UNIVERSITÀ DEGLI STUDI DI MILANO
POLO DIDATTICO E DI RICERCA DI CREMA

Comportamento di eMule sulla rete eD2k

Progetto di

Sistemi Distribuiti

Studente:
Lena Cota Guido 773358

Docente:
Tettamanzi Andrea

Anno Accademico 2010/2011

Indice

Introduzione....3
Cenni sul peer-to-peer....5
Caratteristiche delle reti di eMule....11
Interazioni Client - Server....17
Interazioni Client – Client....31
Identificazione Sicura degli Utenti....46
Appendice A: I File....63
Appendice B: Gestione delle fonti....67
Appendice C: Code e crediti....69
Appendice D: Creazione coppia di chiavi....75
Fonti....78

Introduzione

eMule è una popolare applicazione peer-to-peer (P2P) di *file sharing* per *Microsoft windows*, rilasciata sotto licenza *open source* e implementata in C++. L'IDE utilizzato dagli sviluppatori è infatti *Microsoft Visual C++*, che sfrutta le librerie MFC (*Microsoft Foundation Classes*) per incapsulare le *Windows API*. Va segnalata tuttavia l'esistenza di una versione del programma pienamente compatibile con sistemi Linux e Mac OS X, chiamata **aMule**.

eMule vanta un'attiva e numerosa comunità che contribuisce a mantenere vivo il progetto: da un lato gli sviluppatori riescono a rilasciare nuove versioni del programma a scadenze più o meno regolari (5-6 mesi), dall'altro gli utenti lo hanno reso il programma più scaricato in assoluto da *SourceForge* (più di 545 milioni di download a novembre 2010).

Nel momento in cui si scrive, la versione corrente è la 0.5a. Ciò indica che nonostante il programma sia diffuso da tempo e ormai decisamente stabile, venga ancora considerato dai suoi sviluppatori in fase di beta.

Cenni storici

Il progetto **eMule** nasce il 13 maggio 2002, quando il programmatore tedesco Hendrik Breitkreus (conosciuto col nickname *Merkur*) decise insieme ad un gruppo di programmatori di migliorare prestazioni e funzionalità del client di *file sharing* **eDonkey**. Il loro intento non era dunque creare da zero una nuova rete P2P di condivisione, ma sfruttarne una già esistente rendendola più affidabile, equa e sicura.

Il 7 luglio dello stesso anno rilasciarono su *SourceForge* il sorgente della prima versione, mentre il 9 agosto pubblicarono la prima versione utilizzabile sulla rete.

Grazie alla sua compatibilità con la rete **eDonkey** e alle numerose funzioni aggiuntive, **eMule** si afferma presto come client di riferimento per il *file sharing* su quel tipo di rete. Alcune delle sue caratteristiche vincenti sono state:

- scambio diretto di fonti tra peer, che migliora notevolmente la ricerca dei file;
- sistema di crediti per premiare chi condivide (introdotto nella versione 0.19a);
- controllo e ripristino di download corrotti (introdotto nella versione 0.44a);
- trasmissione dei dati in formato compresso *zlib* per risparmiare banda;
- supporto alla rete *serverless* **Kad**, un'implementazione del protocollo *Kademlia*;
- rilascio con licenza *open source*.

Concetti base

eMule è un programma di *file sharing*, quindi progettato per la condivisione di file all'interno di una rete comune, che nel suo caso sono di tipo *peer-to-peer*. Le reti P2P supportate sono **eDonkey2000** (o **eD2k**, *server based*) e **Kad** (*serverless*).

All'interno delle reti sia i file che i client sono identificati in modo univoco attraverso ID calcolati con funzioni di *hashing*. I client condividono con la rete alcuni file (diventandone fonti) e possono cercarne altri per poi scaricarli. Si noti che il processo di download non è minimamente influenzato dalla rete scelta per la ricerca.

Quando il client ottiene l'elenco delle fonti per un certo file, si mette in coda presso di esse e aspetta il suo turno per scaricarne una parte. Quando raggiunge la prima posizione della coda comincia il download. Il sistema dei crediti incide molto sui tempi di attesa, premiando i client che condividono di più.

Scopo del progetto

Nei prossimi capitoli verrà analizzato il comportamento di **eMule** sulla rete **eD2k**, quindi gli attori e le caratteristiche del protocollo di comunicazione e dell'applicativo.

Dopo un capitolo introduttivo sulla tecnologia *peer-to-peer* sarà fornita una panoramica sulle caratteristiche delle due reti utilizzate da **eMule**. Successivamente si entrerà nei dettagli della rete **eD2k**, considerando prima le interazioni tra *Client e Server* e poi quelle tra *Client e Client*. Si proseguirà con un approfondimento sul meccanismo di *Identificazione Sicura degli Utenti*, e infine gli ultimi capitoli di appendice affronteranno alcune caratteristiche di gestione dei file, delle fonti e delle code di upload.

Cenni sul peer-to-peer

In generale con il termine *peer-to-peer* (P2P) si indica la condivisione di risorse e/o servizi tra computer attraverso uno scambio diretto e paritario. In particolare una rete P2P è una rete di computer che non possiede nodi gerarchizzati secondo il modello canonico client-server, ma nodi (*peer*) che possono essere sia l'una che l'altra cosa, quindi sia fornitori che fruitori di servizi. Per risorse e servizi si intendono ad esempio informazioni, file, banda, supporti di archiviazione o cicli del processore. Ogni *peer* della rete è in grado di determinare autonomamente quando e per quanto tempo li condividerà con gli altri nodi.

Le reti P2P possono essere distinte in due categorie:

- **rete P2P pura**, completamente decentralizzata, quindi senza un'autorità di controllo centrale né per organizzare la rete (fase di *setup*) né per coordinarla in seguito (fase di *sequence*). Ogni nodo ha stessi diritti e funzioni, e non ha una visione completa della rete;
- **rete P2P ibrida**, in cui un sottoinsieme di nodi si comporta come un server ed è in grado di offrire funzioni aggiuntive (ad esempio di indicizzazione dei contenuti o di autenticazione).

Altre due importanti caratteristiche di una rete *peer-to-peer* sono la scalabilità e l'affidabilità. La prima impone che non debba esistere un limite algoritmico o tecnologico alla dimensione della rete, o che comunque non sia legato al numero dei nodi. Per affidabilità si intende invece che il malfunzionamento di un qualsiasi nodo non deve ripercuotersi sull'intero sistema, ancora meglio se su nessun altro nodo.

Modello a tre livelli

Una rete P2P può essere modellata in tre livelli: infrastruttura P2P, applicazioni P2P e comunità P2P.

Infrastruttura P2P

Si pone al di sopra della rete di telecomunicazione esistente e fa da fondamenta per i livelli successivi. Può essere considerata come l'insieme di tecniche e meccanismi che permettono la comunicazione e l'integrazione tra componenti IT, in particolare tra applicazioni.

Il problema dell'integrazione è particolarmente complesso, perché gli ambienti hardware/software sono estremamente eterogenei e raramente interoperabili tra loro. Per questo motivo numerosi consorzi come il *W3C* e il *Global Grid Forum* hanno dedicato molti sforzi per standardizzare le interfacce tra le diverse applicazioni P2P, dando vita ad esempio a protocolli come Jabber (sistema di messaggistica istantanea) o Groove (piattaforma per lo sviluppo di applicazioni P2P).

Un altro problema che bisogna affrontare a livello di infrastruttura è quello della sicurezza, perché l'uso condiviso di risorse avviene spesso tra *peer* che non si conoscono affatto, e che devono per forza di cose fidarsi dell'altro. Tra i sistemi di sicurezza da integrare in questo livello citiamo ad esempio la cifratura delle comunicazioni, i meccanismi di autenticazione, i controlli sull'integrità dei dati, ed altri ancora.

Applicazioni P2P

La applicazioni P2P utilizzano l'infrastruttura per rendere possibile la comunicazione e la collaborazione tra i partecipanti della rete senza la necessità di un controllo centrale.

Esistono molti tipi di applicazioni P2P, ad esempio quelle di messaggistica istantanea, *file sharing*, *grid computing* e di collaborazione. Il confine tra le categorie è spesso sfumato.

Comunità P2P

Se nei primi due livelli il termine *peer* si riferisce a un'entità di tipo tecnologico, nella comunità P2P ci si riferisce all'utente che usa una certa applicazione, quindi a una persona. Il livello si focalizza sui fenomeni di interazione sociale del P2P, in particolare sulla formazione di comunità e sulle dinamiche che si creano al loro interno.

Vantaggi e svantaggi

Tra i vantaggi di una rete P2P vanno considerati:

- una *riduzione dei costi*, intesa sia come diminuzione delle spese di acquisto e mantenimento delle infrastrutture e dei sistemi di comunicazione (vengono sfruttate risorse già esistenti), sia come riduzione del numero di amministratori da stipendiare;
- maggiore *scalabilità*, dal momento che le reti P2P riducono la dipendenza da nodi centrali, dunque ostacolano la formazione di colli di bottiglia distribuendo i flussi di informazione e creando repliche;
- possibilità di creare facilmente reti ad hoc, dato che le reti *peer-to-peer* per loro stessa natura tollerano nativamente le connettività intermittenti.

Ed ora gli svantaggi:

- l'implementazione di sistemi di sicurezza è più complessa rispetto alle reti con paradigma client-server;
- non garantiscono la disponibilità continuata di risorse e servizi, soprattutto se hanno pochi *peer*;
- richiedono un maggior numero di repliche per ogni file.

