non iniettiva*** *che mappa una stringa di lunghezza arbitraria in una stringa di lunghezza predefinita chiamata* ***hash****.*

In prima analisi quindi una *funzione hash* associa ad un determinato insieme di stringhe di lunghezza variabile un ben definito insieme di stringhe di lunghezza fissa (*hash*). Analizzando questi insiemi si può dimostrare come la cardinalità del primo insieme sia decisamente superiore alla cardinalità del secondo insieme. Questo introduce uno dei problemi principali riguardanti lo studio delle funzioni hash: *le collisioni.*

Definizione 2.2 *Siano date due stringhe di lunghezza* ***arbitraria*** x, y *con* $x \neq y$ *ed una funzione di hash* $h(\cdot)$*. Viene definita* ***collisione*** *quando è verificata la seguente:*

$$h(x) = h(y)$$

Dalla Definizione 2.1 però si deduce come la natura *iniettiva* delle funzioni di hash renda inevitabile la presenza di *collisioni.* Quello a cui punta la maggioranza degli algoritmi di hash è di minimizzare la probabilità con la quale avvengono queste collisioni utilizzando delle funzioni di hash appropriate. Un algoritmo hash ha lo scopo di elaborare un qualsiasi insieme di bit al fine di ricavare un *digest*[1] di dimensione fissa che rappresenti **univocamente** l'insieme di bit elaborato.

[1] L'output di un algoritmo di hashing viene comunemente detto digest.

In *eMule* gli *hash* hanno un ruolo di spessore in ambito della gestione dell'errore. In particolare il protocollo *eMule/eDonkey* definisce una segmentazione dei file condivisi. Ogni file condiviso viene quindi suddiviso dal client secondo due segmentazioni principali ognuna con un proprio sistema di hashing.

2.1.1 Parti e hash MD4

La prima suddivisione introduce il concetto di *parte* (Figura 2.1).

Definizione 2.3 *Si definisce* ***parte*** *un frammento di file della dimensione massima di* $9\,728\,000\ B = 9\,500\ KB \simeq 9.28\ MB$

Un file di dimensione *size* (in B) verrà quindi suddiviso in :

- n parti complete da 9.28 MB, con:

$$n = \left\lfloor \frac{size}{9\,728\,000} \right\rfloor$$

- un'eventuale parte finale di dimensione s (in B), se $s > 0$

$$s = size \ \textbf{\textit{mod}}\ 9\,728\,000$$

Nel caso in cui il file abbia *size* multiplo intero della dimensione massima di una *parte* il file sarà composto da n parti complete.

Il protocollo *eMule/eDonkey* prevede che ogni file nella rete venga indicizzato. Per poter indicizzare i vari file è necessario trovare un metodo sistematico che distingua file diversi. La rete *eMule/eDonkey* assegna a ciascun file un *FileID* in modo da garantire un sistema di identificazione più sicuro di quello basato sul nominativo del file. Ogni peer è infatti libero di rinominare i file condivisi e quindi è possibile che copie dello stesso file risultino con nomi differenti, ma non potranno avere *FileID* diversi in quanto copie dello stesso file. In *eMule*, per il calcolo del *FileID* viene fatto uso dell' *algoritmo MD4*. L'*MD4* è una funzione crittografica di hashing che produce, a partire da una qualsiasi mole di bit, un *digest* di 128 bit (16 byte).

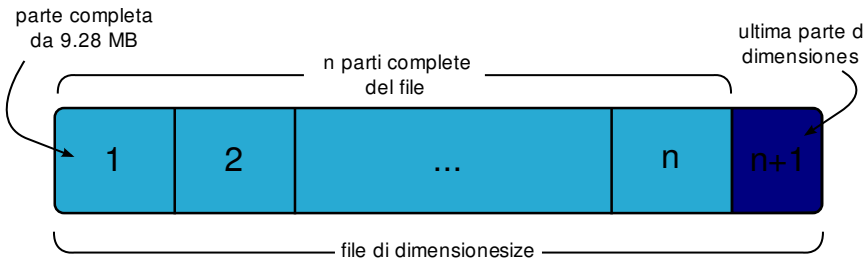


Figura 2.1: Segmentazione in Parti su rete *eMule/eDonkey*.

In paricolare, il *FileID* di un file suddiviso in k parti $(A \dots Z)$ viene calcolato da:

$$FileID = \boldsymbol{md4}(\boldsymbol{md4}(A) \cup \boldsymbol{md4}(B) \cup \cdots \cup \boldsymbol{md4}(Z))$$

Con $\cup$ operatore di *concatenazione* e $\boldsymbol{md4}(\cdot)$ la funzione di hashing.

Praticamente, quando un client mette in condivisione un file nella rete *eMule/eDonkey*, il client provvede a suddividere "virtualmente" il file calcolando per ogni parte l'hash *MD4* corrispondente. Una volta elaborate tutte le parti il client concatena i *digest* ottenuti e riapplica l'algoritmo *MD4* per ottenere il *FileID*. Alla connessione alla rete, il client comunicherà al server di riferimento i file da lui condivisi fornendo in particolar modo i *FileID* di questi. Il server procede poi ad indicizzare questi file sfruttando il loro *FileID*. Nel protocollo il *FileID* è molto importante per gli aspetti riguardanti la ricerca di fonti ed il download di un file, gran parte dei pacchetti definiti dalle specifiche riservano un campo relativo al *FileID* col quale identificare il file.

2.1.2 Chunk e hash SHA1

eMule prevede un'altra suddivisione, definendo il concetto di *chunk*. La segmentazione in *chunk*, assieme alla segmentazione in parti, costituiscono un sistema di segmentazione del file su due livelli. Il file, come visto, è suddiviso in parti; ciascuna delle quali viene a sua volta suddivisa in *chunk*.

Definizione 2.4 *Si definisce* ***chunk*** *un frammento di* ***parte*** *della dimensione massima di* 180 *KB*

Dato che la segmentazione in *chunk* interessa la sola suddivisione di una parte è possibile identificare due possibili casi, determinati dalla dimensione della parte che si sta suddividendo in *chunk*. Come visto in 2.1.1, la dimensione di una parte completa è di 9.28 MB. La segmentazione in chunk, in questo caso, presenta un numero fisso di *chunk* dato da:

$$\#chunk = \left\lceil \frac{9\,500\text{KB}}{180\text{KB}} \right\rceil = 53$$

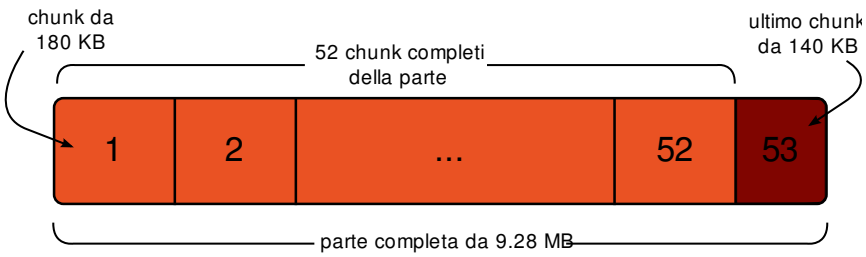


Figura 2.2: Segmentazione in Chunk di una parte completa.

Da Figura 2.2, di questi 53 *chunk* identifichiamo i primi 52 chunk come "completi", ovvero con dimensione di 180 KB. L'ultimo chunk restante è invece caratterizzato da una dimesione di:

$$size = 9\,500 \ \boldsymbol{mod}\ 180 = 140 \text{ KB}$$

Il secondo caso riguarda la suddivisione in *chunk* di una parte di dimensione strettamente inferiore a 9.28 MB. In questo caso il numero di *chunk* di cui è composta la parte sarà in genere inferiore a 53 e la dimensione dell'ultimo chunk non sarà necessariamente di 140 KB.

Il protocollo *eMule/eDonkey* prevede un altro sistema di hash associato alla segmentazione in *chunk*. Su di ogni chunk è infatti calcolato un *hash SHA1*[2]. *SHA1* è una funzione crittografica che produce un *digest* di 160 bit (20 byte) a partire da un input di dimensione arbitraria **inferiore** a $2^{64} - 1$ bit. L'insieme di questi hash permettono un controllo più efficiente degli errori nello scaricamento di un file. Ogni hash permette infatti di controllare eventuali errori avvenuti nello scaricamento di un singolo *chunk*; comportando, al rilevamento di un errore, lo riscaricamento del singolo chunk corrotto, per un massimo di 180 KB. L'utilizzo e lo scambio di questi hash nella rete *eMule/eDonkey* va sotto il nome di *AICH*.

2.2 Seriale contro Parallelo

In 2.1 si è visto come il funzionamento della rete *eMule/eDonkey* si basi sulla segmentazione e sull'*hashing dei file*[3]. È quindi di notevole importanza per un client come *Mulo* calcolare tutti gli hash necessari alla condivisione di un file, nel momento in cui l'utente decide di condividerlo. Le operazioni di calcolo di tutti gli hash di un certo tipo (SHA1 o MD4) per un dato file richiedono un determinato tempo di elaborazione, in gran parte dipendente dalla dimensione del file di cui si sta eseguendo l'*hashing*.

Si prenda, per esempio, un file costituito da n parti complete e si stimi che il tempo di elaborazione richiesto dall'algoritmo MD4 per produrre l'hash di una parte completa (9 500 KB) sia un tempo costante K ($\mathbb{O}(1)$[4]). Il tempo di elaborazione richiesto per il calcolo di tutti gli hash MD4 delle parti sarà dato da:

$$t_{parti} = \sum_{i=1}^{n} K = K \cdot n$$

In aggiunta a questo, bisogna considerare anche il tempo relativo al calcolo del *FileID* (Sezione 2.1.1) per il quale si avrà:

$$t_{FileID} = \beta \cdot n$$

[2]SHA1 è un algoritmo della famiglia Secure Hash Standard.

[3]L'hashing dei file su rete *eMule/eDonkey* consiste nel calcolo di tutti gli hash MD4 e SHA1 relativi ai file.

[4]La notazione matematica $\mathbb{O}(\cdot)$ (O-grande) è utilizzata per descrivere il comportamento asintotico delle funzioni.

L'hashing MD4 dell'intero file portato in esempio impiegherà quindi un tempo di elaborazione totale:

$$t_{MD4} = t_{parti} + t_{FileID} = K \cdot n + \beta \cdot n = \gamma \cdot n \quad (\mathbb{O}(n))$$

Per lo stesso file di n parti complete si stimi ora che il tempo impiegato dall'algoritmo SHA1 per produrre l'hash di un chunk (180 KB) sia anch'esso costante C. Essendo ogni parte completa composta da 53 chunk il file sarà composto da un totale di $53 \cdot n = k$ chunk. Il tempo di elaborazione per il calcolo di tutti gli hash SHA1 di ogni singolo chunk del file sarà quindi:

$$t_{chunk} = \sum_{i=1}^{k} C = C \cdot k = C \cdot 53 \cdot n = B \cdot n$$

In aggiunta a questo si deve considerare il tempo richiesto da *AICH* per la costruzione del proprio *hashset*[5] a partire dagli hash dei chunk appena calcolati. In particolare la creazione dell'*hashset AICH* è divisa in due fasi:

1. a partire dei chunk relativi ad ogni parte viene creato l'hashet AICH relativo alla parte (*partHashset*), ricavando così dal nodo radice dell'hashset l'hash SHA1 della parte (*partHash*).

2. a partire da tutti i *partHash* delle parti viene costituito l'hashset AICH relativo all'intero file (*rootHashset*) dal quale tramite il nodo radice si ricava l'hash SHA1 relativo al file (*rootHash*).

Essendo il tempo complessivo della creazione di un hashset di x nodi foglia dato da:

$$t_{hashset}(x) = \sum_{\ell=0}^{\log_2(x)+1} \alpha \cdot 2^{\ell} = \alpha \cdot (1 + 2 + \cdots + 2^{\log_2(x)+1}) = \alpha \cdot (2 \cdot x + 1) = \beta \cdot x \quad (\mathbb{O}(x))$$

Il tempo totale per la costruzione dell'hashset per il file di n parti complete sarà quindi:

$$t_{hashset} = n \cdot (\beta \cdot 53) + \beta \cdot n = \delta \cdot n \quad (\mathbb{O}(n))$$

Con questo otteniamo il tempo di elaborazione totale impiegato per l'hashing SHA1 del file:

$$t_{SHA1} = t_{chunk} + t_{hashset}$$

Ogni client *eMule/eDonkey* deve quindi attendere il tempo di hashing del file prima di rendere disponibile il file sulla rete. Il metodo più semplice (e più ovvio) per effettuare l'hashing del file da condividere è quello di effettuare una *serializzazione* (Figura 2.3); ovvero effettuare dapprima l'hashing MD4 del file per poi proseguire con l'hashing SHA1 dello stesso (o viceversa). Impiegando un tempo:

$$t_{seriale} = t_{MD4} + t_{SHA1}$$

[5] L'hashset di AICH è un albero binario, i cui nodi contengono un hash SHA1.

In questo modo, di fatto, il file viene letto due volte. La prima quando vengono calcolati gli hash MD4 ed una seconda quando vengono calcolati gli hash SHA1. L'approccio seriale effettua una doppia lettura del file anche se non sarebbe necessario: basterebbe infatti calcolare insieme tutti gli hash SHA1 ed MD4 relativi alla stessa parte per non doverla rileggere. La doppia lettura, quindi, non solo comporta un'inefficienza dal punto di vista logico ma contribuisce anche all'aumento del tempo di elaborazione degli hash.

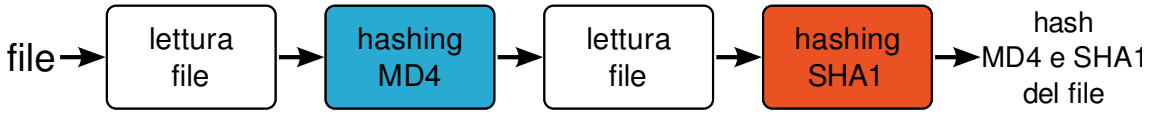


Figura 2.3: Approccio seriale all'hashing.

Nelle formule precedenti va infatti considerato il tempo di accesso al disco ed il tempo di lettura degli input per gli algoritmi di hash; cosa che incide molto in quanto il tempo di accesso alla memoria di massa è di consueto *molto maggiore*[6] rispetto al tempo di elaborazione di un processore.

Si pone un rimedio adottando una *parallelizzazione* delle procedure di hashing. L'approccio parallelo prevede la distribuzione dei compiti da assolvere su due o più *Thread* progettati per lavorare in modo cooperativo. Nel caso specifico dell'hashing del file, si identificano tre funzioni principali: l'hashing MD4, l'hashing SHA1 e, soprattutto, la lettura del file. Il criterio prevede quindi l'ausilio di tre thread $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ rispettivi ai tre compiti principali:

Thread $\mathcal{A}$ Si occuperà di leggere le parti del file sulle quali calcolare gli hash. In particolare una volta letta la parte *i-esima* il thread provvederà ad "avvertire" i thread $\mathcal{B}$ e $\mathcal{C}$ in modo che svolgano il loro lavoro.

Thread $\mathcal{B}$ Avrà il compito di effettuare l'hashing MD4 sulla parte caricata dal thread $\mathcal{A}$ fornendo il relativo hash.

Thread $\mathcal{C}$ Con un compito analogo al thread $\mathcal{B}$, provvederà a calcolare gli hash SHA1 relativi ai chunk della parte caricata dal thread $\mathcal{A}$ ed il corrispondente *partHash*.

Una volta calcolati gli hash delle pari e dei chunk del file, verranno eseguite le operazioni finali su questi: il calcolo del *FileID* e la costruzione dell'*hashset AICH*. La parallelizzazione comporta quindi un miglioramento prestazionale anche su queste ultime fasi. Il miglioramento apportato dall'approccio parallelo consiste poi in una gestione più efficiente degli accessi alla memoria di massa. In ambiente *multiprocessore*, in particolare, si riscontra una velocizzazione nella determinazione degli hash. Il calcolo degli hash SHA1 ed MD4 su una stessa parte verrà infatti effettuato in simultanea dalle *CPU*[7]; riducendo il tempo di

[6] Tempistiche nell'ordine dei millisecondi per l'accesso alle memorie di massa contro le frazioni di nanosecondi dei clock dei processori.

[7] Central Processing Unit - unità di elaborazione centrale.

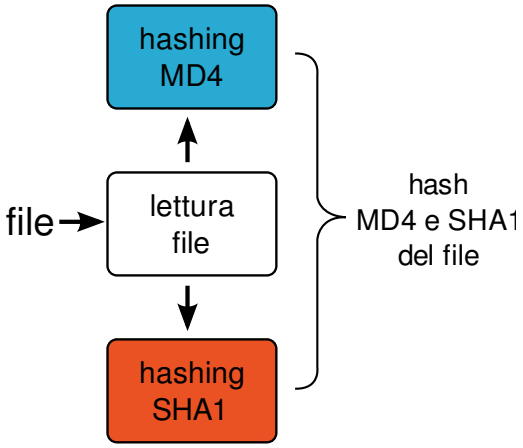


Figura 2.4: Approccio parallelo all'hashing.

elaborazione rispetto all'approccio seriale. Da:

$$seriale \implies T_{hashing\ parte\ i} = T_{i\ MD4} + T_{i\ SHA1}$$

a:

$$parallelo \implies T_{hashing\ parte\ i} = \boldsymbol{max}\{T_{i\ MD4}, T_{i\ SHA1}\}$$

Con $T_{i\ MD4}$ il tempo relativo al calcolo dell'hash MD4 dell' *i-esima* parte e $T_{i\ SHA1}$ il tempo impiegato per il calcolo degli hash SHA1 dei chunk corrispondenti e del relativo *partHash*. L'approccio parallelo, in ambiente multiprocessore, apporta anche un miglioramento nella tempistica delle fasi finali: Da:

$$seriale \implies t_{hashing\ finale} = t_{FileID} + t_{rootHashset}$$

a:

$$parallelo \implies t_{hashing\ finale} = \boldsymbol{max}\{t_{FileID}, t_{rootHashset}\}$$

Il tempo di hashing (MD4 e SHA1) di un file di n parti, utilizzando la parallelizzazione, è dunque:

$$t_{parallelo} = \sum_{i=1}^{n} \boldsymbol{max}\{T_{i\ MD4}, T_{i\ SHA1}\} + \boldsymbol{max}\{T_{FileID}, T_{rootHashset}\}$$

Minore rispetto al tempo richiesto dall'approccio seriale:

$$t_{seriale} = \sum_{i=1}^{n} (T_{i\ MD4} + T_{i\ SHA1}) + t_{FileID} + t_{rootHashset}$$

In particolare il divario tra i tempi di elaborazione degli approcci parallelo e seriale aumenta all'aumentare del numero delle parti di cui è composto il file e quindi con la dimensione di questo.

2.3 Progettazione in Mulo

In *Mulo* un file in condivisione viene rappresentato e gestito attraverso la classe `Shared`. Tale classe viene implementata in *Mulo* con il duplice scopo di mantenere le informazioni sui singoli file condivisi (*path*[8], dimensione, *FileID* e *hashset AICH*) e di mantenere una lista di tutti i contenuti messi in condivisione dal client. La classe offre funzioni per l'aggiunta o la rimozione di file dalla lista di condivisione. Mentre la rimozione di un file dalla condivisione comporta la semplice rimozione di un *oggetto*[9] `Shared` dalla lista, la condivisione di un nuovo file comporta dei procedimenti aggiuntivi; nello specifico:

1. La creazione di un oggetto `Shared` con le informazioni preliminari del file.
2. L'**hashing del file** ed il conseguente salvataggio degli hash nel relativo oggetto `Shared`.
3. L'aggiunta dell'oggetto `Shared` nella lista di condivisione.

La classe `Shared` richiamerà quindi l'esecuzione del sistema di hashing parallelo ogni qual volta verrà aggiunto un file in condivisione, a sua volta il sistema di hashing parallelo provederà ad inizializzare i campi del'oggetto `Shared` relativi agli hash del file, così da permettere l'inserimento del nuovo file nella lista di condivisione. Come visto in Sezione 2.2, il sistema di hashing parallelo si compone di tre thread principali:

`HashManager` thread principale che avvia i due hashing del file e si occupa contemporaneamente di accedere in lettura al file da "hashare" così da fornirlo ai due thread secondari.

`MD4Hasher` thread che calcola tutti gli hash MD4 di cui necessita il file.

`SHA1Hasher` thread che calcola tutti gli hash SHA1 relativi al file.

Un esempio di creazione di un thread con *Mulo* all'interno dell'architettura software presente in *PariPari* è disponibile in Appendice A, per approfondimenti legati ai thread in *PariPari* si rimanda invece a [Ampezzan]. Sebbene l'implementazione dei thread descritti, presi singolarmente, non presenti problematiche, al contrario la gestione simultanea dei tre thread comporta notevoli complicazioni. In particolare bisogna curare l'aspetto cooperativo nel comportamento dei thread: la *sicronizzazione*. Ogni thread deve infatti attendere il proprio turno per svolgere il proprio compito. Non sincronizzando i thread potrebbe accadere che `HashManager` fornisca una nuova parte da "hashare" prima ancora che i thread `MD4Hasher` `SHA1Hasher` abbiano completato l'hashing della parte precedente; o ancora che i thread `MD4Hasher` e `SHA1Hasher` comincino ad operare prima ancora che `HashManager` finisca il passaggio della nuova parte da computare.

[8] Il path, letteralmente "percorso" è un nome che contiene in forma esplicita informazioni sulla posizione del file all'interno del sistema.

[9] Sotto un'ottica orientata agli oggetti, con oggetto si intende una zona di memoria allocata.

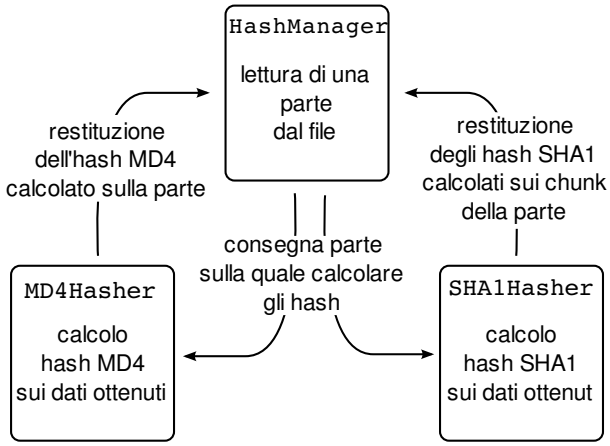


Figura 2.5: `HashManager`, `MD4Hasher` e `SHA1Hasher`: schema delle relazioni.

2.3.1 `MD4Hasher`

Si passa quindi a definire il corpo implementativo del thread `MD4Hasher` il cui compito, precedentemente descritto, può essere esplicitato dal seguente pseudocodice *Java-like*.