File sharing

Il *file sharing* è probabilmente l'applicazione *peer-to-peer* più diffusa. La sua caratteristica principale è che i *peer* che hanno scaricato un file come client, successivamente lo rendono disponibile agli altri *peer* agendo come server.

Il problema centrale è quello del *lookup*, ovvero della localizzazione delle risorse. Nel contesto dei sistemi di *file sharing* sono stati sviluppati tre diversi algoritmi, che illustreremo di seguito.

Flooded request model

Nel *flooded request model* non c'è bisogno di alcuna autorità di coordinamento centrale, quindi è perfetto per reti P2P pure. Le *query* di ricerca di un file sono propagate nella rete secondo un modello di tipo *flooding*, ovvero trasmesse a tutti i *peer* che si trovano entro un certo *time-to-live* (TTL) da quello richiedente. Se la ricerca

ha esito positivo gli viene ritornato l'indirizzo del *peer* che possiede il file, così che possa scaricarlo connettendosi direttamente a lui.

Alcuni problemi del modello:

- più la ricerca va in profondità e più i messaggi di richiesta nella rete aumentano in modo esponenziale, generando un traffico di *overhead* eccessivo;
- non c'è garanzia che la risorsa venga effettivamente localizzata.

Esempio applicazione: Gnutella.

Modello con directory centralizzata

Questo modello rappresenta un perfetto esempio di sistema P2P ibrido, in cui una parte delle funzionalità dell'infrastruttura - quelle di indicizzazione - sono fornite in modo centralizzato da un'entità coordinatrice.

Esempio applicazione: Napster.

Document routing model

Il *document routing model* è quello utilizzato dall'applicazione Freenet. Si tratta di un protocollo particolare in cui i file non sono memorizzati sull'hard disk del *peer* che li fornisce, ma in altri posti della rete. Il motivo di questa scelta è realizzare una rete in cui sia possibile leggere e scrivere informazioni in modo totalmente anonimo. Ciò richiede che il proprietario di un nodo non sappia quali documenti siano effettivamente memorizzati sul proprio hard disk, per cui file e utenti devono essere identificati in modo non ambiguo attraverso un ID univoco.

Il *document routing model* dà all'infrastruttura di rete robustezza e garantisce l'anonimato, ma per contro è particolarmente complesso da gestire e favorisce la formazione di "isole", cioè partizioni della rete in cui le comunità che ne fanno parte non hanno più connessioni col resto della rete.

Topologie di una rete P2P

Topologia centralizzata

La topologia centralizzata si basa sul modello tradizionale client-server, e prevede l'esistenza di un server centrale che tenga traccia dei file e degli utenti connessi. Ogni

volta che un client vorrà accedere alla rete, dovrà contattare il server per informarlo del suo indirizzo IP corrente e del nome di tutti i file che vuole condividere. Le informazioni raccolte dai *peer* saranno utilizzate dal server per aggiornare dinamicamente il database centralizzato (spesso locale) che mappa i nomi dei file negli insiemi di indirizzi IP dei client che li condividono. Quando un client vorrà scaricare un certo file non dovrà fare altro che chiedere al server di fare una ricerca sul proprio database e di restituirgli l'indirizzo del *peer* che lo condivide, con cui potrà connettersi direttamente in seguito. Si noti che il server non salva alcun file in locale, ma solo le informazioni sui client che lo possiedono.

La topologia centralizzata è molto semplice da realizzare ma soffre di due problemi difficilmente evitabili: il server centrale rappresenta sia un *single point of failure* sia un collo di bottiglia.

Topologia ad anello

In questa topologia i *peer* sono disposti in modo da formare un anello che agisce da server distribuito. Questo cluster di macchine lavora in modo da ottenere il miglior bilanciamento del carico e la maggiore disponibilità dei file possibile.

La topologia ad anello richiede che le macchine che ne fanno parte siano abbastanza vicine tra loro sulla rete, il che capita normalmente quando fanno parte della stessa organizzazione.

Topologia gerarchica

La topologia gerarchica è particolarmente adatta per quei sistemi che richiedono una forma di *governance* che comprende deleghe di permessi o di autorità. I DNS ne sono l'esempio classico.

Topologia decentralizzata

La topologia decentralizzata è quella delle architetture P2P pure, in cui tutti i *peer* hanno uguali funzionalità e diritti, e quindi formano una topologia di rete piatta e non strutturata. Quando un *peer* vuole unirsi alla rete dovrà come prima cosa contattare un nodo di *bootstrapping*, ovvero un nodo già connesso che gli fornisca l'indirizzo IP di uno o più *peer* esistenti, rendendolo di fatto parte della rete dinamica. Si noti che ogni *peer* ha una visione parziale della rete, limitata ai suoi vicini (i nodi con cui ha un

collegamento diretto).

Poiché non ci sono server a cui rivolgersi per le ricerche, queste vengono tipicamente propagate nella rete con un modello di tipo *flooding*.

Topologia ibrida

Questo tipo di topologia combina diversi tipi di topologie precedentemente illustrate. Ad esempio la *topologia centralizzata e decentralizzata* è caratterizzata da alcuni *peer* che hanno funzionalità aggiuntive rispetto agli altri e sono chiamati *supernodi*. Questi svolgono più o meno le stesse funzioni dei server nelle topologie centralizzate, ma solo su un sottoinsieme dei *peer* della rete. Inoltre i supernodi sono collegati tra loro in modo decentralizzato. Abbiamo quindi due diversi livelli di controllo: quello dei nodi ordinari che si collegano a un supernodo secondo un sistema centralizzato, e quello dei supernodi che si connettono tra loro in modo non centralizzato.

Impatto su Internet

I software P2P sono stati tra le *killer application* di Internet. Milioni e milioni di utenti li usano abitualmente per scaricare file, guardare media in streaming, telefonarsi via VoIP (Voice over IP). Se questa possibilità per gli utenti è vista come uno strumento utile e divertente, per gli ISP (Internet Service Provider) è decisamente un problema, dal momento che questo tipo di programmi generano un volume di traffico imponente. Secondo uno studio del 2009 di IPOQUE, una società tedesca specializzata in analisi del traffico dati, il *peer-to-peer* rappresenta dal 50% al 70% del traffico totale sulla rete.

Uno dei motivi per cui il traffico generato è così massiccio è che i sistemi P2P spesso costruiscono le loro reti *overlay* in modo casuale, ignorando - per semplicità di progettazione - la topologia di rete sottostante. Considerando ad esempio un software di *file sharing*, la scelta del nodo da cui scaricare non tiene minimamente conto dell'effettiva distanza sulla rete tra i due *peer;* questo significa che un nodo di Milano potrà scaricare tanto da uno di Como quanto da uno di Los Angeles. Per migliorare questo aspetto sono stati proposti metodi come **ALTO** (*Application Layer Traffic Optimization*), il cui scopo è condizionare la scelta delle fonti dando priorità a quelle topologicamente più vicine al *peer* che le sta cercando.

Caratteristiche delle reti di eMule

Le reti P2P supportate da **eMule** sono due: **eD2k** (*server based*) e **Kad** (*serverless*).

Di seguito si darà una rapida visione di entrambe, soffermandosi in particolare sulla seconda, su cui non si tornerà più nei prossimi capitoli.

Rete eD2k

Il client **eMule** nasce come versione potenziata di **eDonkey**, il client dell'omonima rete *peer-to-peer* di *file sharing* **eDonkey2000** (in breve **eD2k**). Quest'ultima è a sua volta basata sul protocollo di trasferimento file *Multisource File Transfer Protocol* (MFTP), un'estensione dell'FTP che permette di scaricare contemporaneamente lo stesso file da diverse fonti. L'osservazione alla base di questa scelta è che in media gli utenti trasmettono file molto più lentamente di quanto li scarichino, e permettere di scaricarli da più parti consente di raggiungere velocità di download accettabili.

Rispetto all'MFTP **eDonkey** mette a disposizione caratteristiche avanzate per il *file sharing*, come ricerche basate su metadati, condivisione parziale dei download, supporto alle collezioni, scambio di fonti tra i client e rilevamento e correzione delle corruzioni nei file trasmessi.

La topologia di rete è ibrida: esiste un primo livello composto dai server (che si occupano dell'indicizzazione dei file e delle fonti), ed un secondo livello formato dai client che si connettono direttamente tra loro per condividere i file.

eD2k è una delle più grandi reti *peer-to-peer* finalizzate al *file sharing*, ma è piuttosto lenta. I suoi ideatori hanno infatti preferito puntare tutto sulla disponibilità dei file condivisi, piuttosto che sulla rapidità con cui vengono scaricati. Chi cerca reti veloci dovrebbe quindi affidarsi a tecnologie alternative come quella dei **Torrent**, mentre **eDonkey** rimane imbattibile per ciò che riguarda il reperimento di risorse introvabili su altre reti.

Client

I client non sono altro che i *peer* degli utenti, quindi le loro attività principali saranno la ricerca, il download e la condivisione dei file. Perché diventino operativi devono prima di tutto connettersi a un server (in modo automatico o manuale), scelto tra quelli elencati in un file salvato in locale. Una volta connessi, i client possono collegarsi a diverse centinaia di client contemporaneamente, così da condividere i file e scambiarsi informazioni sui server o sulle fonti.

Per identificare in modo univoco i client all'interno della rete viene utilizzata la seguente coppia di valori: *UserHash* e *Client ID,* che saranno ripresi nel capitolo successivo.

Server

I server non sono responsabili dell'archiviazione dei file condivisi sui loro dischi (come avveniva ad esempio con *Napster*), ma dell'aggiornamento e la distribuzione delle informazioni sui client che li condividono, chiamati **fonti**. I server quindi non sono altro che centri di comunicazione per i client e localizzatori di file all'interno della rete. Le informazioni sugli utenti e sui dati condivisi sono memorizzate in database interni, che in media tengono traccia di centinaia di migliaia di file e client attivi.

Quando un client chiede di connettersi a un server, quest'ultimo verifica se il client è opportunamente configurato per accettare connessioni da altri client. In caso positivo gli assegna un *ID Alto*, altrimenti un *ID Basso*. Come vedremo ciò avrà importanti ripercussioni sul comportamento del client all'interno della rete.

La configurazione dei server include due tipi di limiti basati sul numero di utenti attivi:

- **soft limit**: quando i server superano questo valore smettono di accettare connessioni da client con ID Basso;
- **hard limit** (>= *soft limit*): quando i server superano questo valore significa che sono pieni, e rifiuteranno richieste di connessione da qualsiasi tipo di client.

Il fatto che il funzionamento della rete **eDonkey** dipenda dalla presenza di server costituisce il suo punto debole, perché devono rimanere costantemente attivi nonostante il pesante traffico di rete cui sono sottoposti e l'esposizione ad attacchi interni o esterni. A questi due problemi negli ultimi cinque anni se ne è aggiunto un

altro, ovvero il diffondersi di server “civetta” creati ad hoc per tracciare gli utenti che condividono dati protetti da diritti d'autore.

Per tutti questi motivi c'è stata una grossa spinta per affiancare la rete decentralizzata **Kad** a quella **eD2k**, supportata stabilmente dal client **eMule** a partire dalla versione 0.40.

File

I file sono identificati in modo univoco all'interno della rete **eDonkey** grazie a una chiave chiamata *FileHash*, che dipende esclusivamente dal contenuto del file e non dal nome con cui è salvato in locale. I principali motivi di questa scelta sono:

- massimizzare la velocità di scaricamento permettendo il download contemporaneo di uno stesso file da più fonti;
- rilevare e correggere i file corrotti, attraverso il sistema **AICH** (maggiori dettagli in *Appendice A*).

I file vengono divisi in parti da 9,28 MB, chiamati **chunk**, a loro volta suddivisi in **blocchi** da 180 KB. Questa segmentazione insieme al *FileHash* è alla base sia del sistema di download multi-fonte che di quello di rilevazione degli errori.

Rete Kad

La rete **Kad** è un'implementazione del protocollo *peer-to-peer* per reti decentralizzate **Kademlia**, ideato da Petar Maymounkov e David Mazières. Il **Kad** non è altro che una *Distributed Hash Table* (DHT) su una rete *overlay*, quindi non ci sono server, ma si tratta di una rete completamente decentralizzata in cui tutte le informazioni sono memorizzate direttamente nei *peer*.

Il protocollo non si limita a specificare la struttura della rete, ma regola la comunicazione tra i nodi e come deve aver luogo lo scambio di informazioni, sempre attraverso connessioni UDP.

Come già detto, il supporto alla rete **Kad** è stato introdotta nella versione 0.40 di **eMule,** come reazione ai numerosi problemi (tecnici e giudiziari) dei server **eD2k.** Negli anni è stata notevolmente migliorata con la correzione di numerosi bug e l'introduzione di nuove funzionalità, soprattutto quelle che riguardano la sicurezza

(protezione generale *anti-flood*, offuscamento del protocollo, filtri anti-spam).

Connessione alla rete

Ogni nodo sul **Kad** viene identificato dall'*ID Nodo*, un numero che viene usato anche per localizzare i valori, come ad esempio l'hash di un file o una parola chiave.

Quando un nodo vuole unirsi alla rete dovrà prima superare un processo di avvio chiamato *bootstrap*, durante il quale deve fornire l'indirizzo IP di almeno un altro nodo che stia già partecipando alla rete **Kademlia**; quest'ultimo può essere conosciuto direttamente dall'utente o essere preso da un elenco. Se è la prima volta che il nodo partecipa alla rete calcola un numero identificativo casuale che non sia stato ancora assegnato a nessuno, e lo usa per l'intera durata della connessione.

Appena entrato nella rete il *peer* chiede agli altri client se riescono a connettersi correttamente alla sua porta comunicata. Si tratta di un controllo molto simile a quelle fatto dai server **eD2k** per assegnare ID Alti o Bassi. Sul **Kad** in caso di successo viene assegnato uno stato "*open*", altrimenti quello "*firewalled*". In quest'ultimo caso è prevista la presenza di un *Buddy* di supporto, ovvero uno o più client con stato *open* che fanno da ripetitori per le connessioni con nodi *firewalled*.

Ricerca

Sul **Kad** la ricerca dei file e delle fonti funziona più o meno con lo stesso meccanismo, e prevede la partecipazione di tutti i nodi con cui si viene in contatto.

Prima di tutto va detto che l'informazione sulla rete **Kademlia** è memorizzata nei cosiddetti **valori**, che possono essere di qualsiasi tipo ma generalmente sono puntatori a file. Possiamo considerare i valori come gli indirizzi delle risorse da condividere, e vengono indicizzati assegnandogli una chiave che ha lo stesso formato dell'*ID Nodo*. In questo modo si potrà calcolare la distanza “nodi - chiavi” esattamente come si faceva tra nodi, così da capire qual è il client con distanza minima dall'hash del file di interesse, da cui poi andrò a chiedere gli IP dei nodi da cui poterlo scaricare.

L'algoritmo **Kademlia** si basa dunque sul concetto di distanza tra nodi, in base al quale è possibile stabilire se due nodi sono più o meno vicini rispetto a un terzo. Si misura calcolando lo *XOR* tra due ID Nodo, o tra un ID Nodo e una chiave; in entrambi i casi viene restituito un numero intero. Si noti che questa distanza non ha nulla a che

vedere con le estensioni geografiche della rete, ma indica semplicemente la distanza tra i range degli identificativi.

Quando si cerca una chiave l'algoritmo esplora la rete in passi successivi, avvicinandosi sempre di più ad ogni passaggio; il processo si ferma quando il nodo contattato restituisce il valore o quando non ci sono più nodi da interrogare. Più precisamente, l'algoritmo procede iterativamente nella ricerca delle chiavi attraverso la rete, avvicinandosi di un bit al risultato ad ogni passo compiuto. Ne consegue che una rete **Kad** con 2^n nodi richiederà al massimo n passi per trovare il nodo cercato. Ciò significa che il numero di nodi contattati durante la ricerca dipende solo marginalmente dalle dimensioni della rete: se il numero dei partecipanti alla rete raddoppia, allora il nodo di un utente deve interrogare solo un nodo in più per ogni ricerca, e non il doppio di quelli che ha contattato prima. **Kademlia** è quindi molto efficiente per le operazioni di ricerca (così come la maggior parte delle strutture DHT), dato che su un totale di n nodi nel sistema ne contatta solo $O(log(n))$ durante la ricerca.

Rete decentralizzata

Il **Kad** è una rete decentralizzata, quindi non soffre di quei colli di bottiglia tipici di una rete con server. Come abbiamo già detto, i client non si devono connettere a un server per accedere alla rete, ma basta un client con indirizzo IP e porta noti che sia già attivo. Ciò rappresenta un'ottima garanzia di continuità per il servizio: un *peer* che abbandona la rete non rappresenta che una frazione minima dei dati e dei file contenuti sui milioni di computer connessi ad **eMule**, quindi la sua disconnessione passerà inosservata. In una rete centralizzata come **eDonkey** invece il fallimento o la chiusura di un grosso server avrebbe ripercussioni importanti sulla qualità del servizio. Una grossa spinta allo sviluppo del **Kad** è venuta proprio dalla chiusura forzata nel 2006 del più grande server dell'epoca – *Razorback2* – ad opera della polizia federale belga. Dal 2008 ad oggi ci sono stati altri attacchi ai principali server **eD2k**, ed il **Kad** è la strada più concreta per un futuro per **eMule**.

Il fatto di essere decentralizzata aumenta anche la sua resistenza agli attacchi di tipo *Denial Of Service*: anche se un intero insieme di nodi venisse preso di mira, gli effetti sulla rete sarebbero molto limitati perché isolerebbe automaticamente i nodi problematici.

Elenco dei contatti

Per garantire il buon funzionamento degli scambi, ogni client **Kad** deve creare un elenco dei contatti che comprenda altri client. Queste informazioni sono generate automaticamente e salvate nel file `notes.dat` (nella cartella `emule/config`), insieme all'elenco dei file che abbiamo già cercato.

L'elenco dei contatti è estremamente dinamico e si evolve regolarmente. Ogni volta che **Kad** effettua una nuova ricerca vengono aggiunti nuovi nodi ed altri vengono rimossi, ad esempio perché non sono più in rete o perché avevano IP dinamici che ora sono cambiati.

Dalla versione 0.49 di eMule l'aggiornamento dell'elenco avviene direttamente in RAM, mentre il file `notes.dat` viene aggiornato solo alla corretta chiusura del client.

Interazioni Client - Server

L'architettura della rete **eD2k** è ibrida e a due livelli: il primo è quello dei server, mentre il secondo è quello dei client che si connettono alla rete per scambiarsi i file. Il principio fondamentale è che la connessione alla rete coincide con la connessione ad uno dei suoi server.

Scenario

All'avvio di **eMule** il client cercherà di connettersi a uno dei server presenti in una lista salvata in locale nel file `Server.met`. Questi gli restituirà un **Client ID** valido per l'intera durata della connessione, e il cui valore (*ID Alto* o *ID Basso*) condizionerà tutte le fasi successive. Questa fase è detta di *handshake*.