Listato 2.1: Pseudocodice per `MD4Hasher`

```
1  thread MD4Hasher{
     byte[] buffer;
     MD4Hash digest;
4    go(){
       while(!mustStop){
         digest = MD4Hash.calculate(buffer);
7      }
     }
   }
```

Dove `buffer` rappresenta la variabile in cui, di volta in volta, il thread `HashManager` va a caricare i dati da elaborare; `digest` rappresenta la variabile in cui `MD4Hasher` salverà l'hash MD4 ottenuto dall'ultima elaborazione eseguita. In `go()` si trova invece la sequenza di istruzione eseguite da `MD4Hasher`. In prima analisi il thread si occupa semplicemente di applicare l'algoritmo MD4 ai *byte* contenuti in `buffer` e di salvare il risultato in `digest`. In *Mulo* viene già implementata una classe dedicata alla gestione ed al calcolo degli hash MD4: `MD4Hash`. La classe implementa l'algorimo MD4 per il calcolo del digest a 16 byte mediante l'apposito metodo `calculate(byte[] contents)`. Inoltre permette l'operazione di confronto tra due oggetti `MD4Hash` per verificare la corrispondenza di due hash MD4, operazione fondamentale per un client *eMule/eDonkey*. Il calcolo dell'hash viene eseguito su `buffer` a ciclo continuo fino a che il thread non viene fermato (`mustStop = true`). Il thread `MD4Hasher` viene utilizzato per il duplice scopo di calcolare i vari hash MD4 delle parti ed l'hash MD4 rappresentante il *FileID*. Il thread principale `HashManager` deve dunque a provvedere di caricare in tempo la nuova parte

da computare in modo che il thread possa, al ciclo successivo, restituire un hash MD4 corretto; ma deve anche fare in modo che, una volta ottenuti gli hash di tutte le parti, la concatenazione di questi venga caricata in `buffer` così da ottenere in `digest` il *FileID* del file.

2.3.2 SHA1Hasher

Leggermente differente risulta invece il discorso per quanto riguarda il corpo del thread `SHA1Hasher`, determinato dal seguente pseudocodice.

Listato 2.2: Pseudocodice per `SHA1Hasher`

```
thread SHA1Hasher{
  byte[] buffer;
  AICHHashset partHashset;
  SHA1Hash digest;
  go(){
    while(!mustStop){
      if(!isEmpty(buffer)){
        digest = calculateSHA1(buffer);

      }
      else{
        digest = partHashset.getRootHash();
      }
    }
  }
}
```

`buffer` e `digest` hanno ruoli analoghi all'implementazione di `MD4Hasher`: in `buffer` si stanziano i bit da elaborare ed in `digest` il risultato ottenuto dall'ultimo calcolo eseguito. Di notevole rilevanza è `partHashset`, sul quale viene mantenuta la struttura dell'hashset AICH necessario al calcolo del *partHash*. In *Mulo* vengono già implementate tutte le classi per la creazione di un hashset AICH (di cui si ha un analisi approfondita in [Muscarella]), in particolare la classe `AICHHashset` che permette la gestione dei nodi dell'hashset ed in particolare di poter accedere e modificare il contenuto dei nodi foglia (**`AICHNode`**`[] getLeaves()`). Una volta settati opportunamente i valori dei nodi foglia dell'hashset la classe mette a dispozione il metodo `SHA1Hash getRootHash()` che effettua la costruzione dell'intero hashset AICH e ne restituisce l'hash del nodo radice. In `go()` invece si possono osservare le operazioni eseguite dal thread `SHA1Hasher`. Al thread sono assegnati due compiti distinti a seconda della fase di hashing in cui si trova il file: nel caso ci si trovi nella prima fase e si sta quindi effettuando l'hashing delle varie parti il thread `HashManager` va a caricare in `buffer` i byte della parte da elaborare (`isEmpty(buffer) = false`) e quindi il thread va ad eseguire l'operazione `digest = calculateSHA1(buffer);`. `calculateSHA1(`**`byte`**`[] buffer)` si occupa di suddividere i vari byte contenuti in `buffer` in chunk e calcolare su questi gli

hash SHA1. Una volta ottenuti gli hash il metodo si occupa di costruire l'hashset AICH `partHashset` in `SHA1Hasher` fornendo ai nodi foglia gli hash appena calcolati e, completata la costruzione dell'hashset, viene ritornato l'hash SHA1 relativo al nodo radice.

Listato 2.3: Pseudocodice per il metodo `calculateSHA1` di `SHA1Hasher`

```
SHA1Hash calculateSHA1(byte[] buffer){
  SHA1Hash[] chunksHash;
  i=0;
  for(chunk in buffer){
    chunkHash[i] = SHA1Hash.calculate(chunk);
    i++;
  }
  i=0;
  for(AICHNode leaf in SHA1Hasher.partHashset.getLeaves()){
    leaf.hash = chunkHash[i];
    i++;
  }
  return SHA1Hasher.partHashset.getRootHash();
}
```

Alle fasi finali dell'hashing del file, al thread `SHA1Hasher` viene invece affidato il compito di calcolare il *rootHash* dell'intero file. In questo caso il thread principale `HashManager` non carica alcun byte in `buffer` (`isEmpty(buffer) = true`). `SHA1Hasher` procede quindi a costruire direttamente l'hashset AICH sfruttando come hash SHA1 dei nodi foglia i *partHash* precedentemente calcolati, salvando al termine il *rootHash* in `digest`. Anche in `SHA1Hasher` è fondamentale il tempismo del thread `HashManager`, che deve preparare i dati da elaborare in modo che `SHA1Hasher` restituisca gli hash SHA1 corretti.

2.3.3 `HashManager`

Listato 2.4: Pseudocodice per `HashManager`

```
main thread HashManager{
  MD4Hasher md4;
  SHA1Hasher sha1;
  byte[] buffer;
  calculateHash(Shared file){
    byte[] concatenatedMD4;
    md4.start();
    sha1.start();
    for(part in file.parts){
      buffer = file.read(part);
      AICHHashset partHashset = file.fileHashset.addPartHashset(part.index);
      md4.buffer = buffer;
      sha1.buffer = buffer;
      sha1.partHashset = partHashset;
      saveHash(file,part.index,sha1.digest,md4.digest);
```

```
16          concatenatedMD4.put(md4.digest);
          }
          md4.buffer = concatenatedMD4;
19        sha1.buffer = null;
          sha1.partHashset = file.fileHashset;
          file.md4Hash = md4.digest;
22        file.sha1Hash = sha1.digest;
        }
    }
```

`HashManager` è il thread principale dal quale viene avviato il funzionamento del sistema di Hashing Parallelo. La sua funzione consiste nel comunicare i dati da computare ai due thread figli (`MD4Hasher` e `SHA1Hasher`) e nel salvare gli hash ottenuti all'inteno della struttura `Shared`, rappresentante il file da condividere e tutte le informazioni necessarie alla condivisione (in primis gli hash). Un riassunto del suo funzionamento è illustrato in Listato 2.5. L'avvio all'hashing è eseguito richiamando il metodo `calculateHash(`**`Shared`** `file`), comunicando il file (rappresentato da un *oggetto* `Shared`) su cui di desidera effettuare l'hashing. Il metodo procede quindi con il lanciare l'esecuzione dei due thread subalterni (Listato 2.5:7-8) per poi subito dopo cominciare a caricare i dati da elaborare per effettuare la prima fase dell'hashing del file (Listato 2.5:9). In particolare per ogni parte di cui il file è composto vengono eseguite diverse operazioni: per prima cosa viene caricata in `buffer` la prossima parte su cui effettuare l'hashing, inoltre viene anche preparata la struttura hashset da utilizzare per ottenere il *partHash*. `Shared` al suo interno contiene varie informazioni, tra cui l'hashset completo del file (*fileHashset*) comprensivo del *rootHashset* e dei vari *partHashset*.

Con l'istruzione in (Listato 2.5:11) viene salvato in `partHashset` un riferimento ad una parte dell'hashset completo relativo al file, nel particolare la porzione relativa al *partHashset* della parte di indice `part.index`. In questo modo è possibile modificare i nodi dell'hashset completo operando su `partHashset`.

Dopodiché vengono aggiornati i riferimenti a `buffer` dei due thread e il riferimento al `partHashset` del thread `SHA1Hasher` in modo che possano svolgere le loro elaborazioni e ottenere così gli hash della parte. Infine (Listato 2.5:15-16), vengono salvati nella struttura `Shared` gli hash ottenuti dall'hashing della parte. In particolare viene mantenuto un buffer (`concatenatedMD4`) su cui inserire uno dopo l'altro i vari hash MD4 delle parti calcolate.

Da (Listato 2.5:18), `HashManager` procede col caricare i dati da hashare per la fase finale dell'hashing. Nello specifico si occupa, per `MD4Hasher`, di caricare in buffer la concatenazione degli hash MD4 delle parti (contenuti in `concatenatedMD4`). Riguardo a `SHA1Hasher`, `HashManager` non comunica alcun dato da elaborare (`sha1.buffer = null`), ma si occupa di fornire il *fileHashset* così che `SHA1Hasher` possa calcolare il *rootHash*. Per ultima cosa, quindi, `HashManager` salva in `Shared` rispettivamente *FileID* (Listato 2.5:21) e *rootHash* (Listato 2.5:22) del file.

2.3.4 Sincronizzazione dei thread

Nelle sezioni 2.3.1, 2.3.2 e 2.3.3 è stata presentata una prima implementazione dei thread che compongono il sistema di hashing parallelo. Analizzando attentamente il codice, è possibile osservare come l'implementazione soffra di evidenti problemi in presenza di *multithreading*. In particolare nei thread non è stato implementato un sistema controllato di accesso alla principale "risorsa critica": `buffer`. Per il funzionamento dell'intero sistema è infatti necessario che il thread principale `HashManager` acceda alla risorsa (e ne modifichi quindi il contenuto) solo quando i thread secondari di hashing (`MD4Hasher` e `SHA1Hasher`) finiscano **entrambi** la computazione su questa. Allo stesso modo, i thread secondari non possono cominciare le operazioni sulla risorsa finché il thread principale non ha terminato il suo compito con essa. Per risolvere problemi di questo tipo si introduce uno strumento per la gestione dell'accesso a eventuali "risorse critiche": i *semafori*[10].

Un *semaforo* è un *tipo di dato astratto* il cui scopo è quello di sincronizzare, in ambiente *multithreading*, l'accesso ad una risorsa condivisa tra thread. Un semaforo (`Semaphore sem`) prevede due operazioni principali:

- `sem.wait()` che permette al thread chiamante di verificare se il semaforo dichiara la risorsa relativa come occupata da altri thread, attendendo che il semaforo la dichiari libera per poi accedervi ed occuparla lui stesso.

- `sem.notify()` che permette di segnalare, ad ogni thread in attesa sul semaforo relativo alla risorsa, che tale risorsa è stata liberata dal thread chiamante.

Con l'introduzione dei semafori è possibile effettuare una riorganizzazione del codice dei thread così da sincronizzare l'accesso a `buffer`. In particolare i thread secondari effettueranno un `wait()` per la risorsa `buffer` in modo che ne attendano il caricamento da parte del thread principale. Il thread principale, una volta caricato il buffer, va ad effettuare dei `notify()` per "risvegliare" i due thread in attesa, così che possano svolgere il loro lavoro. A sua volta `HashManager` effettua un `wait()` per attendere il completamento delle elaborazioni dei due thread secondari (quindi un `wait()` sui `digest` di questi) e i due thread effettuano un `notify()` una volta che gli hash vengono salvati sui `digest`, così da permettere al thread principale di caricare nuovi dati.

Osservando Figura 2.6 è possibile notare come la sincronizzazione comporti un necessario allungamento dei tempi di esecuzione: i thread devono infatti sospendersi alle chiamate di `wait()` nel caso le risorse dei relativi semafori non siano libere allungando in questo modo il tempo necessario a completare il loro compito. In questo caso è tuttavia possibile migliorare i tempi di esecuzione modificando opportunamente l'implementazione di `HashManager`. `HashManager`, una volta caricati i dati da elaborare va a mettersi in attesa fino a quando i due thread secondari non finiscolo il loro calcolo. Solo una volta che questi thread hanno restituito i loro `digest`, `HashManager` prosegue col caricare i successivi dati da elaborare. Nel periodo che va dal caricamento dei dati correnti al momento

[10]Per il discorso qui tenuto si ci si riferisce in particolar modo alla tipologia dei **semafori binari**

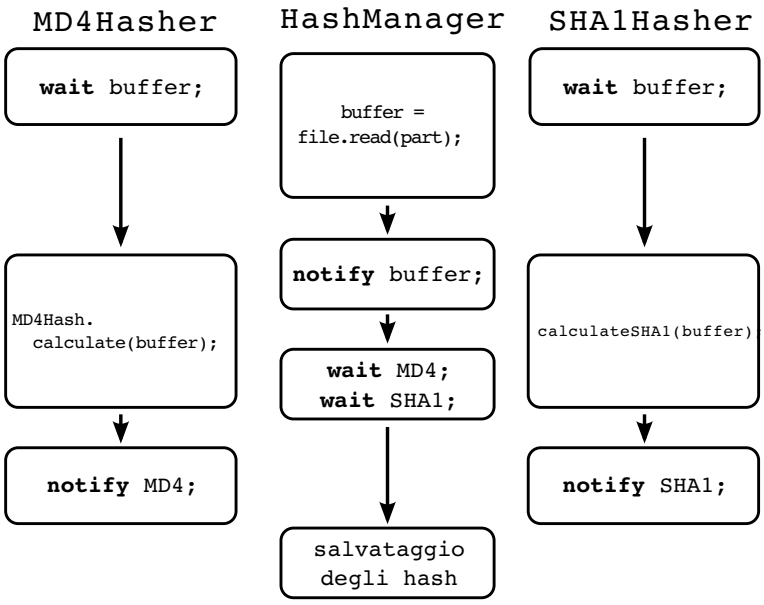


Figura 2.6: Sincronizzazione del sistema di Hashing Parallelo.

i cui i thread secondari notificano il risultato della computazione, il thread `HashManager` rimane in attesa quando invece potrebbe iniziare il caricamento dei prossimi dati da elaborare. La causa pincipale di questa inefficienza è derivato dall'utilizzo di un singolo `buffer` sul quale caricare ed elaborare i dati. Con l'utilizzo di due distinte variabili in cui caricare i dati (`bufferA` e `bufferB`) è possibile far elaborare dai thread secondari i dati precedentemente caricati nel `bufferA` mentre nel frattempo caricare i prossimi dati da elaborare nel `bufferB` per poi, una volta ottenuti gli hash, scambiare i contenuti dei due buffer.

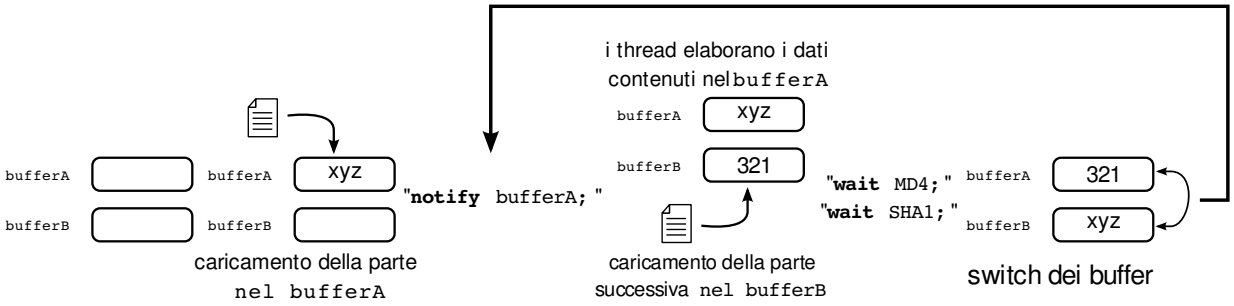


Figura 2.7: Funzionamento "a doppio buffer" di `HashManager`.

Listato 2.5: Pseudocodice Sincronizzato per `HashManager`

```
main thread HashManager{
  MD4Hasher md4;
  SHA1Hasher sha1;
  byte[] bufferA;
  byte[] bufferB;
  Semaphore semMD4;
  Semaphore semSHA1;
  Semaphore semA;
  Semaphore semB;
  calculateHash(Shared file){
    byte[] concatenatedMD4;
    bufferA = file.read(file.parts[0]);
    AICHHashset partHashset = file.fileHashset.addPartHashset(0);
    md4.buffer = bufferA;
    sha1.buffer = bufferA;
    sha1.partHashset = partHashset;
    md4.start();
    sha1.start();
    semA.notify();
    for(partIndex=1 to file.parts.length){
      bufferB = file.read(file.parts[partIndex]);
      AICHHashset partHashset = file.fileHashset.addPartHashset(part.index);
      semMD4.wait();
      semSHA1.wait();
      saveHash(file,part.index,sha1.digest,md4.digest);
      concatenatedMD4.put(md4.digest);
      switch(bufferA,bufferB);
      md4.buffer = bufferA;
      sha1.buffer = bufferA;
      sha1.partHashset = partHashset;
      semA.notify();
    }
    semMD4.wait();
    semSHA1.wait();
    saveHash(file,part.index,sha1.digest,md4.digest);
    concatenatedMD4.put(md4.digest);
    md4.buffer = concatenatedMD4;
    sha1.buffer = null;
    sha1.partHashset = file.fileHashset;
    semA.notify();
    semMD4.wait();
    semSHA1.wait();
    file.md4Hash = md4.digest;
    file.sha1Hash = sha1.digest;
  }
}
```

2.4 Implementazione in Mulo

Nella sezione precedente si sono descritti i passi che hanno portato alla progettazione del sistema di hashing parallelo di *Mulo*. Sono stati definiti i thread che costituiscono il sistema, le loro funzioni e la loro logica di funzionamento. Il passo finale da compiere è il tradurre tutti i concetti e le meccaniche descritte in fase di progettazione, in codice sorgente *Java* definendo nuove classi o modificando quelle già esistenti internamente ai sorgenti di *Mulo*. In questo passaggio le caratteristiche di un linguaggio di programmazione come *Java* semplifica ulteriormente tale procedimento. In particolare in *Java*, ogni oggetto definisce automaticamente le funzioni di `wait()` e `notify()` con le quali è possibile implementare molto semplicemente un *semaforo*. Per una visione completa ed una spiegazione accurata del codice sorgente riguardante il sistema di *hashing parallelo* si rimanda a Appendice B.1.

Una piccola nota va fatta in merito alla classe `SharedAdder.java`. Tale classe si occupa semplicemente di eseguire l'intero procedimento di condivisione di un file lanciando un thread apposito. In questo modo si evita che il flusso principale del programma *Mulo* si interrompa per affettuare tutti i calcoli di condivisione, così che l'utente sia libero di effettuare altre operazioni nel frattempo.

Un altro particolare implementativo degno di nota è la retrocompatibilità con l'approccio Seriale. In particolare al momento in cui il codice lancia l'hashing del file da condividere, viene effettuato un controllo sulle risorse della macchina (in particolare memoria e processore). Le caratteristiche degli approcci parallelo e seriale (visti in Sezione 2.2) riguardano in modo specifico queste due risorse, favorendo il parallelo nel caso in cui la memoria disponibile sia molto limitata (la gestione della memoria nell'hashing parallelo è migliore) e soprattutto in ambiente multiprocessore (in cui il tempo di elaborazione dell'hashing parallelo risulta minore rispetto al tempo impiegato da un hashing seriale).

Listato 2.6: Estratto di `realAdd(...)` in `Shared.java`

```
187       if (Runtime.getRuntime().availableProcessors() == 1 && Runtime.getRuntime().freeMemory
              () >= file.length()) {
188         // parallelism is futile, hashing with the standard methods
202       }
203       else {
204         // parallel hashing
205         newFile = new Shared(file.getAbsolutePath(), size, file.lastModified());
206         HashManager.calculateHash(newFile);
207         filesList.put(newFile.path, newFile);
208         hashesList.put(newFile.md4Hash, newFile);
209         if (Server.weAreConnected()) {
210           Server.currentServer.filesToBeOffered.add(newFile);
211         }
212       }
```

2.4.1 Testing

Con l'avanzamento dell'implementazione del codice sono stati effettuati vari test per verificare la giusta sincronizzazione dei thread e il corretto calcolo degli hash. In un primo tempo si è cercato di progettare dei *test JUnit*[11] per verificare che i thread seguissero la giusta sincronizzazione, tuttavia data la grande difficoltà e soprattutto la semplice impossibilità di poter testare con *JUnit* aspetti legati ai thread in esecuzione si è dovuto ripiegare per un *debug* puro del codice. Analizzando quindi l'esecuzione del codice tramite opportune stampe di debug è stato possibile testare e correggere il funzionamento dei thread. Riguardo la verifica della correttezza degli hash ottenuti si è invece sfruttato un apposito *test JUnit* (Listato 2.7) in cui, creando un file di dimensione voluta (e relativi *FileID* e *rootHash* noti), si procedeva con la condivisione di questo verificando che gli hash calcolati corrispondessero.

Listato 2.7: Estratto di `ParallelHashTest.java`

```
45    /**
       * Try to calculate MD4 and SHA1 Hashes of a File.
       */
48    @Test
      public void testHashComputing() throws IOException {
        this.file = Utilities.createSizedFile(1, 1);
51      Shared.add(this.file.getAbsolutePath(), false);
        Shared sharedFile = Shared.filesList.get(this.file.getAbsolutePath());
        byte[] expectedMD4Hash = new byte[] {
54          (byte)0x4C ,(byte)0x4D ,(byte)0xE8 ,(byte)0x58 ,
            (byte)0x4A ,(byte)0xDB ,(byte)0x91 ,(byte)0x23 ,
            (byte)0xB6 ,(byte)0x08 ,(byte)0xC6 ,(byte)0xB5 ,
57          (byte)0xF6 ,(byte)0x21 ,(byte)0x50 ,(byte)0xAD
        };
        for ( int i = 0 ; i < MD4Hash.SIZE ; i++ ) {
60        assertEquals("Wrong byte in the file MD4hash in the position " + i + ", ",
              expectedMD4Hash[i], sharedFile.md4Hash.bytes[i]);
        }
        SHA1Hash expectedSHA1Hash = new SHA1Hash("BWITYCMRQJPLKVWRY7HIPUDEI6MQXSOT");
63      for ( int i = 0 ; i < MD4Hash.SIZE ; i++ ) {
          assertEquals("Wrong byte in the file SHA1hash in the position " + i + ", ",
              expectedSHA1Hash.bytes[i], sharedFile.sha1Hash.bytes[i]);
        }
66    }
```

[11] JUnit è un unit test framework per il linguaggio di programmazione Java.

2.4.2 Risultati e Conclusioni

La volontà di implementare un sistema hashing parallelo per *Mulo* è nata con lo scopo di ottimizzare le varie operazioni necessarie alla condivisione di un file nella rete *eMule/eDonkey*. Attraverso un'ottimizzazione delle procedure di condivisione è possibile condividere i file in maniera più rapida ed efficiente favorendo l'aumento dei contenuti nella rete. I miglioramenti, per quanto positivi, si rilevano però minimi nella condivisione di file di dimensioni medio-piccole, in questi frangenti l'hashing parallelo risulta, a volte, addirittura meno efficace rispetto all'approccio seriale (Tabella 2.1). La sincronizzazione dei thread

Tipo hashing	Tempo di hashing per un file di varie dimensioni [ms]		
	360 KB	540 KB	1080 KB
Seriale	1 438	2 140	4 156
Parallelo	1 484	2 275	4 024

Tabella 2.1: Tempistiche di hashing per un File di piccole dimensioni.

presenta un grosso freno alle reali velocità di elaborazione, che però è necessario per il funzionamento del sistema. I veri benefici sono apprezzabili effettuando la condivisione di grandi file (Tabella 2.2). In questi casi (soprattutto per file sopra il GB), l'hashing seriale porta ad un grande consumo di memoria dovendo caricare in memoria l'intero file al fine di operare. L'hashing parallelo, invece, con file molto grandi procede a caricare in memoria solo una parte del file alla volta, operando senza grossi sprechi.

Tipo hashing	Tempo di hashing per un file di varie dimensioni [ms]		
	700 MB	1.8 GB	3 GB
Seriale	64 572	131 956	222 043
Parallelo	60 911	78 440	89 588

Tabella 2.2: Tempistiche di hashing per un File di grandi dimensioni.

Capitolo 3

Pacchetti Compressi

Nel seguente capitolo vengono introdotte nozioni sul tema della *compressione dei dati*; specificando gli scopi e gli usi per cui un algoritmo di compressione viene usato in una rete come *eMule/eDonkey*. Viene infine fornita un'analisi generale dei pacchetti compressi utilizzati da *eMule* e del loro funzionamento, per poi discuterne l'implementazione in *Mulo*.