A connessione stabilita il client invierà la lista dei suoi file condivisi al server, che la userà per aggiornare il proprio database per l'indicizzazione dei file e delle fonti.

Quando il client vuole effettuare la ricerca di un certo file sulla rete **eDonkey** invierà una richiesta al server a cui è connesso; questi consulterà il suo database per ottenere l'elenco dei client che condividono quel file (le *fonti*), che poi comunicherà al client perché vi possa stabilire delle connessioni dirette.

Mentre le operazioni descritte finora avvengono tutte attraverso connessioni TCP, vi sono due tipi di interazione tra client e server che prevedono lo scambio di pacchetti UDP. Queste comunicazioni hanno lo scopo di migliorare la ricerca dei file e delle fonti, e di verificare che i server in lista in `Server.met` siano ancora validi e attivi.

Riassumendo graficamente le varie interazioni si ottiene lo schema riportato in *`Figura 1`*.

Nei prossimi paragrafi studieremo nel dettaglio le operazioni descritte nelle righe precedenti, facendo continui riferimenti al codice e chiarendo i passaggi con l'ausilio di diagrammi di flusso.

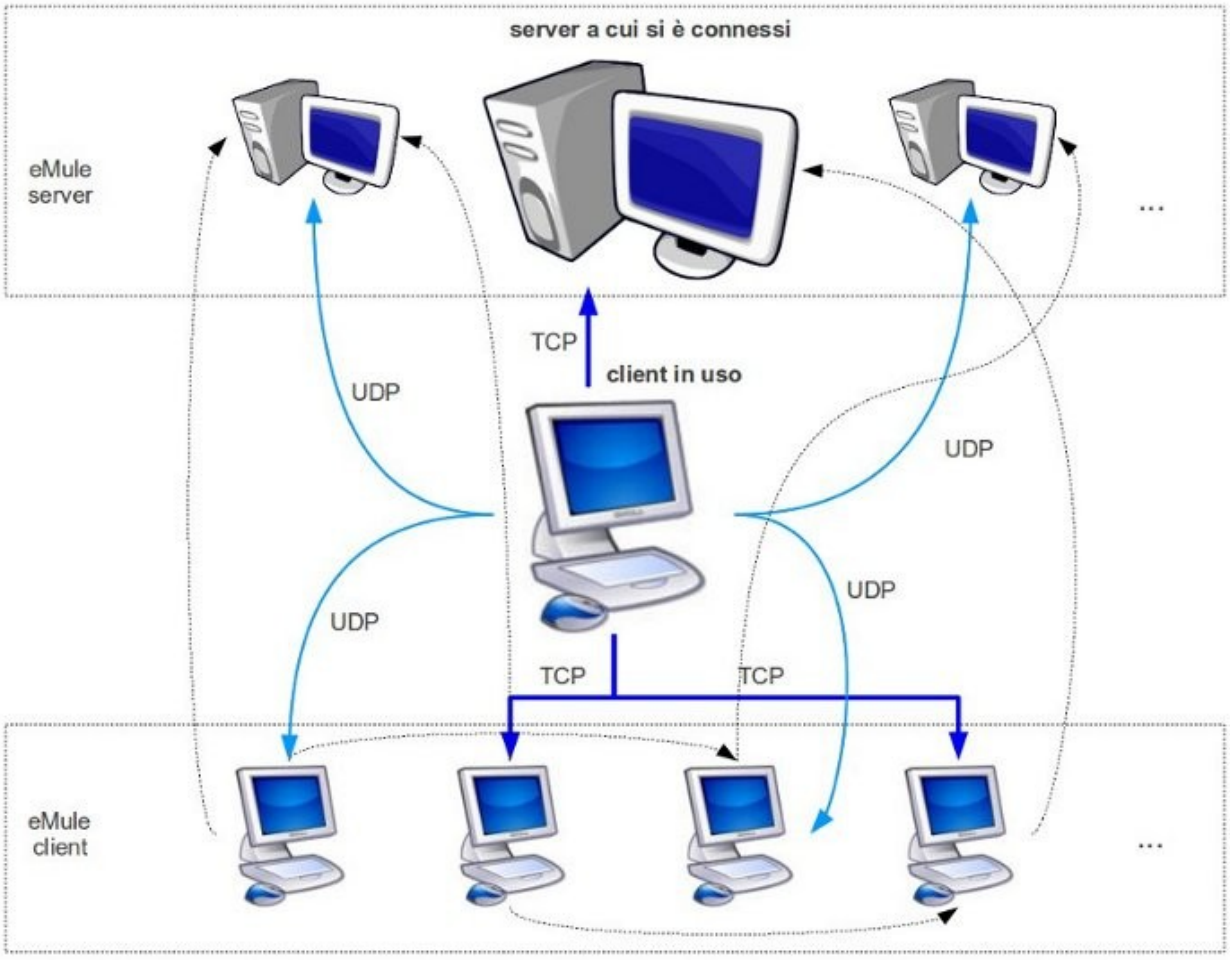

Figura 1: interazioni client-server

Connessione al server

Gran parte della fase iniziale di comunicazione tra client e server è orchestrata dal metodo `ConnectionEstablished()` definito nel sorgente **`sockets.cpp`**, che prevede l'invio di diversi pacchetti TCP che saranno processati dal metodo `ProcessPacket()` definito in **`ServerSocket.cpp`**.

La prima operazione da compiere per un client **eMule** per accedere alla rete è aprire un *socket* di connessione con un server scelto tra quelli elencati nel file locale `Server.met`. Questa lista è completamente configurabile: il client può aggiungere o rimuovere voci, cambiarne le priorità o renderne "statiche" alcune, ovvero fare in modo che l'applicazione non possa toglierle automaticamente dall'elenco (ad esempio perché inattive da troppo tempo). In generale non importa a quale server si

connette il client, perché come vedremo non sarà l'unico canale di reperimento delle fonti. Tuttavia negli ultimi anni sono nati un numero considerevole di *bad server* che cercano di distruggere la rete **eD2k** o guadagnare facendo spam, per cui è buona norma utilizzare una lista server fidata (scaricandola da siti web affidabili o compilandola manualmente aggiungendo solo i server di propria conoscenza).

Se l'operazione di connessione a un server viene conclusa con successo, il client richiede al server di instaurare una nuova sessione inviandogli il pacchetto di richiesta login OP_LOGINREQUEST. Questo contiene una serie di informazioni del client tra cui il suo *UserHash*, il suo indirizzo IP, la porta che sta utilizzando e una serie di tag aggiuntivi (nickname dell'utente, versione del client **eMule**, versione di **eDonkey**, funzionalità supportate).

L'**UserHash** è una chiave di 16 byte (128 bit) che identifica univocamente il client all'interno della rete **eD2k**. Il suo valore è calcolato con un algoritmo di hashing MD5 dal primo server con cui si connette con successo, e rimane invariato per tutte le sessioni di collegamento successive. Si noti che i 16 byte non sono completamente casuali, ma al sesto byte è assegnato il valore 14 e al quindicesimo il valore 111; in questo modo si comunica implicitamente che il client utilizzato è **eMule**. L'*UserHash* viene memorizzato nel file `preferences.dat` (in `emule/config`) e riveste un ruolo determinante nel sistema dei crediti e delle amicizie, motivo per cui è importante prevenire il rischio di appropriamenti indebiti. Tale protezione è opzionale e prevede l'utilizzo combinato dei file `preferences.dat` e `cryptkey.dat`. Il suo funzionamento sarà illustrato nel capitolo *Identificazione Sicura degli Utenti*.

Ritornando all'operazione di connessione, quando il server riceve il pacchetto OP_LOGINREQUEST proverà a stabilire una connessione TCP/IP sulla porta di ascolto del client. L'operazione può concludersi in tre modi:

(1) il client è raggiungibile sulla porta indicata: il server gli assegna un **ID Alto** durante la fase di handshake (il significato di *ID* sarà illustrato fra breve);

(2) il client non è raggiungibile (per vari motivi): il server gli assegna un **ID Basso** durante la fase di handshake;

(3) la connessione viene rifiutata.

Indipendentemente dall'esito dell'operazione precedente vengono ritornati al client i seguenti pacchetti:

- OP_SERVERMESSAGE, che contiene un messaggio di dimensioni variabili che comunica il successo o il fallimento della connessione;
- OP_SERVERSTATUS, che riporta il numero di client connessi al server e il numero totale di file che condividono.

Se poi la connessione ha avuto successo (casi (1) e (2)), viene inviato un terzo pacchetto in cui si comunica al client l'ID assegnatoli dal server: OP_IDCHANGE.

Per ID si intende il **Client ID**, ovvero il secondo tipo di identificativo usato da **eMule** insieme all'*UserHash*. Si tratta di una chiave di 4 byte assegnata dal server con cui il client si è connesso con successo, ed è valido per la sola durata della connessione corrente. Il suo scopo è fornire le informazioni necessarie per una comunicazione corretta e sicura tra client, e per rilevare la presenza di firewall che ostacolano il funzionamento corretto dell'applicazione. Come già anticipato, il *Client ID* può essere di due tipi:

- *ID Alto*, assegnato quando il server riesce a stabilire una connessione col client sulla porta segnalata nel pacchetto OP_LOGINREQUEST (di default la 4662). Il suo valore è calcolato a partire dall'indirizzo IP del client, con cui esiste una corrispondenza biunivoca;
- *ID Basso*, assegnato quando non si verificano le condizioni precedenti. Ha un valore casuale inferiore a 16777216, che non essendo in alcun modo ricavato dalla traduzione dell'IP pubblico del client non permetterà a quest'ultimo di comunicare direttamente con gli altri, ma dovrà farlo tramite server.