3.1 Compressione dei Dati

Con il termine "compressione dei dati" (*data compression*) viene indicato il processo attraverso il quale un'informazione, rappresentabile in forma digitale mediante un certo insieme di bit $\mathcal{A}$, viene **codificata** al fine di poterla rappresentare in forma digitale tramite un insieme $\mathcal{B}$ di bit di *cardinalità* strettamente inferiore ad $\mathcal{A}$. L'informazione risultante, necessitando di meno bit per essere rappresentata, permette di essere salvata occupando meno spazio o di essere inviata tramite un sistema di trasmissione dati occupando meno banda. La compressione in queste applicazioni comporta quindi notevoli vantaggi, permettendo il salvataggio di un maggior numero di dati a parità di spazio occupato e, a parità di tempo di trasmissione, permettendo di spedire una quantità di dati compressi superiore alla relativa quantità spedita "in chiaro" senza compressione. Queste osservazioni aiutano a comprendere quanto la compressione dei dati può essere importante per trasmettere grosse moli di informazioni.

In ogni processo di compressione dei dati il calcolo dei dati compressi avviene mediante l'utilizzo di un apposito *algoritmo di compressione*.

Definizione 3.1 *Sia $\mathcal{I}$ un informazione e sia $\mathbb{F}(\mathcal{I})$ la sua rappresentazione in forma digitale. Si definisce* ***algoritmo di compressione*** *una serie di operazioni atte a costruire una rappresentazione digitale $\mathbb{H}(\cdot)$ tale che:*

$$|\mathbb{H}(\mathcal{I})|^1 < |\mathbb{F}(\mathcal{I})|$$

[1] Con $|\cdot|$ si intende l'operatore di cardinalità.

La sola compressione dei dati non fornisce però alcuna informazione utile senza la definizione di un relativo processo di *decompressione*. Ad ogni algoritmo di compressione deve infatti corrispondere un relativo *algoritmo di decompressione* che a partire dai dati compressi restituisca i dati originali.

Definizione 3.2 *Sia $\mathcal{I}$ un informazione e sia $\mathbb{F}(\mathcal{I})$ la sua rappresentazione in forma digitale. Sia dato un algoritmo di compressione che definisce una propria rappresentazione digitale $\mathbb{H}(\cdot)$ e sia $\mathbb{H}(\mathcal{I})$ la rappresentazione compressa dell'informazione.*

Si definisce ***algoritmo di decompressione*** *una serie di operazioni atte a calcolare $\mathbb{F}(\mathcal{I})$ a partire da $\mathbb{H}(\mathcal{I})$.*

$$\mathbb{F}\big[\mathbb{H}^{-1}[\mathbb{H}(\mathcal{I})]\big]$$

Un algoritmo di compressione ed il suo relativo algoritmo di decompressione vanno a formare un *sistema di compressione dei dati*. Ogni sistema di compressione viene distinto dal tipo di rappresentazione $\mathbb{H}(\cdot)$ che viene implementata dagli algoritmi. Dal tipo di rappresentazione $\mathbb{H}(\cdot)$ utilizzato dal sistema di compressione deriva la qualità della compressione: indici come il *fattore di compressione*[2] vengono spesso utilizzati per capire la bontà di un sistema di compressione. $\mathbb{H}(\cdot)$ caratterizza fortemente il relativo sistema di compressione, a tal punto da determinare due tipologie di sistemi di compressione dei dati, derivate dalle diverse caratteristiche delle rappresentazioni usate:

- *lossy* o con perdita di informazione.
- *lossless*, senza perdita di informazione.

3.1.1 Compressione *lossy*

La compressione *lossy* consiste nell'utilizzo di un sistema di compressione di tipo *lossy*.

Definizione 3.3 *Si definisce sistema di compressione* ***lossy*** *un sistema di compressione dei dati in cui $\mathbb{H}(\cdot)$ non è perfettamente invertibile, ovvero:*

$$\mathbb{H}^{-1}[\mathbb{H}(\mathcal{I})] \simeq \mathcal{I}$$

In altre parole, in un sistema di compressione *lossy* non esiste nessun algoritmo che dai bit dell'informazione compressa possa ricostituire l'informazione originale. Si osserva dunque che il processo di decompressione su un'informazione elaborata con un sistema *lossy*, non potendo ricostruire perfettamente l'informazione compressa, produce un'informazione parziale. Il che comporta un **perdita di informazione**.

In particolare le compressioni *lossy* tendono ad eliminare le informazioni marginali o che possono essere ricavate mantenendo solamente quelle necessarie; in questo modo l'informazione perde precisione a discapitò però di un notevole risparmio riguardo lo spazio di immagazzinamento.

[2] Il fattore di compressione viene calcolato come il rapporto $\frac{\text{dimensione originale}}{\text{dimensione compressa}}$.

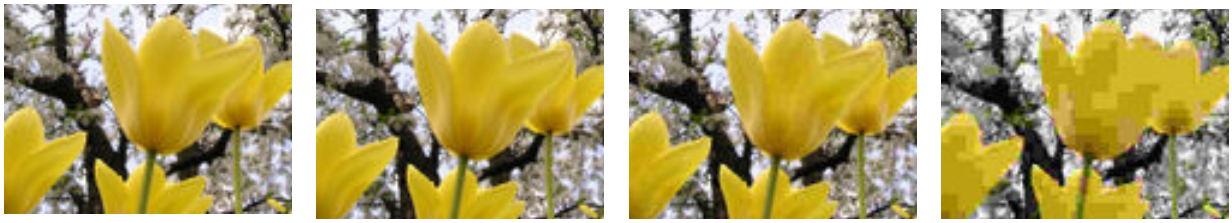


Figura 3.1: Esempio di compressione *lossy* nel formato *JPEG*.

Questo tipo di compressione è utilizzato per scopi nei quali non è necessario ricostruire perfettamente l'informazione originale e la perdita di informazione non compromette la comprensione dell'informazione compressa. Un esempio noto di compressione *lossy* è dato dalla compressione delle immagini *JPEG*[3] nella quale l'alto fattore di compressione è a discapito di una perdita di qualità dell'immagine (Figura 3.1). Altri ambiti in cui la compressione *lossy* viene largamente impiegata sono la compressione audio con il formato *MP3*[4] e la compressione video con il noto formato *XviD*.

3.1.2 Compressione *lossless*

Si parla di compressione *lossless* indicando l'utilizzo di un sistema di compressione dei dati di tipo *lossless*.

Definizione 3.4 *Si definisce sistema di compressione* ***lossless*** *un sistema di compressione dei dati in cui* $\mathbb{H}(\cdot)$ *è perfettamente invertibile, ovvero:*

$$\mathbb{H}^{-1}[\mathbb{H}(\mathcal{I})] = \mathcal{I}$$

Ad ogni algoritmo di compressione è quindi associato un ben definito algoritmo di decompressione tramite il quale è possibile risalire perfettamente all'informazione che è stata compressa. La compressione *lossless*, rispetto alla compressione *lossy*, mira a rimuovere le ridondanze legate ai bit con cui è rappresentata l'informazione anziché modificare l'informazione stessa. Dunque un sistema *lossless* definisce una **nuova codifica** per la rappresentazione dell'informazione in modo che occupi meno spazio per essere immagazzinata.

Il sistema, non alterando l'informazione, è capace di comprimere l'informazione solo entro certi limiti al fine di evitare un'alterazione dell'informazione. Tale limite viene definito nella teoria dell'informazione dal primo teorema di Shannon (*teorema della codifica di sorgente*) secondo il quale:

[3] Joint Photographic Experts Group.
[4] Moving Picture Expert Group-1/2 Audio Layer 3.

Teorema 1 (Codifica di Sorgente) *Sia S una sorgente di informazione avente un'alfabeto $\mathcal{A}$ di simboli. Tale sorgente produca parole di codice X, ognuna caratterizzata da una propria entropia*

$$H(X) = -\sum_{x \in X} p(x) \log_2 \frac{1}{p(x)}$$

Con $p(x)$ la probabilità che la sorgente invii il simbolo x dell'alfabeto $\mathcal{A}$.

Sia $f(\cdot)$ una funzione di codifica che mappa una generica parola di codice X della sorgente in una parola di codice Y composta da simboli di un alfabeto $\mathcal{B}$ di n simboli.

Una parola $Y = f(X)$ non causa perdite sull'informazione contenuta in X se rispetta la seguente:

$$L_y \geq \frac{H(X)}{\log_2 n}$$

Con L_y la lunghezza della parola Y.

In pratica, un'informazione può essere rappresentata, senza perdite, con almeno un numero di bit direttamente proporzionale all'entropia dell'informazione. Nello specifico, l'entropia fornisce un indice sul rapporto tra la quantità di informazione contenuta in un messaggio e la lunghezza del messaggio stesso e quindi il numero minimo di bit necessari dipende dalla "densità di informazione" contenuta nel messaggio che si vuole codificare. Essendo quindi l'algoritmo di compressione di un sistema *lossless* un tipo di *funzione di codifica*, anche la compressione *lossless* è soggetta ai limiti imposti dal Teorema 1; limitando molto i valori del fattore di compressione rispetto alle compressioni di tipo *lossy*.

Esempi di applicazioni legate a compressioni di tipo *lossless* dei dati si trovano in tutti quei programmi di compressione di file generici quali *Winzip*[5], *WinRar*[6], 7-*zip*[7] e molti altri. Anche in ambito della compressione delle immagini vengono usate molte compressioni *lossless*: è il caso di formati molto conosciuti come *GIF*[8] o *PNG*[9]; quest'ultimo utilizzato soprattutto in ambiente web. Moltri altri formati prevedono l'utilizzo di compressioni *lossless*; di questi in particolare si hanno *FLAC*[10], molto usato in ambito audio, e *H.264*[11], nel settore video.

3.2 Compressione in *eMule*

Nell'estensione *eMule* del protocollo *eDonkey* l'utilizzo della compressione applicata ai dati trasmessi permette una riduzione della quantità di byte da spedire. In questo modo è possibile inviare un maggior numero di dati a parità di banda occupata. A grandi

[5] http://www.winzip.com/
[6] http://www.winrar.it/
[7] http://www.7-zip.org/
[8] Graphics Interchange Format.
[9] Portable Network Graphics.
[10] Free Lossless Audio Codec.
[11] Denominato MPEG-4 Advanced Video Coding.

linee il protocollo definisce vari tipi di pacchetti contenti al loro interno dati elaborati mediante un sistema di compressione dei dati *lossless*. È evidente quindi che il protocollo di compressione prevede che i vari client si occupino di comprimere i dati da inviare e di estrarre i dati compressi ricevuti.

In trasmissione un client *eMule* che supporti la compressione procede comprimendo inizialmente i dati da inviare per poi inserire i bit ottenuti dalla compressione negli appositi pacchetti definiti dal protocollo *eMule*. A questo punto il client effettua un controllo sulla dimensione del pacchetto originale e del relativo pacchetto compresso in modo da trasmettere solamente il pacchetto con dimensione minore.

Il sistema di compressione utilizzato da *eMule* si compone degli algoritmi *DEFLATE*[12] e *INFLATE* rispettivamente per compressione e decompressione dei dati. Il supporto alla compressione di un dato peer della rete *eMule/eDonkey* viene comunicato dal peer stesso attraverso l'invio del proprio **tag MiscOptions1**.

3.2.1 Il Tag MiscOptions1

I *tag* sono delle particolari strutture che servono a mantenere informazioni aggiuntive di vario genere che il pacchetto a cui si riferiscono non può fornire. I *tag* sono utilizzati sia in pacchetti per comunicazioni tra peer sia in comunicazioni tra peer e server e ogni pacchetto può mantenere al suo interno un numero variabile di *tag* associati.

Campo	Size [B]	Descrizione
`Type`	1	Identifica il **tipo** di informazione contenuta nel tag: **0x03** per un `Integer`, **0x02** per un tipo `String`, **0x04** per un `Float`, e altro ancora.
`Name`	variabile	Indica il nome del tag. Il nome identifica il signignificato dell'informazione. Può contenere un identificativo da 2 byte o una stringa di lunghezza variabile.
`Value`	variabile	Contiene il valore del tag. La dimensione dipende dal `Type`: 4 byte per `Integer` e `Float` o una lunghezza variabile per `String`.
`Special`	1	Flag che identifica i tag speciali.

Tabella 3.1: Struttura di un *Tag*

In particolare il *tag MiscOptions1* è un particolare tag di tipo `Integer` contenente varie informazioni riguardanti le funzionalità e i protocolli supportati da un determinato peer. Il tag MiscOptions2 viene inviato dal peer all'interno dal pacchetto `HelloRequest` il cui invio è richiesto ogni qual volta il peer deve contattare il server per l'handshake iniziale con la rete o quando il peer si mette in contatto con altri peer. Il tag contiene informazioni relative al peer che lo ha inviato. I 32 bit dell'`Integer` contenuto nel tag hanno significati

[12]http://tools.ietf.org/html/rfc1951

distinti e ognuno descrive differenti caratteristiche relative al peer associato (Tabella 3.2). Di questi bit, i bit dalla posizione 23 alla 20 identificano la versione della compressione

Posizione bit	Significato
31-29	`AICH version`
28	`Unicode`
27-24	`UDP version`
23-20	**`Data Compression version`**
19-16	`Source Ident`
15-12	`Source Exchange`
11-8	`Ext. requests`
7-4	`Comments`
3	`Peer Chache supported`
2	`No 'View Shared Files' supported`
1	`MultiPacket`
0	`Preview`

Tabella 3.2: Analisi del tag MiscOpt1

dei dati supportata dal peer. Nel caso in cui questi bit siano tutti a zero, si considera il peer come non abilitato al supporto dei pacchetti compressi e non può quindi inviarne o riceverne.

Un peer che supporti la compressione può incontrare, nelle varie comunicazioni con gli altri nodi della rete, due principali tipologie di pacchetti che utilizzano la compressione dei dati: i pacchetti del protocollo compresso `0xD4` e i pacchetti `FILE_DATA_COMPRESSED` (`0x40`) dell'estensione *eMule* (`0xC5`).

3.2.2 I pacchetti `0xD4`

I pacchetti *eMule/eDonkey* identificati da un campo `Protocol` (Tabella 3.1) con valore `0xD4` vengono riconosciuti dai peer della rete come pacchetti *protocollo compresso* (d'ora in poi semplicemente pacchetti compressi). La struttura tipica di un pacchetto compresso è mostrata in Tabella 3.3. Un client *eMule* che supporti la compressione, al momento di inviare un pacchetto verifica se il peer ricevente supporta la compressione e se il pacchetto può essere compresso. Nel caso il client supporti la compressione e questa comporti dei vantaggi, il client costruisce un pacchetto compresso da inviare in sostituzione del pacchetto originario. Il client comprime i dati contenuti nel pacchetto (header escluso) e li carica nel campo `Data` del pacchetto compresso. Dalla dimensione dei dati compressi ricava poi il valore da inserire nel campo `Size`. Infine setta il campo `Type` identificando il tipo di pacchetto contenuto nel pacchetto compresso e marchia il pacchetto con campo `Protocol` pari a `0xD4`. Ottenuto il pacchetto compresso il client ne effettua dunque l'invio. In ricezione invece, un qualsiasi client *eMule* che supporti la compressione elabora

Campo	Size [B]	Valore
`Protocol`	1	**`0xD4`**.
`Size`	4	Contiene la lunghezza totale del pacchetto compresso escludendone i primi 6 byte corrispondenti all'header.
`Type`	1	Contiene il tipo di pacchetto.
`Data`	variabile	Contiene i dati compressi del pacchetto.

Tabella 3.3: Struttura di un pacchetto TCP compresso

tutti i pacchetti compressi ricevuti nella stessa maniera. Viene prima letto il campo `Size` così da conoscere la lunghezza dei dati compressi. Dopodiché il client tenta di estrarre i dati originali utilizzando un apposito algoritmo di decompressione dati (*INFLATE*). Se la decompressione viene eseguita correttamente il client ricrea al volo il pacchetto originario. Nello specifico viene creato un nuovo pacchetto nel cui corpo vengono caricati i dati estratti, la dimensione di questi dati viene poi inserita nel campo `Size` del pacchetto, mentre il valore del campo `Type` viene copiato dal pacchetto compresso. Infine viene settato il campo `Protocol`. Per questa ultima operazione il client deriva il valore del campo dal valore del campo `Type`. Se per esempio viene ricevuto un pacchetto compresso con campo `Type` di valore `0x15` (pacchetto OfferFiles) il client setterà il campo `Protocol` a un valore `0xE3` (*eDonkey*) in quanto il pacchetto OfferFiles è definito dal protocollo *eDonkey*. Una volta ricostruito il pacchetto originale il client passa quindi ad analizzarlo come fosse un normale pacchetto ricevuto.

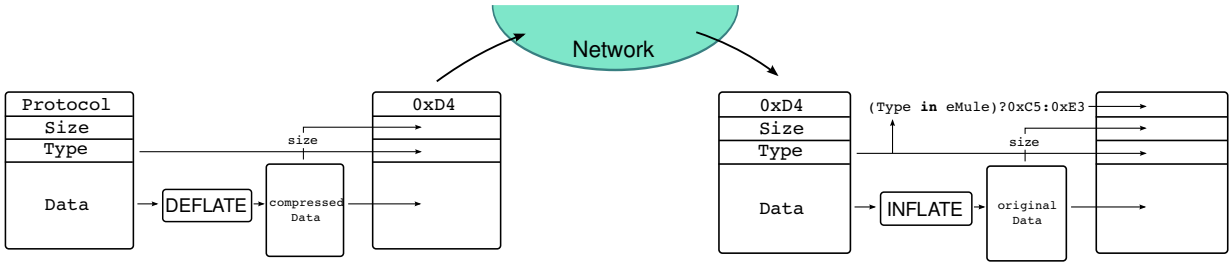


Figura 3.2: Schema di invio/ricezione di un pacchetto compresso.

3.2.3 I pacchetti `FILE_DATA_COMPRESSED`

Oltre ai pacchetti `0xD4`, il protocollo esteso *eMule/eDonkey* definisce un ulteriore tipo di pacchetti in cui viene eseguita la compressione dei dati: il pacchetto `FILE_DATA_COMPRESSED`. Il pacchetto è stato progettato per velocizzare le procedure di scaricamento di contenuti della rete mediante l'utilizzo della compressione.

La struttura di un pacchetto `FILE_DATA_COMPRESSED` (in Tabella 3.4) presenta diverse informazioni. Subito dopo l'header il pacchetto presenta il campo `FileID` tramite il quale viene indicato a quale file della rete appartengono i dati compressi trasportati

Campo	Size [B]	Default	Commento
`Protocol`	1	**`0xC5`**	
`Size`	4		Contiene la lunghezza totale in byte del messaggio escludendone l'header.
`Type`	1	**`0x40`**	Il codice identificativo (*opcode*) del pacchetto `FILE_DATA_COMPRESSED`.
`FileID`	16	NA	Contiene il *FileID* del file.
`Start offset`	4	NA	Indica l'offset iniziale nel file dei dati inviati.
`Reassembled Length`	4	NA	Indica la dimensione totale (in B) dei dati compressi.
`Compressed content`	variabile	NA	Contiene i dati compressi inviati.

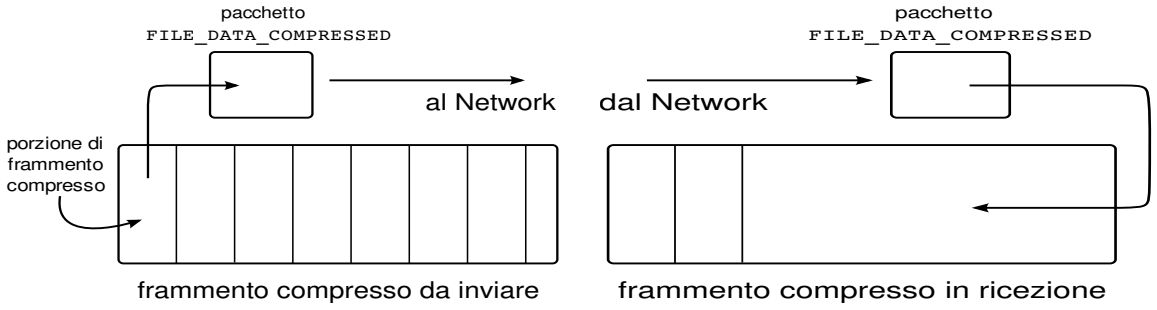
Tabella 3.4: Struttura di un pacchetto `FILE_DATA_COMPRESSED`

dal pacchetto. In particolare i pacchetti `FILE_DATA_COMPRESSED` trasportano al loro interno piccole porzioni di un frammento compresso di file. Ogni pacchetto fornisce quindi indicazioni relative al frammento a cui fanno riferimento. I campi `Start offset` e `Reassembled length` si riferiscono appunto a questo, indicando rispettivamente l'offset all'interno del file e la dimensione compressa del frammento. Da notare come non ci sia nessun campo per identificare l'ordine in cui le porzioni vadano assemblate per riottenere il frammento compresso. Questo può portare ad un certo scetticismo sul funzionamento del meccanismo in quanto l'unico modo per riassemblare correttamente il frammento è che le porzioni contenute nei pacchetti `FILE_DATA_COMPRESSED` arrivino in ordine al destinatario. Straordinariamente questo succede praticamente sempre (salvo casi rarissimi), tradendo le aspettative. Il motivo per cui il sistema funziona è da attribuirsi all'efficienza e del protocollo TCP utilizzato per la trasmissione dei pacchetti, che permette nella quasi totalità dei casi la ricezione ordinata dei pacchetti. Sotto questo punto di vista è accettabile, in presenza di errori, ripere lo scaricamento del frammento.

In trasmissione, ogni pacchetto `FILE_DATA_COMPRESSED` fa riferimento ad un determinato frammento di file identificato dai i campi `FileID` e `Start offset`. Questo frammento, prima dell'invio dei pacchetti corrispondenti, viene compresso ottenedo il relativo frammento compresso. La lunghezza del frammento compresso viene quindi settata nel campo `Reassembled length` di ogni pacchetto `FILE_DATA_COMPRESSED`. A questo punto il client suddivide il frammento compresso in porzioni e carica una di queste porzioni nel campo `Compressed content` del pacchetto. Una volta completata quest'ultima operazione il client provvede allora all'invio del pacchetto nella rete. Variando la porzione caricata in `Compressed content` ad ogni pacchetto inviato il client riesce così a spedire l'intero frammento compresso.

Un client in ricezione colleziona allora tutti i pacchetti `FILE_DATA_COMPRESSED` relativi allo stesso frammento. Una volta ottenuti procede effettuando una **concatenazione**

dei contenuti dei singoli pacchetti (in ordine di arrivo). A questo punto il client effettua un controllo sulla dimensione del frammento compresso ricevuto, verificando che essa corrisponda a quanto riportato dal campo `Reassembled length` dei vari pacchetti. Se c'é corrispondenza si procede con la decompressione del frammento compresso, ottenendo così il frammento originale da salvare sul file alla posizione indicata dal campo `Start offset` dei pacchetti `FILE_DATA_COMPRESSED` ricevuti.



(a) in trasmissione (b) in ricezione

Figura 3.3: Gestione dei pacchetti `FILE_DATA_COMPRESSED`

3.3 Implementazione in *Mulo*

Per permettere a *Mulo* l'invio e la decodifica dei pacchetti conteneti informazioni compresse si è ricorso all'implementazione di due diversi filoni di codice. Il primo riguardante il supporto ai pacchetti compressi (`0xD4`) e il secondo per i pacchetti `FILE_DATA_COMPRESSED`.