Il valore del *Client ID* ha senso solo per distinguere tra *ID Alti* e *ID Bassi*: confrontare due *ID Alti* (o due *ID bassi*) non ha alcun senso, perché il maggiore (minore) dei due non avrà alcun vantaggio (svantaggio) rispetto all'altro.

Se un client ottiene un *ID Basso* potrà comunque scaricare e inviare file, ma con alcuni handicap:

- alcuni server (come ad esempio i *Lugdunum*, i più diffusi) limitano il numero

degli utenti con *ID Basso* che possono connettersi ad essi, o addirittura ne impediscono la connessione. Per questo motivo un *ID Basso* dovrebbe privilegiare i server con pochi utenti, perché avrà maggiori probabilità di rientrare in quel limite;

- due client con *ID Basso* non possono in alcun modo connettersi tra loro.

Il diagramma in *Figura 2* riassume l'intero processo di richiesta di connessione al server. Si noti che i metodi con cui quest'ultimo implementa le varie fasi non sono noti, perché - al contrario dei client - è generalmente a codice chiuso.

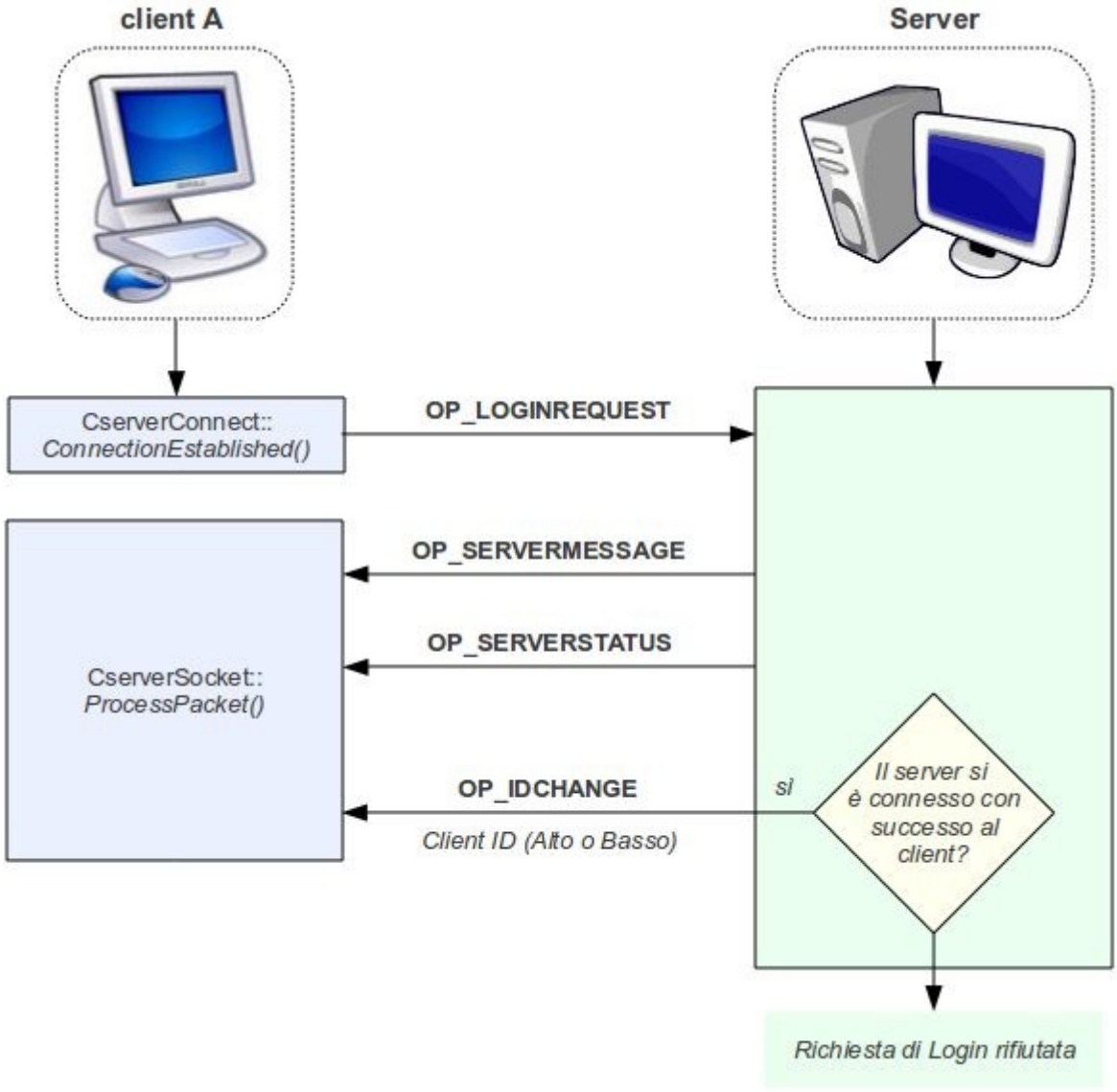

Figura 2: connessione al server

Configurazione connessione

Una volta stabilita con successo una connessione, client e server si scambiano una serie di messaggi di configurazione finalizzati all'aggiornamento dei reciproci stati sulla rete. I passaggi principali sono quello in cui il client invia la propria lista di file condivisi e quello in cui il server aggiorna la lista dei server del client.

I file sorgenti principalmente coinvolti in questa fase e in quelle successive sono:

- **`socket.cpp`**, da cui sarà ripreso il metodo `ConnectionEstablished()`, che continuerà a fare da orchestratore del processo;
- **`SharedFileList.cpp`**, che intuitivamente contiene i metodi responsabili della gestione e dell'invio della lista dei file condivisi dal client;
- **`ServerSocket.cpp`**, in cui è definito il metodo `ProcessPacket()` che gestisce i pacchetti in arrivo sul socket instaurato con il server

Invio della lista dei file condivisi

La compilazione della lista dei file condivisi dal client e il suo successivo incapsulamento nel pacchetto OP_OFFERFILES avviene lanciando la funzione `SendListToServer()`. Il pacchetto comprende:

- l'IP e la porta del client;
- il numero di file nella lista;
- un'insieme di informazioni per ogni file: il *FileHash*, il nome, il tipo e la dimensione.

Il numero di file messi in condivisione non coincide necessariamente con quelli comunicati nella lista, perché questa è limitata a una soglia di 200.

Queste informazioni contribuiranno ad aggiornare (e/o ampliare) la tabella di routing del server, come mostrato in *`Figura 3`*.

Per completezza va detto che l'invio di OP_OFFERFILES è previsto anche in altri casi: quando la lista dei file condivisi viene aggiornata dal client (il pacchetto potrà contenere sia l'intera lista che i singoli riferimenti ai file aggiunti), o per implementare una sorta di messaggio di *keep-alive* per verificare se il server è ancora attivo (il pacchetto conterrà una lista vuota).

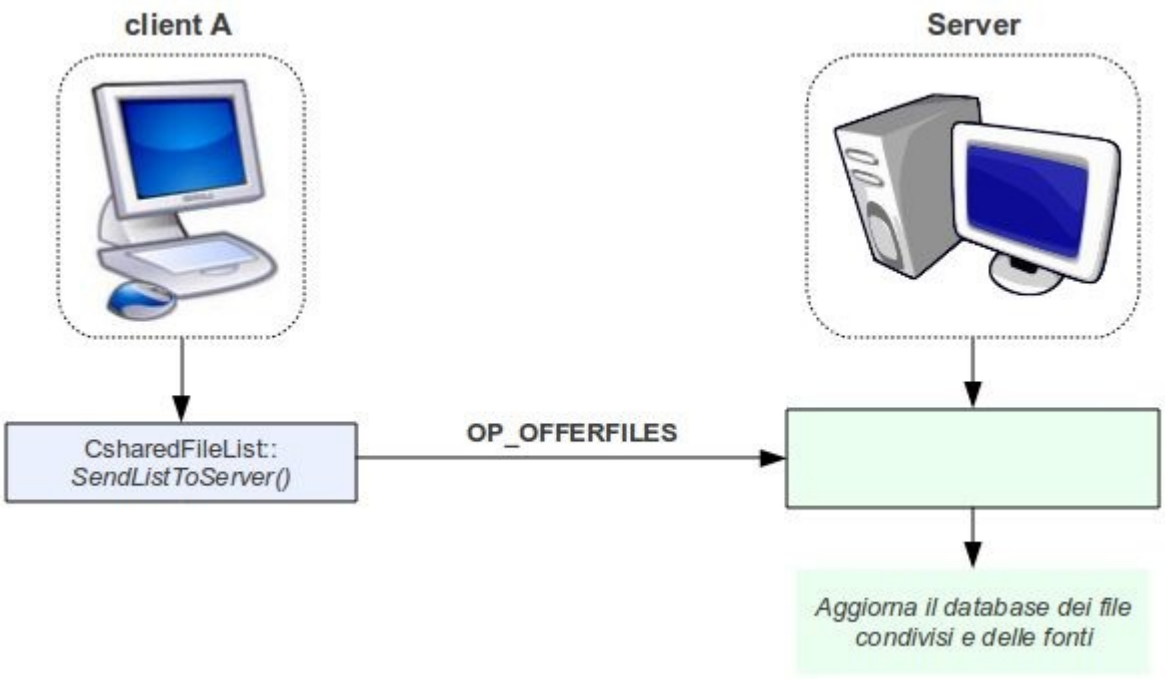

Figura 3: invio della lista dei file condivisi

Aggiornamento lista server

La fase di configurazione della connessione si conclude con la richiesta da parte del client della lista dei server conosciuti dal server con cui è connesso, col fine ovvio di ampliare il numero di server noti. Da notare che questa è l'unica informazione che i server condividono tra loro.