3.3.1 I pacchetti `0xD4`

Nello sviluppo del supporto ai pacchetti compressi l'aspetto più complicato da affrontare consiste nel capire in quali contesti viene utilizzato il protocollo compresso. Le specifiche (Sezione 3.2.2) consentono idealmente a qualsiasi tipo di pacchetto di venire compresso senza definire delle regole a riguardo. A fronte di una lunga analisi, composta da un'approfondito studio del codice sorgente di *eMule* e da uno *sniffing*[13] approfondito dei pacchetti scambiati da un client *eMule* con la rete (quest'ultimo mediante un largo utilizzo del software *Wireshark*[14]), è stato possibile capire quali tipi pacchetti vengono utilizzati nel protocollo compresso (Tabella 3.5). Ispirandosi al codice implementato da *eMule* si è quindi optato per una scelta implementativa che fornisca un alto livello di trasparenza all'utilizzo dei pacchetti compressi.

[13]Si definisce sniffing l'attività di intercettazione passiva dei dati che transitano in una rete.
[14]http://www.wireshark.org/

Nome Pacchetto	opcode	Descrizione
EXCHANGE_SOURCES_RESPONSE_1	**0x82**	Pervenuto in risposta al pacchetto EXCHANGE_SOURCES_REQUEST_1, viene utilizzato per comunicare la lista delle fonti conosciute dal peer.
OFFER_FILES	**0x15**	Inviato dal peer al server di riferimento periodicamente, contiene la lista dei file condivisi dal peer.
SEARCH_RESPONSE	**0x99**	Utilizzato in risposta al pacchetto SEARCH_REQUEST fornisce la lista dei contenuti che corrispondono alla chiave di ricerca utilizzata.

Tabella 3.5: Pacchetti scambiati col protocollo compresso.

In *Mulo* ogni pacchetto viene gestito attraverso l'utilizzo di apposite classi. La classe `Packet` definisce il concetto generale di pacchetto da cui, utilizzando l'*ereditarietà*[15], vengono definite le classi `PacketTCP` e `PacketUDP` che distinguono le due principali categorie di pacchetti. Ogni tipo di pacchetto viene quindi gestito tramite una classe che estende le caratteristiche di una di queste due classi principali. Ogni classe, in particolare, presenta un metodo `toBytes()` che a partire da un'istanza di un certo pacchetto restituisce il pacchetto sotto forma di stringa di byte in modo da poterne effettuale l'invio. La gestione dei pacchetti ricevuti è invece gestita attraverso i metodi `decodePacket(...)` delle classi `PacketTCP` e `PacketUDP`. Questi metodi analizzano una stringa di byte in ingresso al fine di generare in uscita un'istanza relativa alla Classe del pacchetto corrispondente.

L'approccio scelto per l'implementazione consiste nella modifica di queste funzioni in merito a tutte quelle classi che gestiscono i pacchetti che possono essere compressi. Per prima cosa si son definite le funzioni per la compressione e decompressione dei dati (`zip(...)`, `unzip(...)`) sfruttando gli opportuni metodi messi a disposizione dal package `java.util.zip` nel quale sono definiti tutti metodi riguardante la compressione/decompressione mediante gli algoritmi *DEFLATE* e *INFLATE*.

Listato 3.1: Pseudocodice per metodi di conversione pacchetto/pacchetto compresso

```
    byte[] packPacket(byte[] packet) {
2     byte[] compressed_data = zip(packet, HEADER_SIZE, packet.length);
      byte[] compressed_packet;
      compressed_packet.put(0xD4);
5     compressed_packet.put(compressed_data.length);
      compressed_packet.put(packet[5]);
      compressed_packet.put(compressed_data);
```

[15]L'ereditarietà è uno dei principi fondamentali della programmazione ad oggetti che consente di definire una classe come classe derivata a partire da una classe preesistente detta superclasse. La sottoclasse eredita implicitamente tutte le caratteristiche (attributi e operazioni) della classe base.

```
8       return compressed_packet;
      }

11    byte[] unpackPacket(byte[] packet) {
        byte[] decompressed_data = unzip(packet, HEADER_SIZE, packet.length);
        byte[] decompressed_packet;
14      if (packet[0] == 0xD4) {
          if (isEMuleType(packet[5])) {
            // if packet is an eMule type we set eMule protocol
17          decompressed_packet.put(0xC5);
          }
          else {
20          // otherwise we hope the packet use eDonkey protocol
            decompressed_packet.put(0xE3);
          }
23      }
        ...
        decompressed_packet.put(decompressed_data.length);
26      decompressed_packet.put(packet[5]);
        decompressed_packet.put(decompressed_data);
        return decompressed_packet;
29    }
```

Successivamente si sono definite delle funzioni che effettuassero in modo autonomo la conversione di un pacchetto TCP o UDP generico in un pacchetto compresso e viceversa (riassunte per la versione TCP in Listato 3.1). La scelta effettuata è stata di aggiungere alla classe `PacketTCP` i metodi di conversione riguardanti i pacchetti TCP e di implementare i metodi di conversione per pacchetti di tipo UDP nella classe `PacketUDP`. I motivi di tale scelta riguardano principalmente la mole di codice da implementare: sfruttando l'ereditarietà delle classi è possibile implementare questi quattro metodi (`packPacket(...)` e `unpackPacket(...)` nelle due versioni) una singola volta per tutti i pacchetti di tipo TCP e UDP. Per ogni pacchetto di Tabella 3.5 si sono andati a modificare i metodi `toBytes()` delle classi associate; implementando i controlli e le operazioni necessarie alla creazione del relativo pacchetto compresso, riassunti nel seguente pseudocodice.

Listato 3.2: Pseudocodice per l'invio di un pacchetto TCP compresso

```
1   class PacketX extends PacketTCP{
      ...
      byte[] toBytes(){
4       byte[] packet = <rappresentazione in byte del pacchetto>
        if (<supportiamo la compressione>) {
          byte[] compressed_packet = packPacket(packet);
7         if(packet.length > compressed_packet.length){
            packet = compressed_packet;
          }
10      }
        return packet;
```

```
    }
13 }
```

Dove `packPacket(...)` è una delle funzioni di conversione sopra descritte, che produce il pacchetto compresso a partire dal pacchetto originale. Le operazioni riguardanti la compressione di un pacchetto UDP non vengono presentate in quanto analoghe alla versione TCP.

Si sono modificati poi i metodi per la decodifica dei pacchetti ricevuti nelle classi `PacketTCP` e `PacketUDP`. Le modifiche apportate ai metodi `decodePacket(...)` cosistono in una semplice decompressione del pacchetto, prima di essere analizzato, se questo utilizza il protocollo compresso.

Listato 3.3: Pseudocodice per la ricezione di un pacchetto TCP compresso

```
  class PacketTCP{
2   ...
    PacketTCP decodePacket(...){
      ...
5     byte[] packet = <byte ricevuti>;
      if(packet[0] == 0xD4){
        packet = unpackPacket(packet);
8     }
      // si procede con l'analisi dei byte contenuti in packet
      ...
11  }
  }
```

Analogamente, le stesse modifiche vengono riportate sul metodo `decodePacket(...)` della classe `PacketUDP`.

3.3.2 I pacchetti `FILE_DATA_COMPRESSED`

Per quanto riguarda l'implementazione del supporto in *Mulo* dell'invio/ricezione di pacchetti `FILE_DATA_COMPRESSED`, lo sviluppo si è svolto principalmente sulle classi `Download`, `UploadSession` e `UploadManager`, che gestiscono le procedure di download e upload. In `PacketsTCP.java` è stata estesa da `PacketTCP` una nuova classe `PacketTCP-FileDataCompressed` che implementi la struttura del nuovo pacchetto. Le modifiche apportate in `Download.java` riguardano la riprogettazione della una procedura di download in modo che utilizzi anche i pacchetti `FILE_DATA_COMPRESSED`. La nuova procedura di download va a gestire quindi la ricezione di due possibili pacchetti: i `FILE_DATA`, privi di compressione, e i `FILE_DATA_COMPRESSED`. Se il pacchetto ricevuto è `FILE_DATA` il download procede con la procedura classica salvando i dati contenuti. Nel caso in cui il pacchetto sia di tipo `FILE_DATA_COMPRESSED` *Mulo* avvia una procedura alternativa. Nel seguente pseudocodice è possibile osservare la riorganizzazione apportata alla procedura di download.

Listato 3.4: Pseudocodice della procedura di Download

```
while(<download non completato>){
  ...
  PacketTCP received_packet = <nuovo pacchetto ricevuto>;
  if(received_packet instanceof PacketTCPFileDataCompressed){
    // procedura salvataggio dati compressi
    storeDataCompressed(received_packet,...);
  }
  if(received_packet instanceof PacketTCPFileData){
    // procedura standard di salvataggio
  }
  ...
}
// download completato
```

Rimane da definire il corpo del metodo `storeDataCompressed(...)` che deve eseguire le procedure di ricezione di un pacchetto `FILE_DATA_COMPRESSED`, precedentemente descritte.

Come si può notare, nel pseudocodice presentato in Listato 3.5, viene mantenuto un buffer `compressed_data_buffer` in cui i vari contenuti compressi dei pacchetti `FILE_DATA_COMPRESSED` vengono salvati e mantenuti. In caso di errore nella procedura questo buffer viene svuotato e la procedura di download del frammento di file deve ricominciare da capo. Nel cao invece tutto avvenga correttamente man mano che sopraggiungono pacchetti `FILE_DATA_COMPRESSED` il buffer si riempie fino a raggiungere la lunghezza specificata dal campo `Reassembled length` dei pacchetti. A questo punto i dati contenuti nel buffer vengono decompressi (`unzip(...)`) e i byte così ottenuti vengono salvati su disco come se fossero dati provenienti dai pacchetti standard. Al termine della procedura di salvataggio `compressed_data_buffer` viene svuotato così da prepararsi al salvataggio del prossimo frammento compresso.

Listato 3.5: Pseudocodice della procedura `storeDataCompressed(...)`

```
byte[] compressed_data_buffer;
storeDataCompressed(PacketTCPFileDataCompressed packet,...) {
  // controlli sulla correttezza degli offset
  ...
  compressed_data_buffer.put(packet.compressedData);
  if(compressed_data_buffer.length == packet.reassembledLength){
    // frammento compresso di file completamente ricostruito
    byte[] data = unzip(compressed_data_buffer)
    // procedura standard di salvataggio nel file contenuti in data
    ...
    compressed_data_buffer = null;
  }
}
```

Le modifiche apportate in `UploadManager` e `UploadSession` consistono invece in lievi modifiche alla procedura di upload utilizzata da *Mulo*. In particolare quanto *Mulo* deve spedire un file o parte di esso ad un'altro peer della rete le modifiche controllano se sia *Mulo* che il peer destinatario supportano la compressione. Se uno dei due non supporta la compressione, l'upload procede con la procedura regolare. Se, invece, entrambi supportano la compressione; *Mulo* esegue una procedura di upload alternativa che si basa sull'utilizzo dei pacchetti `FILE_DATA_COMPRESSED` ed è riassunta nel seguente pseudocodice.

Listato 3.6: Pseudocodice della procedura `sendNextCompressedDataPacket()`

```
List<long[]> requestedFragments;
byte[] compressedFragment;
int compressed_offset;
sendNextCompressedDataPacket(){
  PacketTCPFileDataCompressed dataPacket;
  if(compressed_offset + BYTES_SENT_PER_PACKET < compressedFragment.length){
    // invio porzione di frammento compresso.
    dataPacket = new PacketTCPFileDataCompressed(compressedFragment,compressed_offset,
        BYTES_SENT_PER_PACKET,...);
    compressed_offset += BYTES_SENT_PER_PACKET;
  }
  else{
    // invio ultima porzione di frammento compresso.
    dataPacket = new PacketTCPFileDataCompressed(compressedFragment,compressed_offset,
        BYTES_SENT_PER_PACKET,...);
    compressed_offset += BYTES_SENT_PER_PACKET;
    requestedFragments.remove(0);
    if(!requestedFragments.isEmpty()){
      compressedFragment = zip(file,nextFragment[0],nextFragment[1]);
    }
  }
  send(dataPacket);
}
```

`requestedFragments` contiene la lista dei frammenti richiesti da inviare al peer. All'avvio dell'upload in `compressed_fragment` vengono caricati i dati compressi del primo frammento da spedire. Ogni qual volta viene invocato il metodo `sendNextCompressedDataPacket()` viene inviato al peer un pacchetto `FILE_DATA_COMPRESSED` contenete una porzione del frammento compresso della dimensione di `BYTES_SENT_PER_PACKET` bytes. Man mano che il metodo viene invocato `compressed_offset` nuove porzioni del frammento vengono inviate fino a che, all'invio dell'ultima porzione, avviene il cambio col successivo frammento. Il metodo procede, se possibile, col comprimere il prossimo frammento presente in `requestedFragments` e salvare i relativi dati compressi in `compressed_fragment` e poter così reiniziare il ciclo.

Per una visione completa e dettagliata dell'intero lavoro prodotto in *Mulo* per il supporto ai pacchetti compressi `0xD4` e ai pacchetti `FILE_DATA_COMPRESSED` si rimanda ad Appendice B.2; dove è disponibile tutto il codice *Java* prodotto.

3.3.3 Testing

Allo stato attuale l'implementazione di *Mulo* risulta già a buon punto permettendo gran parte delle funzioni di base. Per questo modivo l'implementazione in *Mulo* è soggetta a due tipologie di test eseguibili: i test *JUnit*, utilizzati seguendo il pieno principio *XP*, e i test di *funzionamento generale*. Mentre i primi vengono utilizzati per aiutare lo sviluppo di codice "nuovo" e verificano modularmente il funzionamento di *Mulo*; i secondi sono molto importanti per valutare il funzionamento generale e le prestazioni della nuova feature implementata. Nell'implementazione del supporto dei pacchetti compressi `0xD4` in un primo tempo si sono effettuati test di tipo modulare. In particolare, tramite varie operazioni di sniffing con Wireshark, sono stati ricavati i byte in uscita da un client *eMule* riguardanti l'invio di un pacchetto di tipo `OFFER_FILES` con compressione abilitata e non. Ottenute le due versioni del pacchetto si sono progettati dei test *JUnit* (in Listato 3.7) per verificare il funzionamento delle due funzioni chiave `packPacket(...)` e `unpackPacket(...)`.

Listato 3.7: Test per la compressione/decompressione in `PacketsTCPTest.java`

```
2310    /**
2311     * A JUnit Test for the Packet Decompression:
2312     */
2313    @Test
2314    public void testPacketDecompression() {
2315      byte[] compressed_pWireshark = new byte[] {
2316          (byte)0xd4 , // protocol
2317          (byte)0x81 ,0x02 ,0x00 ,0x00 , // length
2318          0x15 , // type
2319          0x78 ,
2956          0x13 ,
2957          0x48 ,
2958          (byte)0xa0 };
2959
2960      byte[] uncompressed_pWireshark = {
2961          (byte)0xE3 , // protocol
2962          0x65 ,0x03 ,0x00 ,0x00 , // size
2963          0x15 , // type
2964          0x09 ,
2965          0x00 ,
3830          0x03 ,
3831          0x04 };
3832
3833      byte[] unPackResult = PacketTCP.unpackPacket(compressed_pWireshark);
3834      assertEquals("Unmatched length", uncompressed_pWireshark.length, unPackResult.length);
```

```
3835      for ( int i = 0 ; i < Math.min(uncompressed_pWireshark.length, unPackResult.length) ; i
              ++ ) {
3836        assertEquals("Wrong byte in the unpacked packet in the position " + i + ", ",
              uncompressed_pWireshark[i], unPackResult[i]);
3837      }
3838    }
3839
3840    /**
3841     * A JUnit Test for the Packet Compression:
3842     */
3843    @Test
3844    public void testPacketCompression() {
3845      byte[] compressed_pWireshark = new byte[] {
4491          (byte)0xa0 };
4492
4493      byte[] uncompressed_pWireshark = {
5367          0x04 };
5368
5369      byte[] packResult = PacketTCP.packPacket(uncompressed_pWireshark);
5370      assertEquals("Unmatched length", compressed_pWireshark.length, packResult.length);
5371      for ( int i = 0 ; i < Math.min(compressed_pWireshark.length, packResult.length) ; i++ )
              {
5372        System.out.println(">" + i + " " + Utils.byteToHex(compressed_pWireshark[i]) + "-" +
              Utils.byteToHex(packResult[i]));
5373        assertEquals("Wrong byte in the packed packet in the position " + i + ", ",
              compressed_pWireshark[i], packResult[i]);
5374      }
5375    }
```

Per quanto riguarda i test relativi al supporto dei pacchetti `FILE_DATA_COMPRESSED` in invio e ricezione si sono sfruttati i test modulari già presenti riguardanti le procedure di download e upload, occupandosi solamente di settare opportunamente il valore *MiscOptions1* (Listato 3.8).

Si è infine ricorso a numerosi test di funzionamento generale in *Mulo* per verificare che i tutti i pacchetti definiti fossero completamente supportati e che le procedure implementate non minassero il funzionamento dell'intero plugin *Mulo*.

3.3.4 Risultati e Conclusioni

L'implementazione del supporto alla compressione porta le funzionalità offerte da *Mulo* un passo più vicino a quelle di *eMule* raggiungendo un buon livello di competitività con il client più diffuso. La compressione comporta infatti discreti abbassamenti sui tempi di download dei file e nel completare le richerche. Dietro a questi miglioramenti si ha però un aumento delle elaborazioni eseguite da parte di *Mulo*. Ad ogni pacchetto compresso ricevuto mulo deve infatti decomprimere i dati prima di poterli analizzare. Allo stesso modo ogni pacchetto compresso che *Mulo* si prepara ad inviare comporta la compressione dei dati originali. Questo *trade-off* tra velocità e peso computazionale può rilevarsi cri-

Listato 3.8: Valore di `MISC_OPTS1_VALUE` in `Config.java`

```
     /**
      * eMule Misc Options 1 field, telling the world what do we support:<br>
72    * - 1st nibble: AICH Version (3 bits) + Unicode Support,<br>
      * - 2nd nibble: UDP Version,<br>
      * - 3rd nibble: Data Compression Version,<br>
75    * - 4th nibble: Secure Identification Support,<br>
      * - 5th nibble: Sources Exchange 1 Version,<br>
      * - 6th nibble: Extended Requests Version,<br>
78    * - 7th nibble: Accept Comments Version,<br>
      * - 8th nibble: Peer Cache Support + *No* View Shared Files Support + MultiPacket Support
           + Preview Support.
      */
81    static int MISC_OPTS1_VALUE = 0x34111216;
```

tico in macchine che hanno a capacità di elaborazione limitate. Tuttavia gli elaboratori attualmente in commercio non soffrono di questi problemi, avendo capacità ben superiori a quelle richieste da *Mulo*, limitando così il problema alle macchine datate.

Capitolo 4

SuperFilesharing

In questo capitolo viene introdotta l'idea di "SuperFilesharing" in *PariPari*. Vengono inizialmente presentate le idee di base e gli scopi che si vogliono raggiungere implementando un sistema di download *multi-rete* in *PariPari* per poi, infine, spiegare il lavoro svolto su *Mulo*.

4.1 Osservazioni sul filesharing

L'esigenza di dover condividere file e informazioni tra persone sparse nel mondo ha portato alla costruzione di molte grandi reti di *filesharing* adatte allo scopo. Ogni rete ha i propri protocolli e le proprie caratteristiche con aspetti positivi e negativi. Un client $\mathcal{A}$ di una rete X è libero di condividere una certa quantità di file personali, che potranno essere liberamente scaricabili dagli altri client della rete X.

Dei file condivisi dal client $\mathcal{A}$, è possibile che un particolare file F venga scaricato da un'altro client $\mathcal{B}$ della stessa rete X. Questo client ,scaricando il file F, aiuta la diffusione e la reperibilità del file all'interno nella rete X. Il client $\mathcal{B}$, ottenuto il file F, può decidere, in quanto utente umano, di condividere a sua volta questo file in una rete Y differente dalla rete X. I client della rete Y possono quindi scaricare il file, aumentando la reperibilità del file F nella rete Y. Si noti che a loro volta i client possono condividere il file F in ancora altre reti differenti Z, W,... rendendo il file F teoricamente reperibile in qualsiasi rete. Riassumendo, si può dire che il file F viene condiviso da diversi client di diverse reti *filesharing*. Questa affermazione, data l'arbitrarietà del file F, può essere applicata a qualsiasi file condiviso esistente deducendo che per ogni file reperibile da una certa rete esiste, con grande probabilità, la possibilità che quello stesso file sia reperibile anche in altre reti. Da questa ultima osservazione viene lecito pensare che sia possibile scaricare un file F sfruttando tutte le reti in cui è reperibile. Questo approccio non solo comporta un maggior numero di fonti contattabili dato che il numero dei client di tutte le possibili reti che possiedono il file è maggiore del numero di client che possiedono il file in una certa rete X; ma inoltre permette, nel caso in cui in una determinata rete il file non sia completamente disponibile, di poterlo scaricare ugualmente appoggiandosi ai client delle

altre reti. In prima osservazione l'approccio *multi-rete* allo scaricamento di un file comporta notevoli vantaggi. Tuttavia esiste un unico problema: *l'identificazione del file*. Deve cioé esistere un metodo di confronto che con certezza possa distinguere quando due file F_1 di una rete X e F_2 di una rete Y siano in realtà lo stesso file F.

4.1.1 L'idea

Da tutte le considerazioni precedenti nasce così l'idea di progettare *SuperFilesharing*: una piattaforma che astrae il concetto di download da reti come *eMule/eDonkey* o *Torrent* raggruppandolo in un generico concetto di *super download*. Volendo definire una piattaforma che possa essere adattabile e facilmente configurabile, *SuperFilesharing* viene strutturato su due livelli:

- nel livello superiore vi è *Multi Network Filesharing Manager*(*MNFM*): la componente che gestisce il download del file da un punto di vista globale, partizionando il suo scaricamento nelle varie reti.

- al livello inferiore si contano vari **moduli** che gestiscono le operazioni sulle varie reti. Le operazioni effettuate su ogni rete vengono gestite da un modulo dedicato.

All'interno di *MNFM* viene gestita tutta la componente logica del sistema. *MNFM* si occupa di assegnare, in maniera efficiente, i frammenti di file da scaricare alle varie reti. L'implementazione di questa logica è particolarmente critica in quanto deve garantire l'assegnamento di tutti i frammenti del file e deve soprattutto assegnare i frammenti in modo che siano disponibili nella rete a loro assegnata. Oltre a questo aspetto *MNFM* ha le mansioni di controllo e gestione degli errori: deve cioé verificare che i frammenti scaricati dalle reti appartengano al medesimo file e, in caso contrario, gestire opportunamente il problema. *MNFM* assolve a queste funzioni comunicando coi vari moduli attraverso vari messaggi, che istruiscono i moduli su quali operazioni effettuare.

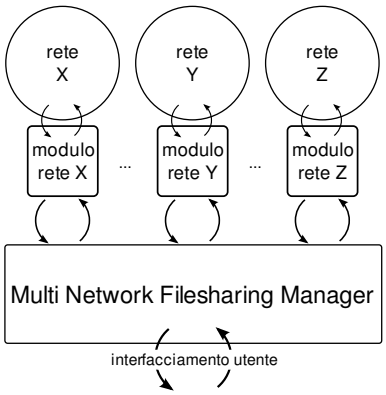


Figura 4.1: Struttura per SuperFilesharing.