La richiesta viene spedita dal solito `ConnectionEstablished()` e viaggia nel pacchetto OP_GETSERVERLIST, che il server processa e a cui a sua volta risponde con due pacchetti:

- OP_SERVERLIST, la lista richiesta. Per ogni server nell'elenco è riportato il l'indirizzo IP e la porta TCP di comunicazione, utile da conoscere in caso di connessione tramite proxy;
- OP_SERVERIDENT, che contiene alcune informazioni sul server di connessione. Oltre all'indirizzo IP e alla porta ne viene comunicato il nome, un *server hash* (usato per debug) e una breve descrizione.

Il diagramma in *Figura 4* riassume quanto appena descritto.

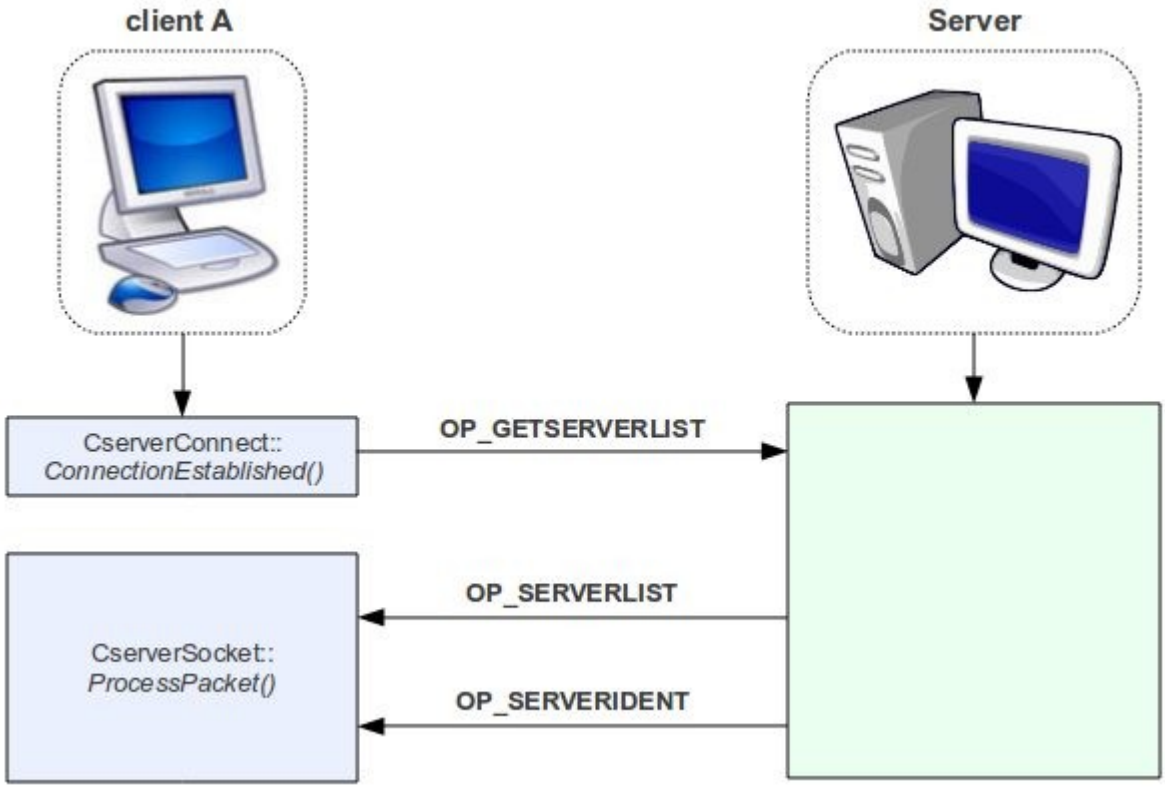

Figura 4: aggiornamento lista server

Ricerca di file e fonti

In questo paragrafo tratteremo la ricerca di un file e delle sue fonti dal punto di vista delle comunicazioni client-server, quindi trascurando altri aspetti come le espressioni logiche ammesse o gli altri filtri di ricerca implementati.

I sorgenti coinvolti in questa fase sono decisamente numerosi, e i principali sono:

- **`SearchResultsWnd.cpp`**, che contiene i metodi per la ricerca di un file sia via TCP che UDP;
- **`DownloadQueue.cpp`**, che contiene i metodi per la ricerca e gestione delle fonti (anche in questo caso sia via TCP che UDP);
- **`PartFile.cpp`**, il cui metodo `AddSources()` viene usato per aggiungere le fonti a un file;
- **`ServerSocket.cpp`**, in cui è definito il metodo `ProcessPacket()` che gestisce i pacchetti in arrivo sul socket instaurato con il server.

Ricerca via TCP

La ricerca di un file comincia sempre con la chiamata del client al metodo `StartNewSearch()`, a cui vengono passati come parametri i criteri di ricerca del file (nome, tipo, ...) sotto forma di espressione. Da qui la ricerca viene propagata sia alla rete **Kad** che a quella **eD2k**, e i risultati convergeranno e saranno visualizzati nella finestra "*Cerca*" della GUI dell'applicazione.

La ricerca nella rete **eDonkey** è implementata in `DoNewEd2kSearch()`, che pacchettizza la richiesta in OP_SEARCHREQUEST e la invia al server. Quest'ultimo effettua una ricerca nel proprio database e ritorna i risultati nel pacchetto OP_SEARCHRESULT. La risposta non è altro che una lista dei file conformi ai criteri di ricerca, e per ognuno di essi deve essere indicato:

- il *FileHash* che lo identifica univocamente nella rete;
- l' indirizzo IP e la porta della sua fonte;
- una serie di etichette descrittive, molte delle quali opzionali. Tra queste citiamo: nome, dimensione, tipo e formato, numero di fonti disponibili, lunghezza brano, bitrate, ...

Una volta scelto il file da scaricare, il client lo passa come parametro al metodo `AddSearchToDownload()`, aggiungendolo alla lista dei file per cui è in coda. La ricerca delle fonti è invece affidata a `Process()` e `ProcessLocalRequest()`, che si occupano della creazione e dell'invio del pacchetto di richiesta OP_GETSOURCES, in cui viene inserito il *FileHash* del file interessato. Il motivo per cui non viene usato il nome del file come identificativo è semplice: poiché il client potrebbe rinominare un suo file condiviso in un qualsiasi momento, si auto-destituirebbe come sua fonte valida nonostante lo sia a tutti gli effetti. Per questo motivo gli si preferisce il *FileHash*, che dipende solo dal contenuto e dalla dimensione del file.

La risposta del server viene incapsulata nel pacchetto OP_FOUNDSOURCES, da cui il client recupera la lista delle fonti, quindi l'indirizzo IP e la porta di ogni client che condivide un file col *FileHash* indicato. Per limitare l'impatto di centinaia di ricerche contemporanee sul carico di lavoro dei server, il numero di fonti comunicate nella risposta è limitato.

Il processo di ricerca viene ripetuto ogni 20 minuti e per massimo 15 file alla volta. Lo scopo per cui sono stati fissati questi valori di soglia è limitare il traffico di *overhead* TCP/IP.

Le fonti ricevute nel pacchetto OP_FOUNDSOURCES vengono infine associate al file con l'invocazione del metodo `AddSources()`.

Ricerca via UDP

Il sistema di ricerca di un file può essere migliorato sovrapponendo a quello che avviene via TCP un secondo che agisce attraverso il protocollo UDP. Il metodo descritto finora è anche chiamato *ricerca locale*, mentre quello che descriveremo nelle prossime righe è detto *ricerca globale*. La ricerca locale è più rapida perché avviene esclusivamente sul server con cui si è connessi, mentre quella globale impiega più tempo ma restituisce più risultati (in termini di fonti valide).

Il formato di richiesta di una ricerca globale è praticamente identico al precedente, ma viene incapsulato nei pacchetti OP_GLOBSEARCHREQ, OP_GLOBSEARCHREQ2 o OP_GLOBSEARCHREQ3 a seconda di alcune caratteristiche, per poi essere infine inviato a tutti i server presenti nel file `Server.met` del client (*Query Server Flooding*).

I server rispondono solo se hanno risultati positivi da comunicare, e lo fanno incapsulando le fonti note nel solito pacchetto OP_SEARCHRESULT. Questo sistema di ricerca è opzionale, avviene attraverso il protocollo UDP, è implementato nel metodo `OnTimer()` e si attiva ad intervalli di tempo regolari.

Anche la ricerca della fonti può essere potenziata dall'utilizzo del canale UDP. Se il client è abilitato, quando rileva che il numero di fonti per un certo file è inferiore a una soglia configurabile, invia un pacchetto UDP di richiesta (OP_GLOBGETSOURCES) a tutti i server presenti nella sua lista locale. I pacchetti possono essere inviati ogni secondo fino a un massimo di 35 richieste per server nell'intervallo di tempo; ciò rende la ricerca di fonti una delle parti più consistenti del traffico UDP generato dal client. Il formato del pacchetto è molto simile alla sua controparte TCP, così come quello di risposta (OP_GLOBFOUNDSOURCES) inviato dai server se la ricerca ha avuto successo. Il loro invio è affidato al metodo `SendGlobGetSourcesUDPPacket()`.