4.2 Progettazione

In Sezione 4.1.1 si sono viste le principali idee che tracciano la struttura di *SuperFilesharing*. Anche se la struttura viene descritta ampiamente lascia comunque un grande interrogativo: "Come si certifica che file su reti distinte siano in realtà lo stesso file?". Questa domanda risulta particolarmente critica per l'implementazione di un sistema concreto e per questo viene deciso in *PariPari* di optare per una versione semplificata del sistema, eventualmente ampliandola dopo aver definito una base solida e funzionante. La semplificazione consiste principalmente nel ridimensionamento delle reti supportate: invece di definire il supporto per qualsiasi rete di filesharing generica (e quindi la gestione di un numero variabile di moduli), la piattaforma in *PariPari* supporta solamente le reti maggiormente utilizzate: *eMule/eDonkey* (con *Kad*) e *Torrent*. Implementare la gestione di due soli moduli rispetto alla gestione di un numero variabile semplifica notevolmente il codice da produrre consentendo di concentrarsi sulla risoluzione del problema di identificazione del file.

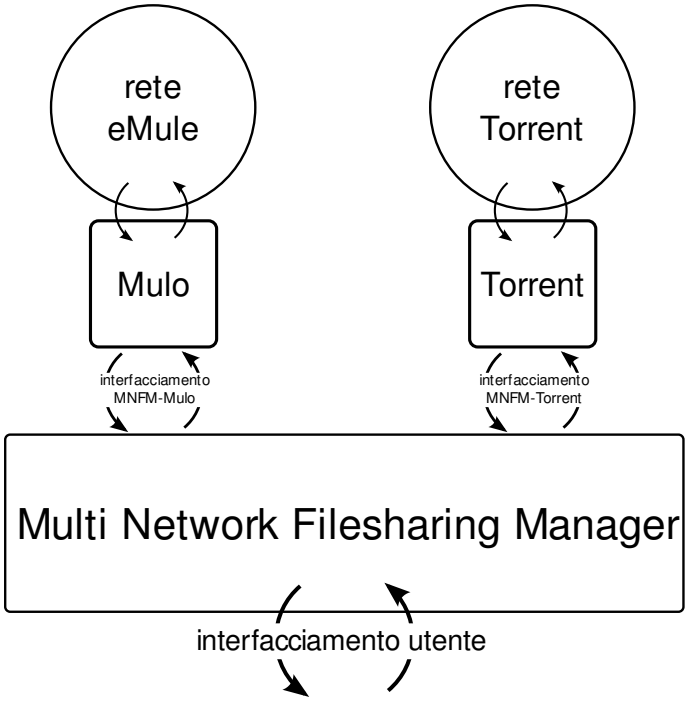


Figura 4.2: Struttura Semplificata per SuperFilesharing in *PariPari*.

4.2.1 Funzionamento di *SuperFilesharing*

Il funzionamento di *SuperFilesharing* consiste in una procedura di download multi-rete in cui un file obiettivo viene scaricato sfruttando i plugin *Mulo* e *Torrent* di *PariPari* come *moduli* del sistema di *SuperFilesharing* per l'interfacciamento alle rispettive reti.

Startup e fasi iniziali del download

Per iniziare lo scaricamento di un file è necessario fornire a *SuperFilesharing* un file *.torrent*[1] e una stringa di ricerca che identificano il file della rete Torrent da scaricare. Nelle prime fasi della procedura di scaricamento *MNFM* assegna a *Torrent* il compito di scaricare l'intero file, a partire dai pezzi iniali e finali. Parallelamente a questo *MNFM* comunica a *Mulo* il nome e la dimensione del file specificati dal *.torrent* fornito dall'utente.

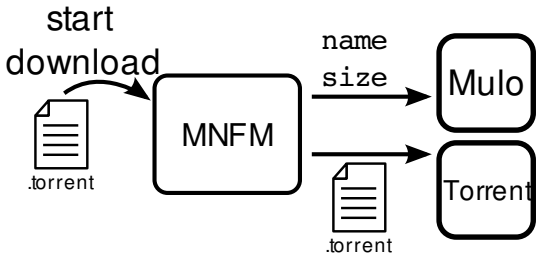


Figura 4.3: Startup in *SuperFilesharing*.

Ricerca dei *match* nella rete *eMule/eDonkey*

Il file *.torrent* fornito al *SuperFilesharing* da una serie di informazioni necessarie all'individuazione e lo scaricamento di un preciso file dalla rete *Torrent*. Per poter scaricare questo file dalla rete *eMule/eDonkey* il modulo *Mulo* deve prima di tutto verificare che tale file compaia tra i contenuti della rete. Per questo motivo *Mulo* ricerca tutti i possibili file della rete *eMule/eDonkey* che presentano nome e dimensione corrispondenti a quanto specificato dal file *.torrent*. I "match" ottenuti dalla ricerca rappresentano tutti quei file che potrebbero corrispondere al file della rete *Torrent* che si sta scaricando.

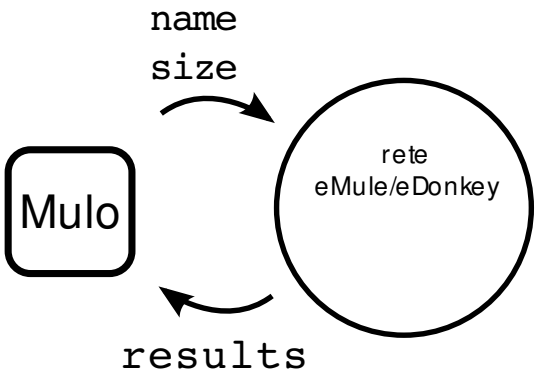


Figura 4.4: Ricerca match in *SuperFilesharing*.

[1] Un file *.torrent* è un file di tipo binario utilizzato dalla rete *Torrent*.

Reperimento degli hash

Sfruttando il sistema *AICH* della rete *eMule*, *Mulo* richiede, per ogni match trovato, tutti gli hash *SHA1* relativi ai chunk di inizio e fine file.

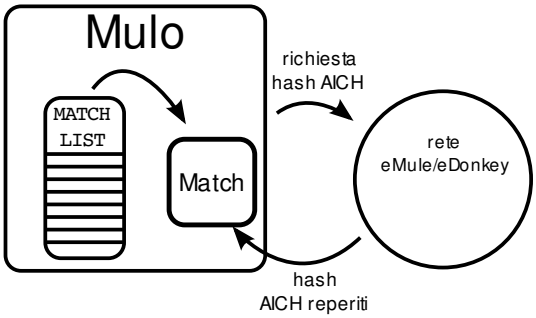


Figura 4.5: Reperimento degli hash in *SuperFilesharing*.

Filtraggio della *matchList*

Tutti i match relativi al file vengono mantenuti da *MNFM* in una apposita lista *matchList*. Per ogni match in lista vengono inoltre mantenute tutte le informazioni relative come numero di fonti e, soprattutto, gli hash SHA1 reperiti. Via via che il modulo *Torrent* completa lo scaricamento delle porzioni iniziali e finali del file, *MNFM* controlla i dati scaricati e, utilizzando la segmentazione in chunk, effettua il calcolo degli hash SHA1 relativi. Ottenuti questi hash *MNFM* filtra la *matchList* eliminando da questa tutti i match i cui hash reperiti non corrispondono a quelli calcolati.

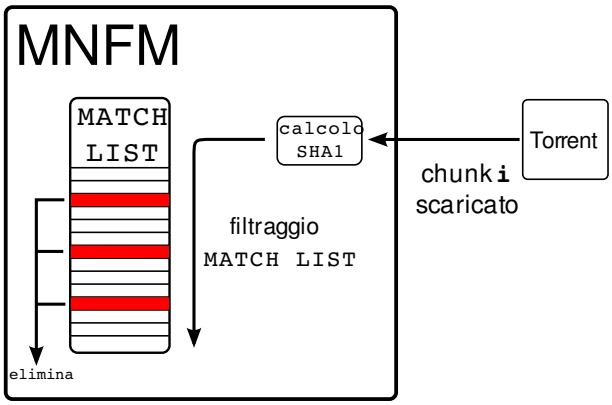


Figura 4.6: Filtraggio dei match in *SuperFilesharing*.

Assegnamento dei chunk

Man mano che la *matchList* viene filtrata da *MNFM*, sempre più match vengono scartati. Questo processo continua fino a che tutti i match della lista vengono scartati o solo un match

rimane in lista. Se la lista risulta vuota dopo il filtraggio significa che nessun file nella rete *eMule/eDonkey* corrisponde al file della rete *Torrent* che si vuole scaricare. *MNFM* procede in questo caso evitando lo scaricamento del file con la rete *eMule/eDonkey*, lasciando lo scaricamento dell'intero file al modulo *Torrent*. Se, dopo il filtraggio, un solo match rimane nella *matchList* vuol dire che questo file della rete *eMule/eDonkey* corrisponde al file da scaricare. *MNFM*, avendo identificato il file da scaricare in entrambe le reti, può quindi procedere assegnando a *Mulo* e *Torrent* le porzioni di file da scaricare, distribuendole tra i due moduli.

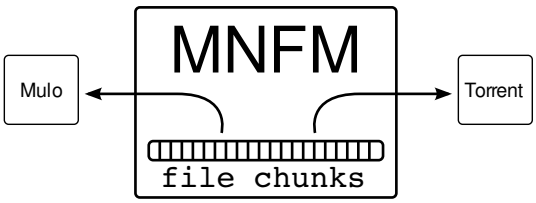


Figura 4.7: Assegnamento dei chunk in *SuperFilesharing*.

Più precisamente le porzioni da assegnare vengono definite da *MNFM* utilizzando la segmentazione in chunk del file su rete *eMule/eDonkey*. Ai due moduli vengono assegnate porzioni di file corrispondenti ai chunk del file. Questa scelta viene fatta per avere un grado di affidabilità maggiore da parte dei chunk scaricati da *Mulo* in quanto è possibile verificare l'assenza di errori nello scaricamento sfruttando la corrispondenza con gli hash *SHA1* di *AICH*.

Mapping chunk-block

Mentre la rete *eMule/eDonkey* utilizza la segmentazione in chunk e il relativo modulo può quindi effettuare il download diretto dei chunk a lui assegnati, i file in *Torrent* sono trattati attraverso due principali segmentazioni:

- Suddivisione in *block*, che suddividono il file in frammenti da 16 KB.
- Suddivisione in *piece*, che suddividono il file in frammenti di dimensione arbitraria, definita dal file *.torrent* relativo, ogni *piece* contiene è composto da due o più *block*.

Data la diversità dalle segmentazioni utilizzate nella rete *eMule/eDonkey*, il modulo *Torrent*, per poter effettuare lo scaricamento del chunk assegnatogli, è costretto ad effettuare un mapping tra i chunk in cui il file è suddiviso nella rete *eMule/eDonkey* e i block in cui viene segmentato nella rete *Torrent*. Ottenendo così i block relativi al chunk da scaricare.

Verifica dei *piece*

Il file *.torrent* fornito a *SuperFilesharing*, oltre a contenere informazioni per l'identificazione del file da scaricare, mantiene al suo interno tutta una serie di informazioni legate al

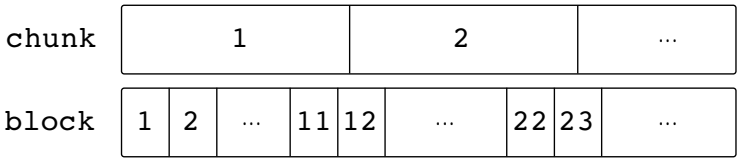


Figura 4.8: Mapping *chunk-block* in *SuperFilesharing*.

controllo e la gestione degli errori. Nello specifico il *.torrent* presenta una lista di hash *SHA1* ognuno corrispondente ad un piece del file. Per controllare che non ci siano errori di scaricamento bisogna quindi attendere lo scaricamento di un piece completo, calcolare il rispettivo hash *SHA1* e verificare che corrisponda a quello riportato dal file *.torrent* associato.

Per questo motivo *MNFM* effettua, ad ogni *chunk* scaricato dai uno dei moduli, un controllo sullo stato dei *piece*. Se i *chunk* relativi ad una porzione di file che corrisponde ad un *piece* sono stati scaricati, *MNFM* calcola l'hash *SHA1* sul piece in questione ed effettua il controllo con quanto riportato sul *.torrent*.

Nel caso in cui gli hash non corrispondano, i dati scaricati relativi al *piece* vengono eliminati e i rispettivi *chunk* riassegnati ai due moduli *Mulo* e *Torrent*. Se troppe verifiche falliscono, può significare che il file su rete *eMule/eDonkey* non corrisponde perfettamente al file della rete *Torrent* che si vuole scaricare. In questo caso allora *MNFM* interrompe lo scaricamento del file col modulo *Mulo*, assegnando tutti i *chunk* rimanenti al modulo *Torrent*.

In caso il controllo vada a buon fine i dati del piece vengono reputati corretti e vengono dunque salvati permanentemente. Unica eccezione viene fatta per i cosidetti chunk "limitrofi". La diversità di segmentazione tra *chunk* e *piece* non permette ad un *piece* di contenere perfettamente un numero intero di *chunk*. La verifica del *piece* comporta quindi una verifica parziale per i chunk limitrofi che non possono essere reputati pienamente verificati. I *chunk* limitrofi si trovano infatti "a cavallo" tra due *piece* consecutivi e per essere considerati pienamente verificati devono venire verificati entrambi i *piece* consecutivi che li contengono.

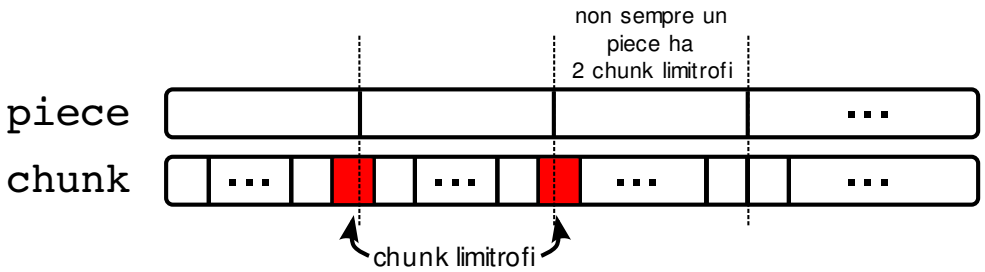


Figura 4.9: Chunk limitrofi di un piece.

Completamento del download

Con l'avanzare del tempo, i moduli *Torrent* e *Mulo* scaricano i *chunk* a loro assegnati. Con l'aumentare dei *chunk* scaricati, sempre più *piece* potranno essere verificati e di conseguenza sempre più dati verranno salvati. Il processo continua in questo modo fino a che tutti i *piece* che compongono il file non sono verificati correttamente. In questo caso *MNFM* considera il file scaricato completamente e *correttamente*, notificando all'utente l'avvenuto successo della procedura di download.

4.2.2 Messaggi *MNFM-moduli*

Rimane da definire in che modo avvengono le comunicazioni tra *MNFM* e i moduli *Mulo* e *Torrent*. Analizzando il funzionamento di *SuperFilesharing* (in Sezione 4.2.1) è possibile osservare diversi momenti in cui avvengono scambi di messaggi tra *MNFM* e i moduli:

- Nelle operazioni legate allo scaricamento dei chunk iniziali e finali del file dalla rete *Torrent*.
- Nelle operazioni per la ricerca dei match del file nela rete *eMule/eDonkey*.
- Per le operazioni di reperimento degli hash *AICH* del file dalla rete *eMule/eDonkey*.
- Nelle operazioni legate all'assegnamento dei chunk.
- Nelle operazioni di salvataggio dei chunk scaricati.
- Per le operazioni relative alla verifica dei *piece*.

All'avvio della procedura di download *MNFM* comunica al modulo *Torrent* un messaggio di `startDownload`, passandogli il file *.torrent* per l'identificazione del file da scaricare. Con questo messaggio il modulo *Torrent* comprende di dover iniziare lo scaricamento dell'intero file, partendo però dai chunk iniziali e finali. Di fatto affinché i dati scaricati dal modulo *Torrent* siano affidabili, vengono scaricati primo e ultimo piece relativi al file in modo da poterne effettuare la verifica. In tutta risposta al messaggio di `startDownload`, il modulo *Torrent* risponde con due messaggi di notifica differenti: `downloadedComparableStartChunks` non appena viene scaricato e verificato il piece iniziale e `downloadedComparableEndChunks` quando viene completamente scaricato e verificato il piece finale.

Sempre allo startup, *MNFM* trasmette al modulo *Mulo* un messaggio `searchWithMulo` passando nome e dimensione del file da ricercare nella rete *eMule/eDonkey*. *Mulo*, così istruito, effettua la ricerca di tutti i possibili match della rete per quel nome file e quel size, restituentone una lista a *MNFM*.

MNFM richiede il reperimento degli hash di un particolare match passandolo a *Mulo* attraverso un messaggio di `getAICHHashWithMulo`. Il modulo *Mulo* comprende così di dover ricercare, tramite *AICH* gli hash *SHA1* dei chunk iniziali e finali. Il modulo risponde al messaggio `getAICHHashWithMulo` con due messaggi di risposta: `foundStartAICHHashes`

comunicando una serie di hash *SHA1* relativi ai chunk iniziali relativi ed il match a cui fanno riferimento. `foundEndAICHHashes` che comunica a metacoso gli hash relativi ai chunk finali ed il match a cui si riferiscono.

Ogni qual volta *MNFM* riorganizza i chunk assegnati ai due moduli, il cambiamento viene notificato attraverso un messaggio di `updateDownloadInfo`, tramite il quale i due moduli aggiornano la loro lista di chunk da scaricare in modo da rispecchiare le nuove assegnazioni.

Allo scaricamento di un chunk, invece, i moduli non vanno a salvare i dati direttamente sul file in quanto le operazioni di salvataggio vengono pienamente gestite da *MNFM*. I moduli passano a *MNFM* i dati scaricati relativi ad un certo chunk attraverso un messaggio `saveChunk`, comunicando in particolar modo i byte scaricati e l'indice del chunk a cui i dati fanno riferimento. In questo modo *MNFM* può salvare i dati del chunk nella giusta posizione e, se possibile, effettuare la verifica dei piece.

Per la verifica di un piece i cui chunk relativi sono tutti stati scaricati, *MNFM* richiede al modulo *Torrent* di effettuare il controllo. *MNFM* richiede questo inviando il piece da verificare attraverso un messaggio di `verify`. Con questo messaggio, il modulo *Torrent* effettua la verifica sul piece, comunicando a *MNFM* se il piece è stato scaricato correttamente o meno.

4.2.3 Mantenimento delle informazioni

Durante tutto il funzionamento di *SuperFilesharing* è necessario che le informazioni necessarie per il proseguimento del processo siano mantenute all'interno di determinate strutture dati facilmente accessibili. Nello specifico *SuperFilesharing* dovrà gestire lo scaricamento di più file, rendendo necessaria una struttura sulla quale salvare le la lista download in corso.

Per ogni file da scaricare in lista vengono mantenute le relative informazioni: il file *.torrent* che lo identifica, il match della rete *eMule/eDonkey* a cui corrisponde e, soprattutto, la lista dei chunk da cui è composto ed il loro stato. Osservando il funzionamento descritto in Sezione 4.2.1 è possibile notare come ogni chunk di cui è composto il file da scaricare può trovarsi in 5 stati distinti:

EMPTY il chunk non è ancora stato assegnato a nessuno dei due moduli *Mulo* o *Torrent* e deve quindi ancora essere scaricato.

MULO il chunk è stato assegnato al modulo *Mulo*, ma deve essere ancora scaricato completamente.

TORRENT il chunk è stato assegnato al modulo *Torrent*, ma non è ancora stato scaricato del tutto.

UNVERIFIED il chunk è stato scaricato dal modulo a cui era stato assegnato, ma deve essere ancora verificato.

VERIFIED il chunk è stato scaricato ed è stato correttamente verificato.

Il salvataggio degli stati dei chunk è di vitale importanza per la procedura di scaricamento: al termine dello scaricamento tutti i chunk si trovano nello stato **VERIFIED**, e grazie a questo *MNFM* è in grado di distinguere, basandosi sugli stati dei chunk, quando un download è stato completato.

Per il mantenimento della lista dei download e delle relative informazioni si decide quindi di strutturare un apposito file *XML*. Questo *XML* funziona da tramite tra *MNFM* e i moduli *Mulo* e *Torrent* nel senso che tramite questo *MNFM* comunicherà le nuove assegnazioni dei chunk. Alla ricezione del messaggio `updateDownloadInfo` i moduli accedono in lettura al file *XML* per osservare le nuove assegnazioni impartite da *MNFM* e da queste derivano i chunk da scaricare.

Un possibile esempio della struttura del file *XML* è dato in Listato 4.1, dove si puo notare la lista dei download (tag `<Sdownloadlist>`) con al suo interno i vari download (tag `<Sdownload>`). Ogni `<Sdownload>` va a contenere tutte le informazioni generali sul file in scaricamento e in particolare l'elenco delle parti *eDonkey* da cui è composto (tag `<part>`) ogni parte a contiene dunque le informazioni sugli stati dei chunk ad essa appartenenti (parametro `status`), i quali vengono rappresentati da lettere identificative:

E per lo stato *EMPTY*

M per lo stato *MULO*

T per lo stato *TORRENT*

U per lo stato *UNVERIFIED*

V per lo stato *VERIFIED*

Listato 4.1: Esempio di `downloadList.xml` in *SuperFilesharing*

```
<?xml version="1.0" encoding="UTF8"?>
<Sdownloadlist version="0.9" items="1">
  <Sdownload name="Deep+purple+-+Bananas+-+01+-+House+of+pain.mp3"
  size="5139144" status="active" totalTime="0"
  md4="E252A6048AC3E87ED3AE13395142F8BC"
  sha1="LJFI5K6OBKSQBDKXGTWOJK5IWPUPSWWU" ... >
    <part n="0" status="UUUUUTMMMTTEETUVVVUTEUEUVVVVV"
    md4="E252A6048AC3E87ED3AE13395142F8BC"
    sha1="LJFI5K6OBKSQBDKXGTWOJK5IWPUPSWWU" />
  </Sdownload>
</Sdownloadlist>
```

4.3 Implementazione

Una volta ben chiare le meccaniche di funzionamento di *SuperFilesharing* e le strutture dati ad esso collegate è possibile procedere all'implementazione del sistema. Per prima cosa è necessario definire come avvengono gli scambi di messaggi tra i moduli *Mulo* e *Torrent* e *MNFM*. Viene deciso che le comunicazioni tra *MNFM* ed i moduli consisteranno in richiami di determinate funzioni di interfacciamento implementate dai moduli e da *MNFM* per poter effettuare le operazioni. Tutti i messaggi di comunicazione visti in Sezione 4.2.2 vengono quindi implementati come appositi metodi di interfacciamento.

In questa tesi viene svolta l'analisi del lavoro riguardante la sola implementazione delle funzioni di interfacciamento tra *MNFM* e *Mulo*. L'implementazione delle funzioni si svolge in una porzione ben precisa del codice di *Mulo*:

`Match.java` , che definisce una struttura sulla quale mantenere le informazioni di una potenziale corrispondeza del file da scaricare sulla rete *eMule/eDonkey*.

`HashFinder.java` , che implementa tutte le operazioni per il recupero degli hash *AICH*.

`Download.java` , dove di aggiunge la nuova modalità di download dalla rete *eMule/eDonkey*.

`MuloMNFMInterface.java` , che contiene al suo interno tutti i metodi richiamabili da *MNFM*.

La classe `Match` rappresenta di fatto un file della rete *eMule/eDonkey* e per tanto deve mantenere informazioni necessarie alla sua reperibilità, prime fra tutto il *FileID*, il *rootHash AICH* e *la quantità di fonti disponibili*. Oltre a queste viene mantenuta in match la lista per gli hash da reperire e un riferimento a `Download` per quando *MNFM* deciderà di avviare il download del match, dopo gli opportuni controlli. I match vengono istanziati a partire dai risultati di una ricerca *eMule/eDonkey*. In *Mulo* i risultati delle ricerche sono gestiti dalla classe `SearchResult`, in `Match` viene pertanto definito il metodo `createFromSearchResult(SearchResult result)`, che a partire da un oggetto `SearchResult` istanzia il rispettivo `Match`.