In *Figura 5* sono riportati tutti i passaggi della procedura di ricerca file, sia TCP che

UDP; mentre nella *Figura 6* vengono mostrati quelli di ricerca fonti.

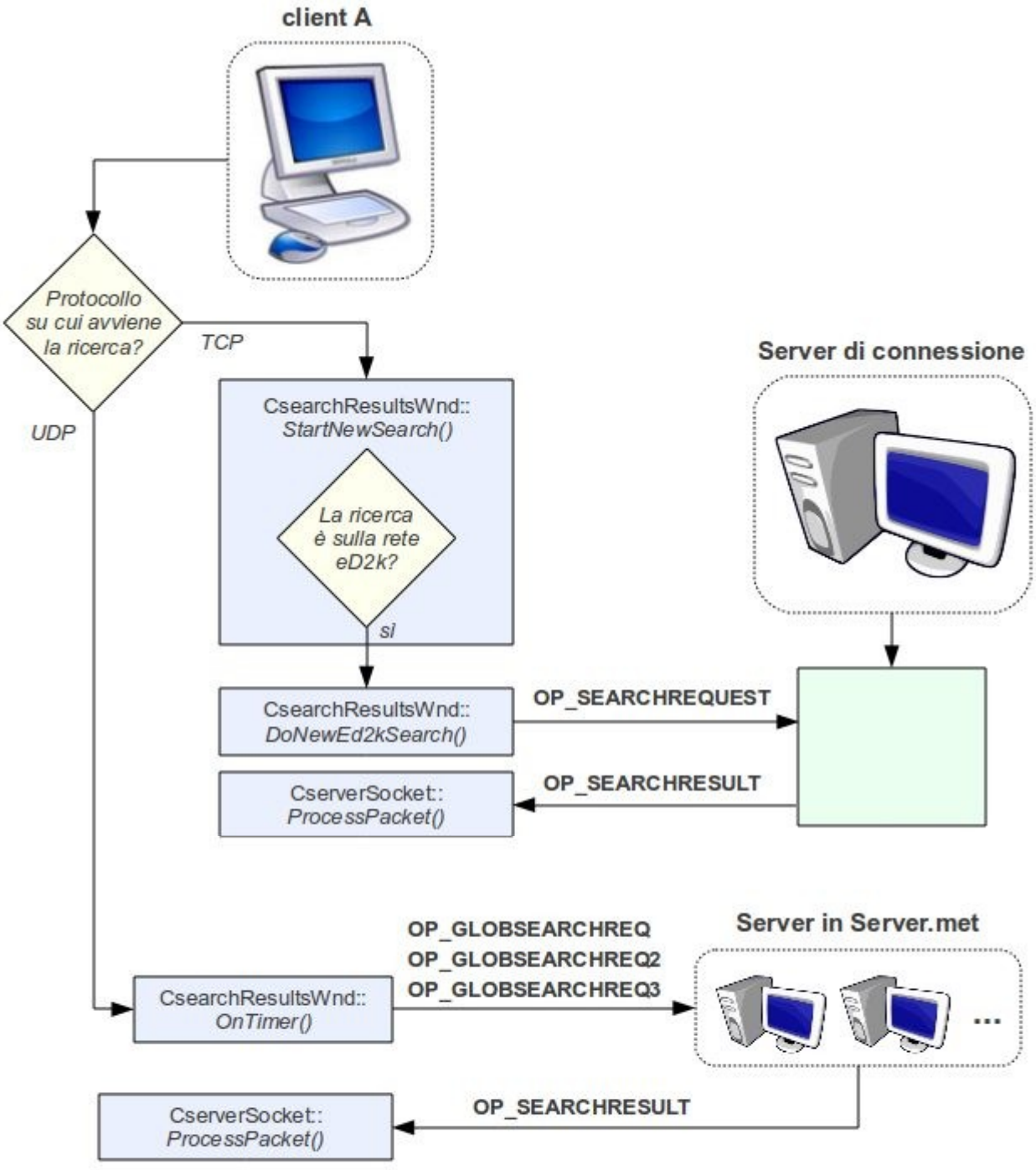

Figura 5: ricerca file via TCP e UDP

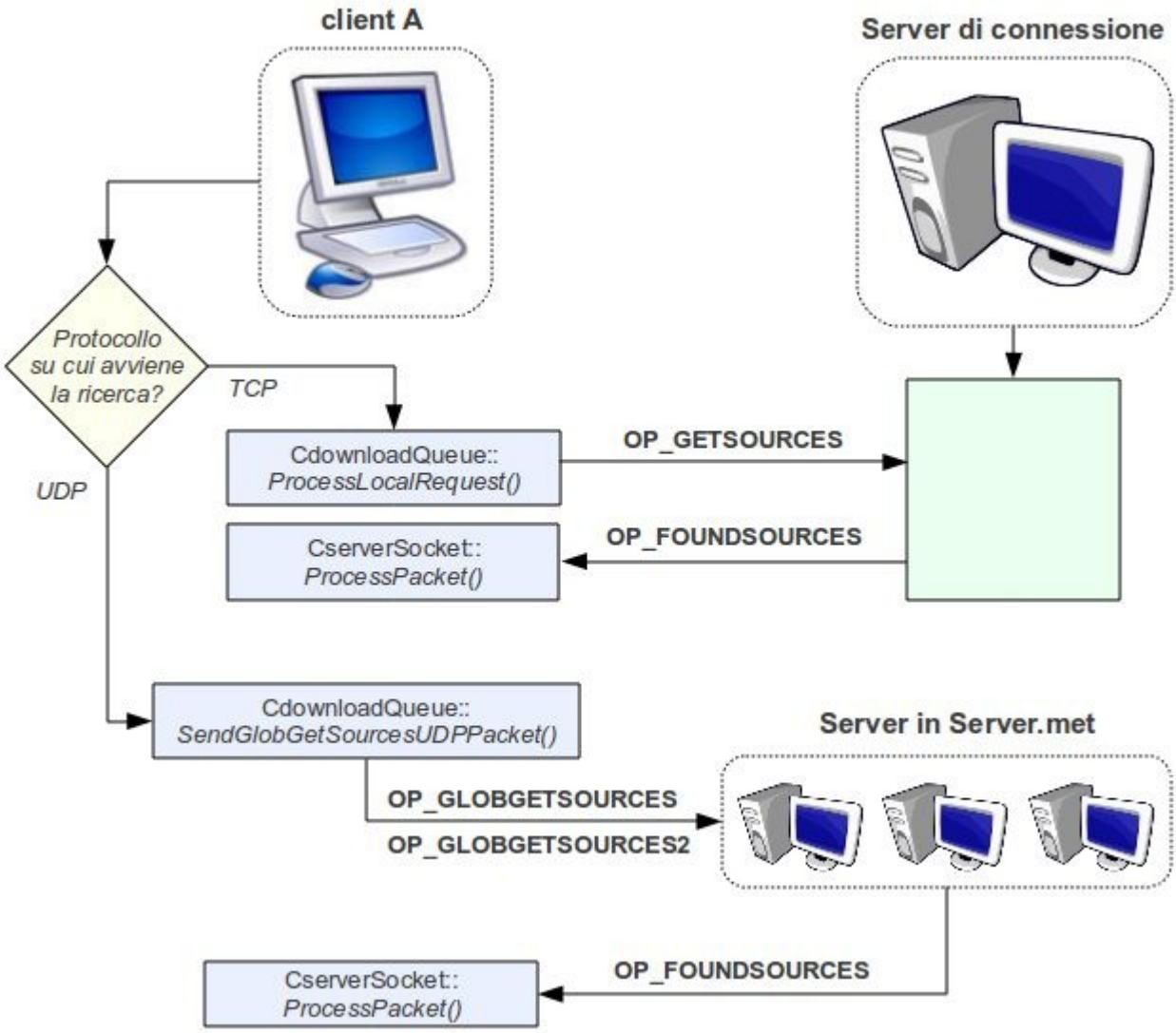

Figura 6: ricerca fonti via TCP e UDP

Meccanismo di Callback

I client che ottengono un *ID Basso* durante la fase di connessione al server non possono essere contattati direttamente da altri client, perché la loro porta di comunicazione è chiusa. Il **meccanismo di Callback** permette a questi client di accettare comunque connessioni in entrata e quindi condividere i loro file con gli altri.

Consideriamo un client A e un client B connessi allo stesso server, rispettivamente con *ID Alto* e *ID Basso.* Il *meccanismo di Callback* si articola nel modo seguente:

1. A cerca un file, e il server gli risponde che B è una fonte con *ID Basso*;
2. A invia al server un pacchetto OP_CALLBACKREQUEST, chiedendo che sia B a mettersi in comunicazione con lui (A non lo può fare perché non può ricavare l'indirizzo dell'altro). La creazione e l'invio del pacchetto è implementata nel metodo `TryToConnect()` (implementato in **`BaseClient.cpp`**);
3. il server verifica che B sia ancora connesso con lui. Se sì gli inoltra il pacchetto OP_CALLBACKREQUESTED, che contiene l'indirizzo IP e la porta di A; altrimenti invia ad A il pacchetto OP_CALLBACK_FAIL che si limita a comunicare il tentativo fallito;
4. se B era connesso, una volta ricevuto il pacchetto OP_CALLBACKREQUESTED proverà ad instaurare una connessione con A.

In *Figura 7* è riportato uno schema riassuntivo ad alto livello, che considera due client A e B (rispettivamente con ID Alto e ID Basso) connessi allo stesso server.