Listato 4.2: Metodo **`createFromSearchResult(SearchResult result)`**

```
1   /**
     * Give a new instance of Match from a SearchResult object
     * @param result the SearchResult instance
4    * @return the new Match instance
     */
    static Match createFromSearchResult(SearchResult result) {
7     MD4Hash fileHash = new MD4Hash(result.getHash());
      String fileName = result.getName();
      long fileSize = result.getSize();
10    int fileSources = result.getSources();
      return new Match(fileHash, fileName, fileSize, fileSources);
```

```
    }

    /**
     * Constructor.
     * @param hash the MD4 hash, the FileID
     * @param name the name
     * @param size ths size
     * @param sources the initial # of sources
     */
    private Match(MD4Hash hash, String name, long size, int sources) {
      this.hash = hash;
      this.name = name;
      this.size = size;
      this.sources = sources;
      int numParts = (int)Math.ceil((double)this.size / K.PART_SIZE);
      int lastPartChunks = (int)Math.ceil((double)( this.size % K.PART_SIZE ) / K.CHUNK_SIZE);
      int partsToGetAICH = ( numParts > 1 ) ? ( numParts > 2 && lastPartChunks < 5 ) ? 3 : 2 :
           1;
      this.parts = new Part[partsToGetAICH];
      for ( int p = 0 ; p < this.parts.length - 1 ; p++ ) {
        this.parts[p] = new Part();
      }
      this.parts[this.parts.length - 1] = new Part(this.size - ( numParts - 1 ) * K.PART_SIZE)
          ;
    }
```

Si noti come nell'inizializzazione di un'istanza di `Match` si definisce quali hash vengono salvati: precisamenti quelli associati ai chunk della prima e ultima parte del file. Nel caso in cui l'ultima parte non sia composta da un numero sufficiente di chunk (5 o più), si provvede a mantenere anche gli hash dei chunk relativi alla penultima parte del file, così da ottenere un buon numero di hash per i confronti.

In `HashFinder.java` vengono implementati due thread per il reperimento degli hash *AICH*. Sebbene il sistema di reperimento degli hash *AICH* viene già implementato in `Download.java`, è stato necessario reimplementarlo. Il problema fondamentale sta nel fatto che l'utilizzo dei metodi di reperimento degli hash *AICH* di `Download` prevede la creazione di un oggetto `Download`. `Download` è una classe atta a contenere tutte le informazioni per lo scaricamento dei file e non solo al mantenimento dei soli hash *AICH*. Sebbene l'implementazione di codice che utilizzasse quei metodi sia semplice e veloce, il fatto di dover istanziare un oggetto `Download` comporterebbe il caricamento in mmoria di un gran numero di strutture dati che, per il solo fine del reperimento di qualche hash *SHA1*, non verrebbero mai utilizzati. Inoltre, alla luce del fatto che le procedure di reperimento degli hash *AICH* vengono richiamate per ogni match della rete *eMule/eDonkey* trovato, e che il numero di questi match può essere notevole, l'utilizzo di `Download.java` per reperire gli hash porterebbe ad istanziare una moltitudine di strutture inutilmente e di conseguenza ad un largo spreco di memoria. Viene così re-implementato il sistema di reperimento degli hash *AICH*, componendolo di una serie thread. Il primo, `SourcesGetter`

che si occupa di reperire le fonti disponibili per il match e, per ogni nuova fonte incontrata, lancia un thread `AICHGetter`. `AICHGetter` si occupa di collegarsi alla nuova fonte per verificare se questa supporta il sistema di controllo degli errori *AICH*. In caso affermativo il thread richiede i vari hash da reperire.

Listato 4.3: Metodo `go()` di `SourcesGetter` in `HashFinder.java`

```
    @Override
2   public void go() throws InterruptedException {
      boolean found = false;
      while (!this.mustStop()) {
5       long currentTime = System.currentTimeMillis();
        if (this.match.allPartsCompleted()) {
          this.contactedSources.clear();
8         Out.printImportant("AICH FOUND- closing thread");
          running.remove(this);
          this.thread.interrupt();
11        continue;
        }

14      if (this.match.rootHash != null && !found) {
          // Once we found rootHash we clear allknownSources to ask partHashset also to
              previously known peers.
          this.contactedSources.clear();
17        found = true;
        }

20      if (( this.contactedSources.size() >= this.match.sources || this.contactedSources.size
            () >= MAX_SOURCES ) && this.match.rootHash == null) {
          Out.printImportant("Too few Sources give us roothash to trust it- closing thread");
          this.contactedSources.clear();
23        running.remove(this);
          this.thread.interrupt();
          continue;
26

        }
        // Seeking sources
29      if (Server.weAreConnected()) {
          if (!Config.USE_GLOBAL_SOURCE_SEARCH && currentTime >= this.lastSourcesQuery +
              Config.SERVER_SOURCES_INQUIRY_INTERVAL) {
            this.lastSourcesQuery = System.currentTimeMillis();
32          this.findSources(); // asks main server for new sources
          }
          else if (Config.USE_GLOBAL_SOURCE_SEARCH && ( currentTime >= this.lastSourcesQuery +
               Config.SERVER_SOURCES_INQUIRY_INTERVAL * Config.SERVERS_QUERIED_PER_ROUND /
              Server.list.size() || this.contactedSources.size() == 0 )) {
35          this.lastSourcesQuery = System.currentTimeMillis();
            this.findSources(); // asks servers for new sources
          }
38      }
```

```
        if (Kad.isConnected() && currentTime >= this.lastKadSourcesQuery + Config.
            KAD_SOURCES_INQUIRY_INTERVAL) {
          this.lastKadSourcesQuery = System.currentTimeMillis();
41        this.findKadSources();
        }

44      MuloThread.wait(this, (int)( Config.LONG_TIMEOUT - ( System.currentTimeMillis() -
            currentTime ) ));
      }
      running.remove(this);
47  }
```

Si noti in particolare nel codice di Listato 4.3 l'utilizzo dei metodi `findSources()` e `findKadSources()`. Questi metodi si occupano di reperire le fonti del file ricercandole rispettivamente: `findSources()` da tutti i server *eMule/eDonkey* conosciuti e `findKadSources()` dalla rete *Kad*. Ogni fonte ottenuta dalle ricerche viene inserita in una lista delle fonti contattate (`contactedSources`). Se la fonte è già in questa lista, significa che è stata già contattata e non viene quindi ricontattata. I due metodi, ad ogni fonte reperita effettuano queste operazioni: vericicano se la fonte è in `contactedSources`, se si analizzano la prossima fonte, se no aggiungono la nuova fonte nella lista e la contattano lanciando un thread `AICHGetter` dedicato.

Listato 4.4: Metodo `go()` di `AICHGetter` in `HashFinder.java`

```
1   public void go() throws InterruptedException {
      int receivedHashes = 0;
      // tricky use of break like goto
4     while (true) {
        for ( int attempts = 0 ; attempts < Config.MAX_CONNECTION_TRIES ; attempts++ ) {
          if (this.peer.connectPeer()) {
7           break;
          }
        }
10      if (!this.peer.isConnected()) {
          break;
        }
13      if (!( this.peer.supportsAICH() && this.peer.supportsMultiPackets() )) {
          break;
        }
16      if (this.match.rootHash == null) {
          AICHHash currentHash = null;
          SHA1Hash hash = this.retrieveRootHash(this.peer);
19        if (hash != null) {
            List<AICHHash> tempRootHashes = this.match.temporaryReceivedRootHashes;
            synchronized (tempRootHashes) {
22            if (this.match.rootHash == null) {
                for ( AICHHash h : tempRootHashes ) {
                  if (h.hash.equals(hash)) {
25                  h.id++;
```

```
                    currentHash = h;
                  }
28                receivedHashes += h.id;
                }
                if (currentHash == null) {
31                tempRootHashes.add(new AICHHash(1, hash));
                }
                else if (receivedHashes >= Config.MIN_ROOT_HASH_SOURCES && ( (float)
                    currentHash.id / receivedHashes ) > Config.MIN_ROOT_HASH_CONSENSUS) {
34                Out.printImportant("FOUND rootHash:" + currentHash.hash);
                  // TODO we may notify MNFM we get AICH RootHash
                  synchronized (this.match) {
37                  this.match.rootHash = currentHash.hash;
                  }
                  tempRootHashes.clear();
40                // current peer had correct rootHash, we can ask partHashset without end the
                      thread.
                  continue;
                }
43            }
            }
          }
46        break;
        }
        if (!this.match.rootHash.equals(this.retrieveRootHash(this.peer))) {
49        break;
        }
        for ( int part = 0 ; part < this.match.parts.length ; part++ ) {
52        if (!this.match.parts[part].requested) {
            this.match.parts[part].requested = true;
            if (( this.match.parts[part].completed = this.retrievePartHash(this.peer, part) ))
                {
55            Out.printImportant("AICH hashet " + part + "-esima PARTE OTTENUTO [partRoothash:
                  " + this.match.parts[part].hash + "]");
              // TODO we may notify MNFM we get Part AICHHashes
              continue;
58          }
            this.match.parts[part].requested = false;
          }
61      }
        break;
      }
64    this.peer.forceDisconnect();
      MuloThread.endThread();
    }
```

Le operazioni del thread `AICHGetter` (in Listato 4.4) riproducono fedelmente le operazioni atte al reperimento degli hash *SHA1* utilizzando il sistema *AICH*. In particolare prima di poter ottenere gli hash dei vari chunk è indispensabile conoscere il *rootHash* relativo

al file. Tale hash va però reperito dai peer della rete *eMule/eDonkey*. Il *rootHash* viene allora reperito più volte, da peer della rete differenti, in modo da ottenere un certo margine di sicurezza sulla genuinità del *rootHash* ottenuto. Seguendo gli standard della rete *eMule/eDonkey* per il sistema *AICH*, per certificare la genuinità del *rootHash* si assicura che almeno 10 peer comunichino il medesimo *rootHash*. Ottenuto il *rootHash* il thread può quindi richiede al peer i vari hash *AICH* relaivi ai chunk. Si noti come la decisione del *rootHash* venga effettuata reperendo "in parallelo" i vari *rootHash* dai peer.

In `Download.java` viene invece implementato lo scaricamento dei chunk assegnati da *MNFM* a *Mulo*. Tra lo scaricamento di *Mulo* impartito da *MNFM* e lo scaricamento standard di *Mulo* si evidenziano due sostanziali differenze: quali chunk vengono scaricati e come vengono salvati. Mentre nel primo caso è *MNFM* a impartire i chunk da scaricare ed il salvatagio avviene passando i dati scaricati a *MNFM*, nella seconda modalità i chunk da scaricare sono tutti quelli di cui il file è composto ed il loro salvataggio avviene direttamente sul file. A parte queste differenze per il resto le procedure sono le medesime:

- scarica il prossimo chunk da scaricare.
- verifica il chunk scaricato.
- se il chunk è verificato, salvalo.

In `Download` vengono quindi introdotti due nuovi campi: `chunksToDownload`, una lista contenente gli indici dei chunk da scaricare e `cooperative` una variabile booleana che a seconda del suo valore modifica il flusso di esecuzione della procedura di download rendendolo un download standard o un download "cooperativo" (che scarica i chunk assegnati da *MNFM*).

Listato 4.5: Costruttore "cooperativo" di `Download.java`

```
  boolean cooperative = false;

3 LinkedList<int[]> chunksToDownload = null;

  Download(Match m) {
6   this.cooperative = true;

    this.parts = new Part[(int)Math.ceil((double)m.size / K.PART_SIZE)];
9   for ( int p = 0 ; p < this.parts.length ; p++ ) { // p: part number
      this.parts[p] = new Part(p);
    }
12  if (this.parts.length == 1) { // single part
      this.parts[0].md4Hash = m.hash;
    }
15  this.partsToBeCompleted = this.parts.length;
    this.partsToBeVerified = this.parts.length;
    this.queuingSources = new Hashtable<Ed2kID,Peer>(1);
18
    // priority chunks are decided by MNFM
```

```
        this.highPriorityChunks = null;
21      // chunks to download will be putted reading xml
        this.chunksToDownload = new LinkedList<int[]>();

24      this.enterTime = System.currentTimeMillis(); // enter queue
        DownloadManager.inactive.add(this);
        this.status = ( Config.AUTOSTART_DOWNLOADING && Server.weAreConnected() ? FileStatus.
            STARTING : FileStatus.PAUSED );
27      //DownloadManager.metadataWasChanged = true;
        MuloThread.notify(DownloadManager.class); // wake up
      }
```

In Listato 4.5, si può osservare come viene inizializzato un download cooperativo: per prima cosa viene settato `cooperative` a `true`, per poi inizializzare tutte le strutture necessarie a `Download`. Si noti come la lista `chunksToDownload` venga inizializzata come lista vuota in quanto sarà cura di *MNFM* comunicare in seguito a *Mulo* i chunk da inserire. Dopo l'avvio, il download cooperativo prosegue la sua esecuzione come fosse un comune download fino a che non arriva il momento di richiedere i chunk da scaricare ai peer della rete. Viene quindi modificato il metodo `askNextChunksTo(Peer peer)`, che si occupa di richiedere al peer dato i chunk da scaricare.

Listato 4.6: Metodo `askNextChunksTo(Peer peer)` di `Download.java`

```
1     /**
       * Looks for still unassigned chunks, assigns one or three of them to this peer (depending
            on whether he's a reliable
       * source) and sends him a request. High priority chunks are the first to be requested,
           the others come in order. If
4      * there are no unassigned chunks left, <code>false</code> is returned.
       */
      private boolean askNextChunksTo(Peer source) {
7       if (this.allChunksAssigned) {
          return false;
        }
10      Part.Chunk chunk;
        long[] offsets = new long[6];
        // we ask three chunks if the source is reliable, only one otherwise
13      int wanted = ( source.isReliableSource() ? 3 : 1 );
        int found = 0;
        boolean skippedSomething = false;
16
        // If this is a Cooperative Download launched by metacoso we have to download only the
            priority chunks!
        if (this.cooperative) {
19        for ( int[] chunkAddress : this.chunksToDownload ) {
            if (source.downloadSession.partAvailability != null && !source.downloadSession.
                partAvailability[chunkAddress[0]]) { // he doesn't have this part
              skippedSomething = true;
22            continue;
```

```
        }
        chunk = this.parts[chunkAddress[0]].chunks[chunkAddress[1]];
        if (chunk.status == ChunkStatus.UNASSIGNED) { // ok, add this one
          offsets[2 * found] = chunk.fileOffset + chunk.bytesReceived;
          offsets[2 * found + 1] = chunk.fileOffset + chunk.size;
          chunk.assignTo(source);
          if (++found == wanted) { // enough
            return source.send(new PacketTCPFileDataRequest(this.file.md4Hash, offsets));
          }
        }
      }
    }
    else {
      // First look at preferred chunks
      if (this.highPriorityChunks != null) {
        for ( int[] chunkAddress : this.highPriorityChunks ) {
          if (source.downloadSession.partAvailability != null && !source.downloadSession.
              partAvailability[chunkAddress[0]]) { // he doesn't have this part
            skippedSomething = true;
            continue;
          }
          chunk = this.parts[chunkAddress[0]].chunks[chunkAddress[1]];
          if (chunk.status == ChunkStatus.UNASSIGNED) { // ok, add this one
            offsets[2 * found] = chunk.fileOffset + chunk.bytesReceived;
            offsets[2 * found + 1] = chunk.fileOffset + chunk.size;
            chunk.assignTo(source);
            if (++found == wanted) { // enough
              return source.send(new PacketTCPFileDataRequest(this.file.md4Hash, offsets));
            }
          }
        }
      }

      // Now look at all chunks (including already watched)
      for ( int p = 0 ; p < this.parts.length ; p++ ) { // p: part number
        if (source.downloadSession.partAvailability != null && !source.downloadSession.
            partAvailability[p]) { // he doesn't have this part
          skippedSomething = true;
          continue;
        }
        Part part = this.parts[p];
        if (part.status != PartStatus.INCOMPLETE) { // nothing to ask here
          continue;
        }
        for ( int c = 0 ; c < part.chunks.length ; c++ ) { // c: chunk number
          chunk = part.chunks[c];
          if (chunk.status == ChunkStatus.UNASSIGNED) { // ok, add this one
            offsets[2 * found] = chunk.fileOffset + chunk.bytesReceived;
            offsets[2 * found + 1] = chunk.fileOffset + chunk.size;
```

```
70              chunk.assignTo(source);
                if (++found == wanted) { // enough
                  return source.send(new PacketTCPFileDataRequest(this.file.md4Hash, offsets));
73              }
              }
            }
76        }
        }
        // Nothing left to be asked
79      this.allChunksAssigned = !skippedSomething;
        if (found > 0) {
          return source.send(new PacketTCPFileDataRequest(this.file.md4Hash, offsets));
82      }
        return false; // nothing found
      }
```

Il codice in Listato 4.6 può essere riassunto in un semplice procedimento:

- Se il download è cooperativo (`cooperative = true`), allora procedi a richiedere al peer dei chunk della lista `chunksToDownload`.
- Altrimenti (`cooperative = false`) procedi a richiedere al peer dei chunk scelti tra quelli che compongono l'intero file.

Altro aspetto da curare nell'esecuzione sta nel salvataggio dei dati scaricati, viene a questo scopo modificato il metodo di salvataggio di un chunk (`chunkSave()`) in modo che se richiamato da un download di tipo cooperativo i dati da salvare vengano comunicati a *MNFM* anziché essere salvati direttamente su file.

Listato 4.7: Metodo `chunkSave()` di `Chunk` in `Download.java`

```
        /**
         * Tries to store this chunk data on disk, and deletes data from memory only if
             COMPLETED.<br>
3        * On success: COMPLETED => SAVED (not verified), VERIFIED => VERIFIED_SAVED, any
             other status is kept.<br>
         * On failure: status is kept.
         * @return Whether the file on disk was successfully updated.
6        */
        private boolean chunkSave() {
          if (this.status == ChunkStatus.SAVED || this.bytesSaved >= this.bytesReceived) {
9           return true;
          }
          try {
12          // if there is a cooperative download launched by MNFM it don't store the chunk to
                a file but it have to
            // pass the chunk's byte to MNFM.
            if (Download.this.cooperative) {
15            // MNFM.storeData(this.data,this.fileOffset);
```

```
            }
            else {
              Download.this.file.storeData(this.data, this.fileOffset);
            }
            if (this.status == ChunkStatus.COMPLETED) {
              this.status = ChunkStatus.SAVED;
              this.data = null; // free memory
            }
            else if (this.status == ChunkStatus.VERIFIED) {
              this.status = ChunkStatus.VERIFIED_SAVED;
              this.data = null; // free memory
            }
            this.bytesSaved = this.bytesReceived;
            DownloadManager.metadataWasChanged = true;
            return true;
          }
          catch (IOException e) {
            // If it won't ever be saved, part check will fail and AICH or ICH will eventually
                 find this corrupt chunk
            return false;
          }
        }
```

Per concludere, in `MuloMNFMInteface.java` sono stati implementati tutti i metodi con cui *MNFM* può istruire *Mulo* sul da farsi.

`List<Match> searchWithMulo(String name, long size)`

Restituisce, sfruttando nome e dimensione del file da scaricare, una lista di possibili `Match` sulla rete *eMule/eDonkey*. In particolare viene effettuata una ricerca nella rete coi metodi offerti dal plugin *Mulo* ottenendo una lista di `SearchResult`, per ognuno dei quali viene creato il rispettivo `Match` attraverso il metodo `createFromSearchResult(SearchResult result)`.

Listato 4.8: Metodo `searchWithMulo()`

```
    /**
     * Return a list of Match with given name and size
     * @param name
     * @param size
     * @return the match list
     */
    public static List<Match> searchWithMulo(String name, long size) {
      Search search = new Search(Search.Type.ADVANCED, name); // strip first whitespace
      search.min = size / K.MB;
      search.max = size / K.MB;
      search.start();
      LinkedList<Match> matchList = new LinkedList<Match>();
      for ( SearchResult result : search.results.values() ) {
```

```
        matchList.add(Match.createFromSearchResult(result));
    }
    return matchList;
}
```

void getAICHHashWithMulo(Match match)

Passando come parametro il match di cui reperire gli hash SHA1, viene eseguito il thread `SourcesGetter`, che al termine del suo compito carica nel match gli hash reperiti.

Listato 4.9: Metodo `getAICHHashWithMulo()`

```
/**
 * Retrieve AICH for the given match using custom duplicated code of Download/
     DownloadManager
 * @param match
 */
public static void getAICHHashWithMulo(Match match) {
  new SourcesGetter(match).start();
}
```

void startDownloadWithMulo(Match match)

Comincia il `Download` "cooperativo" di un `Match`.

Listato 4.10: Metodo `startDownloadWithMulo()`

```
/**
 * Start Download the given match.
 * @param match
 */
public static void startDownloadWithMulo(Match match) {
  match.download = new Download(match); // create cooperative Download relative to Match
  updateDownloadInfo(match); // get chunksToDownload from the xml
}
```

void updateDownloadInfo(Match match)

Viene richiamato da *MNFM* ogni qual volta gli assegnamenti dei chunk ai moduli *Mulo* e *Torrent* relativi ad un certo download vengono modificati. Il metodo scansiona il file *XML* reperendo le informazioni sull'assegnamento dei chunk associate al download e da queste aggiunge i chunk assegnati a *Mulo* nella lista `chunksToDownload`.

Listato 4.11: Metodo `updateDownloadInfo()`

```
/**
```

```
     * MNFM tells Mulo that Data information of the downloading match are changed, so Mulo
         have to re-check the xml.
     * @param match
     */
    public static void updateDownloadInfo(Match match) {
      try {
        XMLReader reader = XMLReaderFactory.createXMLReader();
        reader.setContentHandler(new SuperDownloadListXMLHandler(match.download, reader));
        reader.parse(new InputSource(Mulo.getFileRead(Config.MULO_DOWNLOAD_LIST)));
        return;
      }
      catch (MalformedURLException e) { // getFileRead may return null
        PPLog.printWarning("Super Download list XML not found!!!");
      }
      catch (Exception e) { // SAX, IllegalArgument, NullPointer, etc.
        PPLog.printError("Malformed download list XML, renaming to \"" + Config.
            MULO_DOWNLOAD_LIST + ".old\" and creating a new one", e);
      }
      // Error
    }
```

4.4 Situazione e sviluppi futuri

Allo stato attuale l'implementazione di *MNFM* non è ancora completamente sviluppata. Sebbene gran parte delle funzioni di interfacciamento a *Torrent* e la totalità delle funzioni di interfacciamento a *Mulo* siano state implementate e testate, rimangono ancora da ultimare tutte le funzioni di *MNFM*. Queste funzioni formano il fulcro di tutto il sistema e la loro implementazione determina il funzionamento di *SuperFilesharing*. A parte le funzioni dell'assegnamento dei chunk gran parte delle funzioni in *MNFM* prendono già forma, analizzando il funzionamento di *SuperFilesharing*. Dall'osservazione emerge come *MNFM* necessita di strutture appropriate sulle quali mantenere gli stati dei chunk, informazioni essenziali per capire a che punto si trova la procedura di download. Per esempio è osservabile che quando tutti i chunk raggiungano lo stato `VERIFIED` il download è completato ed il file viene salvato correttamente. L'idea è che la struttura che mantenga le informazioni sui chunk possa facilitare le operazioni di *MNFM*, prime fra tutte quelle riguardanti la verifica dei piece. Si vuol cioè che la struttura renda semplice capire quando i chunk relativi ad un dato piece sono stati scaricati (stato `UNVERIFIED`).

Si pensa quindi di definire una classe `SuperFile` che al suo interno contiene tutte le informazioni del file da scaricare e principalmente due liste: `parts`, che rappresenta la lista delle parti di cui è composto il file, e `pieces`, che rappresenta la lista dei piece in cui è suddiviso il file. Ogni parte viene rappresentata da un oggetto `Part` che mantiene tutte le informazioni necessarie sulla parte come il suo indice, i suoi offset e `chunks` la lista dei chunk che compongono la parte. Ogni piece viene rappresentato da un oggetto `Piece` che ne mantiene le informazioni necessarie tra le quali `relativeChunks`, una

lista che mantiene al suo interno dei riferimenti ai chunk che corrispondono al piece. In questo modo preso un piece, risulta semplice a *MNFM* controllare se è possibile verificarlo: basta infatti che i chunk a cui fa riferimento la lista `relativeChunks` siano tutti nello stato `UNVERIFIED`. Ogni chunk viene infatti rappresentato da un oggetto `Chunk` che ne mantiene tutte le informazioni necessarie come indici, offset e stato.

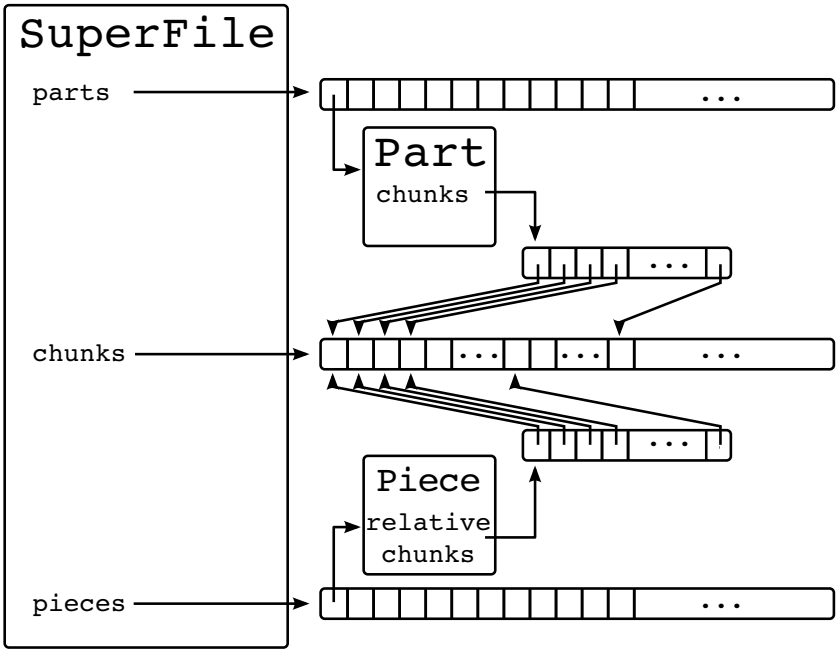


Figura 4.10: `SuperFile`, `Part`, `Piece` e `Chunk`: schema delle relazioni.

L'implementazione di queste strutture porterebbe a rendere più semplice le procedure da effettuare durante la verifica dei piece. Se per esempio un certo `Piece p` viene verificato, la procedura prevede che tutti i chunk ad esso relativo vengano messi in stato `VERIFIED`. Con l'utilizzo della lista `relativeChunks` questa operazione risulta semplificata in quanto è sufficiente iterare i chunk della lista e settare il loro stato, effettuando gli opportuni controlli per i chunk limitrofi. In `relativeChunks` vengono inseriti tutti i chunk relativi ad un dato piece e pertanto i chunk limitrofi, trovandosi a cavallo di due piece, compaiono in entrambe le liste `relativeChunks` dei due piece. È bene osservare che un chunk è considerato limitrofo quando è verificata una delle seguenti:

- il suo *offset iniziale* è inferiore all'*offset iniziale* del piece a cui fa riferimento.
- il suo *offset finale* è superiore all'*offset finale* del piece a cui fa riferimento.

Pertanto in un `Piece p` potranno esserci al massimo 2 chunk limitrofi che occuperanno il primo e ultimo posto della lista `relativeChunks`. Lo stato dei chunk limitrofi viene quindi posto a `VERIFIED` solo se lo stato dei chunk tra cui è compreso (chunk precedente e chunk successivo) risulta anch'esso `VERIFIED`. Questo controllo risulta possibile solo per

merito del modo in cui vengono definite le strutture in quanto da `p.relativeChunks` si ottiene il chunk limitrofo iniziale (`Chunk c = p.relativeChunks.get(0)`) e finale (`Chunk c = p.relativeChunks.get(p.relativeChunks.size()-1)`) dai quali, tramite il loro indice, è possibile ricavare i chunk a loro consecutivi.

Listato 4.12: Aggiornamento degli stati in seguito alla verifica di un piece

```
if(verifyPiece(piece)){
  for(chunk : piece.relativeChunks){
    chunk.status = V;
  }
  startindex = piece.relativeChunks[0].index
  if(piece.startOffset > chunks[startindex].startoffset && chunks[startindex-1].status !=
      V ){
    chunks[startindex].status = U
  }
  endindex = piece.relativeChunks[relativeChunks.size() - 1].index
  if(piece.endOffset < chunks[endindex].endoffset && chunks[endindex+1].status != V ){
    chunks[startindex].status = U
  }
}
else{
  muloFails++;
  if(muloFails < MAXFAILS){
    for(chunk : piece.relativeChunks){
      chunk.status = (chunk.status == U)?E:chunk.status;
    }
    assignChunks();
    Mulo.updateDownloadInfo();
    Torrent.updateDownloadInfo();
  }
  else{
    Mulo.stopDownload(match)
    for(chunk : piece.relativeChunks){
      chunk.status = (chunk.status == U)?T:chunk.status;
    }
    Torrent.updateDownloadInfo();
  }
}
```

Oltre a queste idee in procinto di essere sviluppate in *MNFM*, grandi interrogativi si hanno per la metodologia da adottare per l'assegnamento dei chunk ai due moduli. Dai vari test effettuati sorge poi una certa fragilità del sistema nella metodologia adottata per l'identificazione del file della rete *Torrent* (fornito dal file *.torrent*) nella rete *eMule/eDonkey*. Lo sfruttamento degli hash *SHA1* dei chunk, utilizzati da *AICH*, può non sempre essere possibile. Questo problema è legato alla procedura di *AICH* per il reperimento del

rootHash, tramite il quale sono poi ottenibili i vari hash dei chunk. Per ottenere un *rootHash* affidabile è necessario che almeno 10 peer della rete *eMule/eDonkey* che supportano *AICH* comunichino lo stesso *rootHash*.

Nel caso migliore pertanto, per sfruttare gli hash *AICH* in *SuperFilesharing*, è necessario che almeno 10 peer della rete che possiedono il match di cui si vogliono reperire gli hash, supportino *AICH*. Questa condizione esclude automaticamente la possibilità di reperire gli hash per tutti quei match che sono condivisi da meno di 10 peer in quanto anche se uno di questi match corrispondesse al file, non ottenendo i suoi hash, *MNFM* non può verificarne la corrispondenza. Si ricorda comunque che il problema è legato solo ai file più nuovi della rete e veramente poco diffusi, pertanto basta attendere che la diffusione del file nella rete aumenti affinche più utenti che supportino *AICH* lo condividano dai quali sarà possibile reperire gli hash tanto agognati.

Appendici

Appendice A

Utilizzo dei Thread in *PariPari*

Listato A.1: Esempio di thread in *Mulo*

```
class SampleThread extends PariPariRunnable {
  private static PariPariThread thread;
  static sunchronized void start() throws IllegalStateException {
    if(thread!=null){
      throw new IllegalStateException("Thread attualmente in esecuzione");
    }
    thread = MuloThread.startThread(new ThreadSample(),"SampleThread");
  }
  @Override
  @Deprecated
  public void go(){
    try{
      while(!this.mustStop()){
        // operazioni eseguite dal thread
      }
      Log.printNotice("SampleThread interrotto");
    }
    catch(Exception e){
      Log.printError("Exception in SampleThread",e);
    }
    finally{
      MuloThread.endThread();
      thread = null;
    }
  }
}
```

Appendice B

Codice

B.1 Hashing Parallelo

Shared.java

Listato B.1: `realAdd(File file, boolean recursive)` in `Shared.java`

```
164   /**
       * Performs the real, recursive (for folders) addition.
       * @param file Reference to the file or directory (*must* exist).
167    * @param recursive Whether to recur into subfolders, if this is a folder.
       * @return Whether file/directory was not already shared and the shared files XML has been
           successfully updated.
       */
170   private static synchronized boolean realAdd(File file, boolean recursive) {
        if (file.getAbsolutePath().startsWith(Config.absoluteTempDir)) {
          return false;
173      }
        if (file.isFile()) {
          long size = file.length();
176       if (size > K.MAX_FILE_SIZE) {
            Log.printWarning("The file is too large to be shared", file.getAbsolutePath());
            return false;
179       }
          if (file.isHidden()) { // skip hidden files
            return false;
182       }
          Shared newFile = filesList.get(file.getAbsolutePath());
          if (newFile != null) { // already in
185         return false;
          }
          if (Runtime.getRuntime().availableProcessors() == 1 && Runtime.getRuntime().freeMemory
              () >= file.length()) {
188         // parallelism is futile, hashing with the standard methods
            MD4Hash[] fileMD4Hashes = MD4Hash.calculate(file);
```

```
            newFile = new Shared(file.getAbsolutePath(), size, fileMD4Hashes[0], null, file.
                lastModified());
191         SHA1Hash[] fileSHA1Hashes = SHA1Hash.calculate(newFile);
            newFile.sha1Hash = fileSHA1Hashes[0];
            filesList.put(newFile.path, newFile);
194         hashesList.put(newFile.md4Hash, newFile);
            if (Server.weAreConnected()) {
              Server.currentServer.filesToBeOffered.add(newFile);
197         }
            for ( int p = 0 ; p < newFile.parts.length ; p++ ) { // p: part number
              newFile.parts[p] = newFile.new Part(p, fileMD4Hashes[p + 1], fileSHA1Hashes[p +
                  1]);
200           newFile.parts[p].setAvailable();
            }
          }
203       else {
            // parallel hashing
            newFile = new Shared(file.getAbsolutePath(), size, file.lastModified());
206         HashManager.calculateHash(newFile);
            filesList.put(newFile.path, newFile);
            hashesList.put(newFile.md4Hash, newFile);
209         if (Server.weAreConnected()) {
              Server.currentServer.filesToBeOffered.add(newFile);
            }
212       }
        }
        else { // directory, we also add the files within
215       Shared newDir = dirsList.get(file.getAbsolutePath());
          if (newDir != null) { // already in
            return false;
218       }
          newDir = new Shared(file.getAbsolutePath());
          dirsList.put(newDir.path, newDir);
221       if (recursive) { // only if recursive we add the directory itself
            dirsList.put(newDir.path, newDir);
          }
224       File[] dirFiles = file.listFiles();
          if (dirFiles == null) { // we have no access to this directory
            Log.printWarning("Could not access the directory", file.getAbsolutePath());
227         return false;
          }
          totalItemsToBeAdded += dirFiles.length;
230       for ( int i = 0 ; i < dirFiles.length ; i++ ) {
            if (dirFiles[i].isFile()) {
              Out.print("Item " + ++itemsAdded + " of " + totalItemsToBeAdded + ": " + dirFiles[
                  i].getAbsolutePath() + " (" + Utils.toKiB(dirFiles[i].length()) + ")");
233           realAdd(dirFiles[i], false); // recursion
            }
            else if (recursive) { // directory, recursive
```

```
236             Out.print("Item " + ++itemsAdded + " of " + totalItemsToBeAdded + ": " + dirFiles[
                    i].getAbsolutePath() + K.PATH_SEPARATOR + " (subfolder, indexing)");
                realAdd(dirFiles[i], true); // recursion
              }
239           else { // directory, not recursive
                Out.print("Item " + ++itemsAdded + " of " + totalItemsToBeAdded + ": " + dirFiles[
                    i].getAbsolutePath() + K.PATH_SEPARATOR + " (subfolder, skipping)");
              }
242         }
          }
          return true;
245     }
```

HashManager.java

Listato B.2: Sorgente di **HashManager.java**

```
1  package paripari.mulo;

   /**
4   * @author Stefano Pelizzaro
    */

7  import java.io.IOException;
   import java.nio.ByteBuffer;

10 /**
    * Class to manage the computing of SHA1 and MD4 hashes
    */
13 public class HashManager {

     /**
16    * Max number of attempts to read a file part
      */
     private static final int MAX_ATTEMPS = 3;
19   /**
      * Link to MD4Hasher thread
      */
22   private static MD4Hasher md4;

     /**
25    * Link to SHA1Hasher thread
      */
     private static SHA1Hasher sha1;
28
     /**
      * Data Buffer
31    */
     private static byte[] buffer;

34   /**
      * Calculate MD4 and SHA1 hashes for the given file
      * @param file the Shared file
37    */
     static synchronized void calculateHash(Shared file) {
       if (file.fileHashset == null) {
40       file.fileHashset = new AICHHashset(file);
       }
       Log.printNotice("MD4 & SHA1 hashing \"" + file.path + "\"");
43     AICHNode[] partRoots = file.fileHashset.getLeaves();
       AICHHashset partHashset;
       ByteBuffer concatenatedMD4Hashes;
46     int bytesToRead = (int)K.PART_SIZE;
```

```
        long offset = 0;
        try {
49        // Getting number of file parts
          int numParts = file.parts.length;
          concatenatedMD4Hashes = ByteBuffer.allocate(MD4Hash.SIZE * numParts);
52        partHashset = file.fileHashset.addPartHashset(file, 0);

          // Loading bufferA...
55        if (numParts == 1) {
            bytesToRead = (int)file.size;
          }
58        for ( int attempts = 1 ; attempts <= MAX_ATTEMPS ; attempts++ ) {
            try {
              buffer = file.getData(offset, bytesToRead);
61          }
            catch (IOException e) {
              if (attempts == MAX_ATTEMPS) {
64              throw new IOException("Cannot read File.");
              }
              MuloThread.sleep(1000);
67            continue;
            }
            break;
70        }
          offset += (int)K.PART_SIZE;

73        // Initializing SHA1Hasher & MD4Hasher.
          sha1 = new SHA1Hasher(buffer, partHashset);
          md4 = new MD4Hasher(buffer);
76
          // Starting SHA1Hasher...
          sha1.start();
79
          // Starting MD4Hasher...
          md4.start();
82
          for ( int part = 1 ; part < numParts ; part++ ) {

85          partHashset = file.fileHashset.addPartHashset(file, part);

            // Loading bufferB...
88
            if (part + 1 == numParts) {
              bytesToRead = (int)( file.size % bytesToRead );
91          }
            for ( int attempts = 1 ; attempts <= MAX_ATTEMPS ; attempts++ ) {
              try {
94              buffer = file.getData(offset, bytesToRead);
              }
```

```
            catch (IOException e) {
97            if (attempts == MAX_ATTEMPS) {
                throw new IOException("Cannot read File.");
              }
100           MuloThread.sleep(1000);
              continue;
            }
103         break;
          }
          offset += (int)K.PART_SIZE;
106
          // Waiting SHA1Hasher...
          sha1.waitDigest();
109
          // Waiting MD4Hasher...
          md4.waitDigest();
112
          // Saving Hash
          file.parts[part - 1] = file.new Part(part - 1, md4.getDigest(), sha1.getDigest());
115       partRoots[part - 1].data.hash = sha1.getDigest();
          file.parts[part - 1].setAvailable();
          concatenatedMD4Hashes.put(md4.getDigest().bytes);
118
          // Switching Buffers...
          md4.update(buffer);
121       sha1.update(buffer, partHashset);

          // Notify Hasher Threads.
124       sha1.notifyBuffer();
          md4.notifyBuffer();
        }
127     // Waiting SHA1Hasher...
        sha1.waitDigest();

130     // Waiting MD4Hasher...
        md4.waitDigest();

133     file.parts[numParts - 1] = file.new Part(numParts - 1, md4.getDigest(), sha1.getDigest
            ());
        partRoots[numParts - 1].data.hash = sha1.getDigest();
        file.parts[numParts - 1].setAvailable();
136     concatenatedMD4Hashes.put(md4.getDigest().bytes);

        // We have SHA1 and MD4 hashes of all parts,
139     // let's calculate File's SHA1 and MD4 Hashes...
        if (numParts == 1) {
          sha1.update(null, file.fileHashset);
142       sha1.notifyBuffer();
          sha1.waitDigest();
```

```
        }
145     else {
          md4.update(concatenatedMD4Hashes.array());
          sha1.update(null, file.fileHashset);
148
          sha1.notifyBuffer();
          md4.notifyBuffer();
151       sha1.waitDigest();
          md4.waitDigest();
        }
154     // It's Done!!!
        file.md4Hash = md4.getDigest();
        file.sha1Hash = sha1.getDigest();
157     Out.printDebug("Hash for file : MD4-" + md4.getDigest() + " SHA1-" + sha1.getDigest())
            ;

        // Stopping Threads.
160     sha1.stop();
        md4.stop();
        sha1.notifyBuffer();
163     md4.notifyBuffer();
      }
      catch (IOException e) {
166     Log.printError("Error reading file " + file.path, e);
        sha1.stop();
        md4.stop();
169     sha1.notifyBuffer();
        md4.notifyBuffer();
        // calculateHash(file);
172   }
    }
  }
```

MD4Hasher.java

Listato B.3: Sorgente di **MD4Hasher.java**

```
package paripari.mulo;

/**
 * @author Stefano Pelizzaro
 */

import paripari.core.PariPariRunnable;
import paripari.core.PariPariThread;

/**
 * A thread that calculates the MD4 hash of a Message/Data.
 */
class MD4Hasher extends PariPariRunnable {

  /** MD4Hasher thread, or <code>null</code> if not running. */
  private PariPariThread thread;

  /**
   * The Message/Data to be hashed.
   */
  private byte[] buffer;

  /**
   * The calculated MD4Hash.
   */
  private MD4Hash digest;

  /**
   * A synchronizer for <code>byte[] buffer</code> operations.
   */
  private final Object bufferSemaphore;

  /**
   * A synchronizer for <code>MD4Hash digest</code> operations.
   */
  private final Object digestSemaphore;

  /**
   * Tell when <code>byte[] buffer</code> can be hashed.
   */
  private boolean bufferReady;

  /**
   * Tell when <code>MD4Hash digest</code> has been calculated.
   */
  private boolean digestReady;
```

```
48    /**
       * Private constructor so it's not callable from outside.
       * @param buffer the Message/Data to be hashed.
51     */
      MD4Hasher(byte[] buffer) {
        this.buffer = buffer;
54      this.digestReady = false;
        this.bufferReady = false;
        this.bufferSemaphore = new Object();
57      this.digestSemaphore = new Object();
      }

60    /**
       * Starts the MD4Hasher thread.
       * @throws IllegalStateException If the hasher thread is already running.
63     */
      synchronized void start() throws IllegalStateException {
        if (this.thread != null) {
66        throw new IllegalStateException("MD4Hasher is already running");
        }
        this.thread = MuloThread.startThread(this, Mulo.MULO + "-MD4Hasher");
69    }

      /**
72     * Stops the MD4Hasher thread, if it is running.<br>
       * Note that it may take a few seconds before the threads are actually shut.
       */
75    void stop() {
        if (this.thread == null) {
          return;
78      }
        this.thread.interrupt();
      }
81
      /**
       * MD4Hasher Thread main loop: Cyclically calculate MD4Hash of the current Message/Data in
            <code>byte[] buffer</code>
84     * and put it in <code>MD4Hash digest</code>,so it notify digest is ready and wait for a
           new Message/Data to hash.
       */
      @Override
87    @Deprecated
      public void go() {
        Thread.currentThread().setName(Mulo.MULO + "-MD4Hasher"); // XXX Core bug
90      try {
          // N.B. in the first iteration we don't need to wait because
          // the buffer is already loaded...
93        while (!this.mustStop()) {
```

```
        // some work to do...
96      this.digest = MD4Hash.calculate(this.buffer);

        this.notifyDigest();
99
        this.waitBuffer();

102   } // outer loop
      Log.printNotice("MD4Hasher thread interrupted, quitting");
      this.thread = null;
105   MuloThread.endThread();
    }
    catch (Exception e) { // not supposed to happen
108   Log.printError("Exception in MD4Hasher thread, autorestarting it", e);
      this.thread = MuloThread.startThread(this, Mulo.MULO + "-MD4Hasher");
    }
111 }

  /**
114  * Getter Method for <code>MD4Hash digest</code>
   * @return the calculated MD4Hash of the current Message/Data.
   */
117 MD4Hash getDigest() {
    return this.digest;
  }
120
  /**
   * Set the new Message/Data to be hashed.
123  * @param buffer the Message/Data.
   */
  void update(byte[] buffer) {
126   this.buffer = buffer;
  }

129 /**
   * Suspend the calling Thread until <code>notifyBuffer()</code> is called.
   */
132 void waitBuffer() {
    if (!this.bufferReady) {
      MuloThread.wait(this.bufferSemaphore);
135   }
    this.bufferReady = false;
  }
138
  /**
   * Wake up a Thread suspended on <code>waitBuffer()</code>.
141  */
  void notifyBuffer() {
```

```
        this.bufferReady = true;
144     MuloThread.notify(this.bufferSemaphore);
      }

147   /**
       * Suspend the calling Thread until <code>notifyDigest()</code> is called.
       */
150   void waitDigest() {
        if (!this.digestReady) {
          MuloThread.wait(this.digestSemaphore);
153     }
        this.digestReady = false;
      }
156
      /**
       * Wake up a Thread suspended on <code>waitDigest()</code>.
159    */
      void notifyDigest() {
        this.digestReady = true;
162     MuloThread.notify(this.digestSemaphore);
      }

165 }
```

SHA1Hasher.java

Listato B.4: Sorgente di **SHA1Hasher.java**

```
package paripari.mulo;

/**
 * @author Stefano Pelizzaro
 */

import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import paripari.core.PariPariRunnable;
import paripari.core.PariPariThread;

/**
 * A thread that calculates the SHA1 hash of a Message/Data.
 */
class SHA1Hasher extends PariPariRunnable {

  /** SHA1Hasher thread, or <code>null</code> if not running. */
  private PariPariThread thread;

  /**
   * The Message/Data to be hashed.
   */
  private byte[] buffer;

  /**
   * AICHHashset of the File Part (used for SHA1Hash computing).
   */
  private AICHHashset partHashset;

  /**
   * The calculated SHA1Hash.
   */
  private SHA1Hash digest;

  /**
   * A synchronizer for <code>byte[] buffer</code> operations.
   */
  private final Object bufferSemaphore;

  /**
   * A synchronizer for <code>MD4Hash digest</code> operations.
   */
  private final Object digestSemaphore;

  /**
   * Tell when <code>byte[] buffer</code> can be hashed.
```

```
     */
48   private boolean bufferReady;

     /**
51    * Tell when <code>MD4Hash digest</code> has been calculated.
     */
     private boolean digestReady;
54
     /**
     * Private constructor so it's not callable from outside.
57   * @param buffer the Message/Data to be hashed.
     * @param partHashset AICHHashset of File Part buffer refers to.
     */
60   SHA1Hasher(byte[] buffer, AICHHashset partHashset) {
       this.buffer = buffer;
       this.partHashset = partHashset;
63     this.digestReady = false;
       this.bufferReady = false;
       this.bufferSemaphore = new Object();
66     this.digestSemaphore = new Object();
     }

69   /**
     * Starts the unique SHA1Hasher thread.
     * @throws IllegalStateException If the hasher thread is already running.
72   */
     synchronized void start() throws IllegalStateException {
       if (this.thread != null) {
75       throw new IllegalStateException("SHA1Hasher is already running");
       }
       this.thread = MuloThread.startThread(this, Mulo.MULO + "-SHA1Hasher");
78   }

     /**
81    * Stops the SHA1Hasher thread, if it is running.<br>
     * Note that it may take a few seconds before the threads are actually shut.
     */
84   void stop() {
       if (this.thread == null) {
         return;
87     }
       this.thread.interrupt();
     }
90
     /**
     * SHA1Hasher Thread main loop: Cyclically calculate SHA1Hash of the current Message/Data
         in
93   * <code>byte[] buffer</code> and put it in <code>SHA1Hash digest</code>,so it notify
         digest is ready and wait for a
```

```
     * new Message/Data to hash.
     */
96   @Override
     @Deprecated
     public void go() {
99     Thread.currentThread().setName(Mulo.MULO + "-SHA1Hasher"); // XXX Core bug
       try {
         // N.B. in the first iteration we don't need to wait because
102      // the buffer is already loaded...
         while (!this.mustStop()) {

105        // some work to do...
           if (this.buffer != null) {
             this.digest = this.calculateSHA1(this.buffer);
108        }
           else {
             this.digest = this.partHashset.getRootHash();
111        }

           this.notifyDigest();
114
           this.waitBuffer();
         }
117      Log.printNotice("SHA1Hasher thread interrupted, quitting");
         this.thread = null;
         MuloThread.endThread();
120    }
       catch (Exception e) {
         Log.printError("Exception in SHA1Hasher thread, autorestarting it", e);
123      this.thread = MuloThread.startThread(this, Mulo.MULO + "-SHA1Hasher");
       }
     }
126
     /**
      * Computes the SHA1 hash of a part.
129   * @param buffer part to be hashed.
      * @return The SHA1 hash.
      */
132  private SHA1Hash calculateSHA1(byte[] buffer) {
       AICHNode[] leaves = this.partHashset.getLeaves();
       int partLength = buffer.length;
135    int numChunks = (int)( Math.ceil((float)partLength / K.CHUNK_SIZE) );
       for ( int chunkIndex = 0 ; chunkIndex < numChunks ; chunkIndex++ ) {
         int chunkSize = ( ( chunkIndex + 1 ) * K.CHUNK_SIZE < partLength ? K.CHUNK_SIZE :
             partLength - chunkIndex * K.CHUNK_SIZE );
138      leaves[chunkIndex].data.hash = this.calculate(buffer, chunkIndex * K.CHUNK_SIZE,
             chunkSize);
       }
       return this.partHashset.getRootHash();
```

```
141     }

        /**
144      * Computes the SHA1 hash of a chunk.
         * @param message chunk to be hashed.
         * @param offset start index of data to hash.
147      * @param length size of data to hash.
         * @return The SHA1 hash.
         */
150     SHA1Hash calculate(byte[] message, int offset, int length) {
          try {
            MessageDigest hashCalculator = MessageDigest.getInstance("SHA-1");
153         hashCalculator.update(message, offset, length);
            return new SHA1Hash(hashCalculator.digest());
          }
156       catch (NoSuchAlgorithmException e) { // shall not happen, but must catch
            throw new AssertionError("SHA-1 algorithm not available");
          }
159     }

        /**
162      * Getter Method for <code>SHA1Hash digest</code>
         * @return the calculated SHA1Hash of the current Message/Data.
         */
165     SHA1Hash getDigest() {
          return this.digest;
        }
168
        /**
         * Set the new Message/Data to be hashed and the relative AICHHashset.
171      * @param buffer the Message/Data or null to merge SHA1 hashes saved in <code>partHashset
             </code>.
         * @param partHashset the AICHHashset.
         */
174     void update(byte[] buffer, AICHHashset partHashset) {
          this.buffer = buffer;
          this.partHashset=partHashset;
177     }

        /**
180      * Suspend the calling Thread until <code>notifyBuffer()</code> is called.
         */
        void waitBuffer() {
183       while (!this.bufferReady) {
            MuloThread.wait(this.bufferSemaphore);
          }
186       this.bufferReady = false;
        }
```

```
189     /**
         * Wake up a Thread suspended on <code>waitBuffer()</code>.
         */
192     void notifyBuffer() {
          this.bufferReady = true;
          MuloThread.notify(this.bufferSemaphore);
195     }

        /**
198      * Suspend the calling Thread until <code>notifyDigest()</code> is called.
         */
        void waitDigest() {
201       while (!this.digestReady) {
            MuloThread.wait(this.digestSemaphore);
          }
204       this.digestReady = false;
        }

207     /**
         * Wake up a Thread suspended on <code>waitDigest()</code>.
         */
210     void notifyDigest() {
          this.digestReady = true;
          MuloThread.notify(this.digestSemaphore);
213     }

    }
```

SharedAdder.java

Listato B.5: Sorgente di **SharedAdder.java**

```
1  package paripari.mulo;

   /**
4   * @author Stefano Pelizzaro
    */

7  import paripari.core.PariPariRunnable;
   import paripari.core.PariPariThread;

10 /**
    * A Thread to add file to the Shared list
    */
13 class SharedAdder extends PariPariRunnable {

     /** SharedAdder thread, or <code>null</code> if not running. */
16   private static PariPariThread thread;

     /**
19    * Path of file to add
      */
     private final String path;
22
     /**
      * true if file is a directory, false otherwise
25    */
     private final boolean recursive;

28   /**
      * Constructor for SharedAdder Thread
      * @param path the file
31    * @param recursive true if file is a directory, false otherwise
      */
     public SharedAdder(String path, boolean recursive) {
34     this.path = path;
       this.recursive = recursive;
     }
37
     /**
      * Starts the SharedAdder thread.
40    * @throws IllegalStateException If the hasher thread is already running.
      */
     synchronized void start() throws IllegalStateException {
43     if (SharedAdder.thread != null) {
         throw new IllegalStateException("SharedAdder is already running");
       }
46     SharedAdder.thread = MuloThread.startThread(this, Mulo.MULO + "-SharedAdder");
```

```
    }

49  /**
     * Add a Shared file on Shared list
     */
52  @Override
    @Deprecated
    public void go() {
55    Thread.currentThread().setName(Mulo.MULO + "-SharedAdder"); // XXX Core bug
      try {
        if (Shared.add(this.path, this.recursive)) {
58        Out.printImportant("\"" + this.path + "\" is now in the shared list.");
        }
      }
61    catch (IllegalArgumentException e) {
        Out.printImportant("\"" + this.path + "\" not found.");
      }
64    catch (Exception e) { // should not happen
        Log.printError("Exception in SharedAdder thread, autorestarting it", e);
        SharedAdder.thread = MuloThread.startThread(this, Mulo.MULO + "-SharedAdder");
67    }
      Log.printNotice("quitting SharedAdder thread");
      SharedAdder.thread = null;
70    MuloThread.endThread();
    }
  }
```

B.2 Pacchetti Compressi

`PacketTCP`

Listato B.6: Modifiche di `PacketTCP` in `Packets.java`

```
    final static PacketTCP decodePacket(byte[] packet, Peer source) {
      if (packet == null || packet.length == 0) {
3       return null;
      }
      PacketTCP decodedPacket;
6     boolean success = false;
      boolean compress = false;
      if (packet.length >= HEADER_SIZE) { // long enough to get protocol, size and type
9       // get the right subclass constructor for this protocol & type

        if (packet[0] == Packet.Protocol.COMPRESSED.id) {
12        packet = unpackPacket(packet);
          compress = true;
        }
15      Constructor<?> constructor = ( packet[0] == Packet.Protocol.ED2K.id ?
            ed2kTypeConstructors[packet[5] & 0xFF] : eMuleTypeConstructors[packet[5] & 0xFF] )
            ;
        try {
          if (constructor == null) { // make a generic packet with this type code
18          byte[] subPacket = new byte[Utils.bytesToInt(packet, 1) + HEADER_SIZE - 1];
            System.arraycopy(packet, 0, subPacket, 0, subPacket.length);
            PPLog.printWarning("Unknown PacketTCP (0x" + Utils.byteToHex(packet[5]) + ")" + (
                source == null ? "" : " from " + source.ed2kID ), subPacket);
21          decodedPacket = new PacketTCP(packet[0], packet[5]);
          }
          else { // constructor exists, call it
24          decodedPacket = (PacketTCP)constructor.newInstance(packet, source);
            success = true;
          }
27      }
        catch (InvocationTargetException e) { // the constructor threw some kind of exception
          PPLog.printWarning("Malformed (or wrongly filtered) " + ( constructor == null ? "
              PacketTCP" : constructor.getDeclaringClass().getSimpleName() ) + " (0x" + Utils.
              byteToHex(packet[5]) + ")" + ( source == null ? "" : " from " + source.ed2kID ),
               e);
30        decodedPacket = new PacketTCP(packet[0], packet[5]);
          decodedPacket.malformed = true;
        }
33      catch (Exception e) { // shouldn't happen
          PPLog.printError("Exception while trying to decode a packet" + ( source == null ? ""
               : " from " + source.ed2kID ), e);
          decodedPacket = new PacketTCP(packet[0], packet[5]);
36        decodedPacket.malformed = true;
```

```
        }
      }
      else { // not long enough for the type: we make a packet with type = 0x00
        PPLog.printWarning("Too short PacketTCP" + ( source == null ? "" : " from " + source.
            ed2kID ), packet);
        decodedPacket = new PacketTCP(packet[0], (byte)0);
        decodedPacket.malformed = true;
      }
      if (!success && packet.length >= 5) { // long enough to get size
        decodedPacket.size = Utils.bytesToInt(packet, 1) - 1;
      }
      if (compress) {
        decodedPacket.protocol = Packet.Protocol.COMPRESSED.id;
      }
      return decodedPacket;
    }

    public static byte[] unpackPacket(byte[] packet) {
      byte[] decompressed_data = Utils.unzip(packet, HEADER_SIZE, Utils.bytesToInt(packet, 1)
          - 1);
      ByteBuffer decompressed_packet = ByteBuffer.allocate(HEADER_SIZE + decompressed_data.
          length);
      decompressed_packet.position(0);
      if (packet[0] == Protocol.COMPRESSED.id) {
        if (eMuleTypeConstructors[packet[5] & 0xFF] != null) {
          // if packet is an eMule type we set eMule protocol
          decompressed_packet.put(Packet.Protocol.EMULE.id);
        }
        else {
          // otherwise we hope the packet use eDonkey protocol
          decompressed_packet.put(Packet.Protocol.ED2K.id);
        }
      }
      else {
        // otherwise we hope the packet use eDonkey protocol
        decompressed_packet.put(Packet.Protocol.KAD.id);
      }
      decompressed_packet.order(ByteOrder.LITTLE_ENDIAN).putInt(decompressed_data.length + 1);
      decompressed_packet.put(packet[5]);
      decompressed_packet.put(decompressed_data);
      return decompressed_packet.array();
    }

    /**
     * zip uncompressed TCP packets
     * @param packet the packet to zip
     * @return the packet zipped
     */
    public static byte[] packPacket(byte[] packet) {
```

```
        byte[] compressed_data = Utils.zip(packet, HEADER_SIZE, Utils.bytesToInt(packet, 1) - 1)
            ;
84      ByteBuffer compressed_packet = ByteBuffer.allocate(HEADER_SIZE + compressed_data.length)
            ;
        compressed_packet.position(0);
        compressed_packet.put(Packet.Protocol.COMPRESSED.id);
87      compressed_packet.order(ByteOrder.LITTLE_ENDIAN).putInt(compressed_data.length + 1);
        compressed_packet.put(packet[5]);
        compressed_packet.put(compressed_data);
90      return compressed_packet.array();
      }
```

PacketUDP

Listato B.7: Modifiche di `PacketUDP` in `Packets.java`

```
    final static PacketUDP decodePacket(byte[] originalPacket, InetAddress ip, int port,
        boolean calledByObfuscation) {
2     if (originalPacket == null || originalPacket.length == 0) {
        return null;
      }
5     PacketUDP decodedPacket;
      boolean compressed = false;
      if (originalPacket.length >= HEADER_SIZE) { // long enough to get protocol and type
8       // get the right subclass constructor for this protocol & type
        Constructor<?> constructor; // = null;
        byte[] packet = unobfuscatePacket(originalPacket, ip);
11      if (packet[0] == Protocol.COMPRESSED.id || packet[0] == Protocol.KAD_COMPRESSED.id) {
          packet = unpackPacket(packet);
          compressed = true;
14      }
        if (packet[0] == Protocol.ED2K.id) {
          constructor = ed2kTypeConstructors[packet[1] & 0xFF];
17      } else if (packet[0] == Protocol.EMULE.id) {
          constructor = eMuleTypeConstructors[packet[1] & 0xFF];
        } else if (packet[0] == Protocol.KAD.id) {
20        // update lastContact variable that stores the last time we recieved a KAD packet!
          KadContact.setLastContact();
          constructor = kadTypeConstructors[packet[1] & 0xFF];
23      } else {
          // No constructor found <- unreachable point...theoretically
          constructor = null;
26      }
        try {
          if (constructor == null) { // make a generic packet with this type code
29          byte[] subPacket = new byte[HEADER_SIZE];
            System.arraycopy(packet, 0, subPacket, 0, subPacket.length);
            PPLog.printWarning("Unknown PacketUDP (0x" + Utils.byteToHex(packet[1]) + ")" + (
                ip == null ? "" : " from " + ip + ':' + port ), subPacket);
```

```
32            decodedPacket = new PacketUDP(packet[0], packet[1]);
            }
            else { // constructor exists, call it
35            decodedPacket = (PacketUDP)constructor.newInstance(packet, ip, port);
            }
          }
38        catch (InvocationTargetException e) { // malformed packet
            PPLog.printWarning("Malformed (or wrongly filtered) " + ( constructor == null ? "
                PacketUDP" : constructor.getDeclaringClass().getSimpleName() ) + " (0x" + Utils.
                byteToHex(packet[1]) + ")" + ( ip == null ? "" : " from " + ip.getHostAddress()
                + ':' + port ), e.toString() + "\n\nPacket: " + Utils.bytesToHexStringJavaReady(
                packet));
            decodedPacket = new PacketUDP(packet[0], packet[1]);
41          decodedPacket.malformed = true;
          }
          catch (Exception e) { // shouldn't happen
44          PPLog.printError("Exception while trying to decode a packet" + ( ip == null ? "" : "
                 from " + ip + ':' + port ), e);
            decodedPacket = new PacketUDP(packet[0], packet[1]);
            decodedPacket.malformed = true;
47        }
        }
        else { // not long enough for the type: we make a packet with type = 0x00
50        PPLog.printWarning("Too short PacketUDP" + ( ip == null ? "" : " from " + ip + ':' +
              port ), originalPacket);
          decodedPacket = new PacketUDP(originalPacket[0], (byte)0);
          decodedPacket.malformed = true;
53      }
        if(compressed) {
          if (decodedPacket.protocol == Packet.Protocol.KAD.id) {
56          decodedPacket.protocol = Packet.Protocol.KAD_COMPRESSED.id;
          }
          else {
59          decodedPacket.protocol = Packet.Protocol.COMPRESSED.id;
          }
        }
62      return decodedPacket;
      }

65    /**
       * unzip compressed UDP packets
       * @param packet the packet to unzip
68     * @return the packet unzipped
       */
      public static byte[] unpackPacket(byte[] packet) {
71      byte[] decompressed_data = Utils.unzip(packet, HEADER_SIZE, packet.length - HEADER_SIZE)
            ;
        ByteBuffer decompressed_packet = ByteBuffer.allocate(HEADER_SIZE + decompressed_data.
            length);
```

```
        decompressed_packet.position(0);
74      if (packet[0] == Protocol.COMPRESSED.id) {
          if (eMuleTypeConstructors[packet[5] & 0xFF] != null) {
            // if packet is an eMule type we set eMule protocol
77          decompressed_packet.put(Packet.Protocol.EMULE.id);
          }
          else {
80          // otherwise we hope the packet use eDonkey protocol
            decompressed_packet.put(Packet.Protocol.ED2K.id);
          }
83      }
        else {
          decompressed_packet.put(Packet.Protocol.KAD.id);
86      }
        decompressed_packet.put(packet[1]);
        decompressed_packet.put(decompressed_data);
89      return decompressed_packet.array();
      }

92    /**
       * zip uncompressed UDP packets
       * @param packet the packet to zip
95     * @return the packet zipped
       */
      public static byte[] packPacket(byte[] packet) {
98      byte[] compressed_data = Utils.zip(packet, HEADER_SIZE, packet.length - HEADER_SIZE);
        ByteBuffer compressed_packet = ByteBuffer.allocate(HEADER_SIZE + compressed_data.length)
            ;
        compressed_packet.position(0);
101     compressed_packet.put(Packet.Protocol.COMPRESSED.id);
        compressed_packet.put(packet[1]);
        compressed_packet.put(compressed_data);
104     return compressed_packet.array();
      }
```

sendNextCompressedDataPacket

Listato B.8: **sendNextCompressedDataPacket** in **UploadSession.java**

```
      /**
       * Sends a new data packet to the peer using compression, from the list of requested
           fragments.
3      * @return Number of useful (file data) bytes sent. 0 if there are no requested fragments
           or send failed.
       */
      synchronized int sendNextCompressedDataPacket() {
6       if (this.requestedFragments == null || this.requestedFragments.isEmpty()) { // no
            requests pending! :/
          return 0;
```

```
    }
    int dataLength = 0; // overhead excluded
    PacketTCPFileDataCompressed dataPacket;
    synchronized (this.requestedFragments) { // to avoid it be changed
      long[] nextFragment = this.requestedFragments.get(0);
      try {
        if (this.compressedOffset + Config.BYTES_SENT_PER_PACKET * 12 / 10 >= this.
            compressedFragment.length) {
          // the requested fragment is being completed
          // note that we might be sending up to 120% of Config.BYTES_SENT_PER_PACKET (to
              avoid having a last tiny
          // packet)
          dataPacket = new PacketTCPFileDataCompressed(
              this.file.md4Hash, this.compressedFragment,
              this.compressedOffset, nextFragment[0],
              this.compressedFragment.length);
          this.requestedFragments.remove(0);
          // Fragment uploaded sucessfully.
          this.compressedOffset = 0;
          // If there is another Fragment we zip it.
          if (this.requestedFragments != null
              && !this.requestedFragments.isEmpty()) {

            nextFragment = this.requestedFragments.get(0);
            try {
              this.compressedFragment = Utils.zip(this.file.getData(nextFragment[0], (int)(
                  nextFragment[1] - nextFragment[0] )));
            }
            catch (IOException e) {
              // if something goes wrong, we set compressedFragment to null and send the
                  fragment
              // with no compression.
              this.compressedFragment = null;
            }
          }
        }
        else {
          // a part of the requested fragment is being sent
          dataPacket = new PacketTCPFileDataCompressed(
              this.file.md4Hash, this.compressedFragment,
              this.compressedOffset, nextFragment[0],
              Config.BYTES_SENT_PER_PACKET);
          this.compressedOffset += Config.BYTES_SENT_PER_PACKET; // next time we'll start
              from here
        }
      }
      catch (IOException e) {
        PPLog.printError("IOException reading file " + this.file.path, e);
        this.file.closeHandle();
```

```
        return 0;
      }
    }
    if (!this.peer.send(dataPacket)) { // send failed
      return 0;
    }
    dataLength = dataPacket.reassembledLenght;
    this.roundSentBytes += dataLength;
    this.peer.dataSentBytes += dataLength;
    return dataLength;
  }
```

storeDataCompressed

Listato B.9: **storeDataCompressed** in **Download.java**

```
/** Saves the data contained in the packet in the proper memory position, if packet and
    source are valid. */
private boolean storeDataCompressed(PacketTCPFileDataCompressed packet, Peer source) {
  if (packet.startOffset > this.file.size) {
    PPLog.printWarning("Offsets of compressed data packet received from " + source.getIP()
        + " are invalid", "File: " + this.file.getName() + " - Size: " + this.file.size +
        " - Start offset: " + packet.startOffset);
    source.compressedDataBuffer = null;
    return false;
  }
  if (packet.reassembledLenght < packet.compressedData.length) { // shouldn't happen
    PPLog.printError("Absurd compressed data packet received from " + source.getIP(), "
        File: " + this.file.getName() + " - Start offset: " + packet.startOffset + " - Zip
        Data length: " + packet.compressedData.length);
    source.compressedDataBuffer = null;
    return false;
  }
  if (source.compressedDataBuffer != null) {
    long startOffset = source.compressedDataBuffer.getLong(0);
    if (startOffset != packet.startOffset) {
      PPLog.printError("Absurd compressed data packet received from " + source.getIP(), "
          File: " + this.file.getName() + " - Wrong start offset: " + packet.startOffset +
          " - Expected start offsett: " + startOffset);
      source.compressedDataBuffer = null;
    }
  }
  if (source.compressedDataBuffer == null) {
    source.compressedDataBuffer = ByteBuffer.allocate(8 + packet.reassembledLenght).order(
        ByteOrder.LITTLE_ENDIAN);
    source.compressedDataBuffer.putLong(packet.startOffset);
  }
  if (( source.compressedDataBuffer.remaining() ) < packet.compressedData.length || packet
      .reassembledLenght != source.compressedDataBuffer.capacity() - 8) {
```

```
25        PPLog.printError("Absurd compressed data packet received from " + source.getIP(), "
              File: " + this.file.getName() + " - Start offset: " + packet.startOffset + " - Zip
               Data length: " + packet.compressedData.length);
          source.compressedDataBuffer = null;
          return false;
28      }
        source.compressedDataBuffer.put(packet.compressedData);
        if (!source.compressedDataBuffer.hasRemaining()) {
31        int p = (int)( packet.startOffset / K.PART_SIZE ); // part number
          int c = (int)( packet.startOffset % K.PART_SIZE ) / K.CHUNK_SIZE; // chunk number
          Part.Chunk chunk = this.parts[p].chunks[c]; // the chunk we're writing to
34        byte[] decompressedData = Utils.unzip(source.compressedDataBuffer.array(), 8, packet.
              reassembledLenght);
          chunk.storeData(decompressedData, packet.startOffset, source);
          int endOffset = (int)( packet.startOffset + decompressedData.length );
37        if (decompressedData.length > chunk.size - (int)( packet.startOffset - chunk.
              fileOffset )) {
            // Then this fragment of data belongs to two chunks
            if (++c < this.parts[p].chunks.length) { // same part
40            chunk = this.parts[p].chunks[c];
            }
            else if (++p < this.parts.length) { // the fragment crosses two different parts!
43            chunk = this.parts[p].chunks[0];
            }
            else { // goes beyond end of file :/ shouldn't happen
46            throw new AssertionError("Packet offsets: " + packet.startOffset + "/" + endOffset
                    + ", file size: " + this.file.size + ", part/chunk: " + p + "/" + c);
            }
            chunk.storeData(decompressedData, packet.startOffset, source); // again
49        }
          source.compressedDataBuffer = null;
          return true;
52      }
        else {
          return false;
55      }
      }
```

Elenco delle figure

Elenco delle tabelle

Elenco dei listati

Bibliografia

[eDonkeyProtocol] Yoram Kulbak and Danny Bickson, *The eMule Protocol Specification*, 2005.

[DiPieri] Alessandro Di Pieri, *PariMulo: Autologin e Annotazioni degli utenti*, 2010.

[Piccolo] Christian Piccolo, *PariKad*, 2010.

[Muscarella] Martina Muscarella, *PariPari-Mulo: AICH*, 2009.

[Daberdaku] Sebastian Daberdaku, *PariMulo: Credits*, 2010.

[Ampezzan] Roberto Ampezzan, *PariMulo 2009*, 2009.