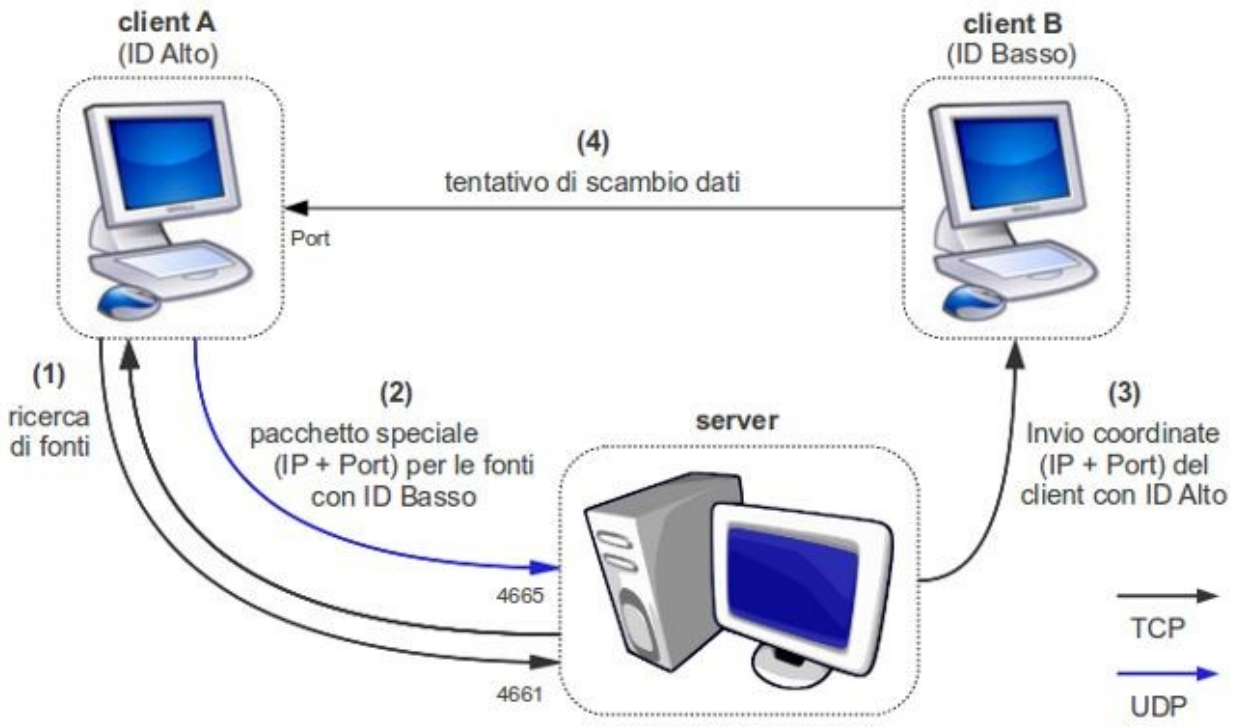

Figura 7: meccanismo di Callback

Perché il meccanismo funzioni il client che lo richiede deve avere un *ID Alto*, o l'altro non saprebbe comunque come contattarlo; questo è il motivo per cui due client con *ID Basso* non potrebbero in alcun modo comunicare tra loro.

Verifica dello stato dei server

L'ultimo tipo di interazione prevista tra client e server nella rete **eDonkey** è quella attraverso cui il client verifica che il server sia ancora attivo. Il metodo responsabile dell'operazione è `ServerStats()` (definito in **`ServerList.cpp`**), che invia periodicamente (di solito ogni 5 secondi) il pacchetto UDP OP_GLOBSERVSTATREQ a uno dei server salvati nella sua lista locale. Il pacchetto include un *challenge* casuale di 4 byte che il server deve includere nella sua risposta (incapsulata in OP_GLOBSERVSTATRES). Se il *challenge* ricevuto è uguale a quello inviato allora il server è considerato ancora attivo. In caso di fallimento il client incrementa invece di uno il contatore dei fallimenti associati ad ogni server: se il suo valore supera una certa soglia allora viene considerato "morto" e viene rimosso dalla lista in `Server.met`.

Interazioni Client – Client

Le interazioni dirette tra client rappresentano l'obiettivo e l'attività principale di un'applicazione *peer-to-peer* orientata alla condivisione di file come **eMule**.

Scenario

Due client possono interagire tra loro solo se sono già connessi ad un server, non necessariamente lo stesso. Si tratta di un requisito fondamentale per accedere alla rete **eD2k** e i suoi servizi, ivi compresi quelli di condivisione e scambio file.

Ogni client gestisce le richieste di trasferimento dati (*upload*) attraverso l'utilizzo di *code*, in cui inserisce tutti quei client per cui rappresenta una fonte. Se quando gli arriva una richiesta ha la coda vuota, allora comincerà a trasmettere dati; altrimenti aggiungerà il client in una certa posizione della coda.

Quando un client raggiunge la cima della coda di una sua fonte, instaura con essa una connessione temporanea durante la quale gli viene inviato una o più parti del file desiderato.

Un client può scaricare lo stesso file da fonti diverse, dunque può trovarsi contemporaneamente in molte code di attesa.

Per incoraggiare la condivisione **eMule** implementa un *sistema di crediti* che accorcia i tempi di attesa per i client che condividono di più.

Connessione tra client

Una volta creato il socket tra i client, la connessione vera e propria avviene solo in seguito a un *handshake* iniziale.

I principali file sorgenti coinvolti in questa fase sono:

- **`BaseClient.cpp`**, in cui sono definiti i metodi per la creazione e l'invio dei messaggi di *handshake*;
- **`ListenSocket.cpp`**, che contiene i due metodi per la ricezione dei pacchetti,

sia quelli scritti secondo il protocollo **eDonkey** standard che quelli del *protocollo esteso di* ***eMule.***

Il primo messaggio ad essere inviato viene incapsulato nel pacchetto OP_HELLO, in cui il client scrive alcune informazioni che lo riguardano:

- il suo *UserHash*;
- il suo *Client ID* e la porta di connessione;
- l'indirizzo IP e la porta del server a cui è attualmente connesso;
- una serie di campi aggiuntivi che indicano alcune proprietà, come il nickname assegnato dall'utente o la versione del client.

Il pacchetto viene generato all'interno del metodo `SendHelloTypePacket()`, e successivamente inviato dalla funzione `SendHelloPacket()`.

Il client remoto che riceve il pacchetto verifica se conosce già il client chiamante, e a seconda dei casi crea una nuova istanza della classe `CUpDownClient` in cui salvare le informazioni contenute in OP_HELLO, oppure aggiorna quelle già in suo possesso. Completata questa operazione controlla se la versione del client chiamante è **eMule** o un'altra versione (ad esempio **eDonkey**):

- nel primo caso invia due pacchetti di risposta, OP_EMULEINFO e OP_HELLOANSWER;
- nel secondo caso risponde solo con OP_HELLOANSWER.

Nei prossimi sotto-paragrafi sarà approfondita la gestione di due di questi pacchetti.

Gestione di OP_EMULEINFO

Il pacchetto OP_EMULEINFO viene creato e spedito da `SendMuleInfoPacket()`, e contiene una serie di campi appartenenti al protocollo esteso di **eMule** coi quali informa l'altro client se supporta alcune caratteristiche avanzate dell'applicativo. Tra queste citiamo ad esempio la compressione dei dati, le comunicazioni UDP, lo scambio fonti, l'identificazione sicura ed altre ancora. Viene chiamato protocollo esteso proprio perché estende le funzionalità del protocollo **eDonkey** su cui si basa. Non a caso il metodo che riceve questo tipo di pacchetti si chiama `ProcessExtPacket()`, per distinguerlo dal `ProcessPacket()` che si prende in carico quelli standard.

OP_EMULEINFO viene processato dalla funzione `ProcessHelloTypePacket()`, che può concludersi in due modi:

- con l'invio di un pacchetto OP_EMULEINFOANSWER (che ha la stessa struttura dell'altro);
- con il *ban* del client remoto, ovvero il suo inserimento in una lista di "indesiderati" che non possono più connettersi al client locale. Uno dei motivi per cui si può essere bannati in questa fase è fallire il processo di identificazione sicura utente.

Gestione di OP_HELLO

Il pacchetto OP_HELLO appartiene al protocollo standard di eDonkey, dunque può essere ricevuto da qualsiasi client compatibile alla rete. La sua gestione prevede la creazione di un pacchetto (OP_HELLOANSWER) da parte della funzione `SendHelloTypePacket()`, in cui vengono sostanzialmente replicate le informazioni contenute nel messaggio a cui risponde. Con il suo invio ad opera del metodo `SendHelloAnswer()` e la sua successiva ricezione, si conclude la fase di *handshake* iniziale: se tutto è andato a buon fine i due client sono ora connessi.

Nella pagina seguente in *Figura 8* è presentato un diagramma riassuntivo dell'intera fase di connessione tra due client.

Riconoscimento crediti e identificazione sicura

Come già anticipato, il meccanismo dei crediti è stato introdotto per incentivare la condivisione dei file. Il principio è che più crediti si vantano presso un client e più velocemente si scalerà la sua coda di upload. Una descrizione dettagliata del suo funzionamento sarà fornito in *Appendice*.

Per evitare che impostori impersonino l'identità di un altro client approfittando dei crediti da lui accumulati, **eMule** introduce il sistema di *Identificazione Sicura Utente* (*SUI*) basato su firma digitale. Anche questo argomento non sarà affrontato in questo capitolo, ma verrà approfondito nel prossimo.

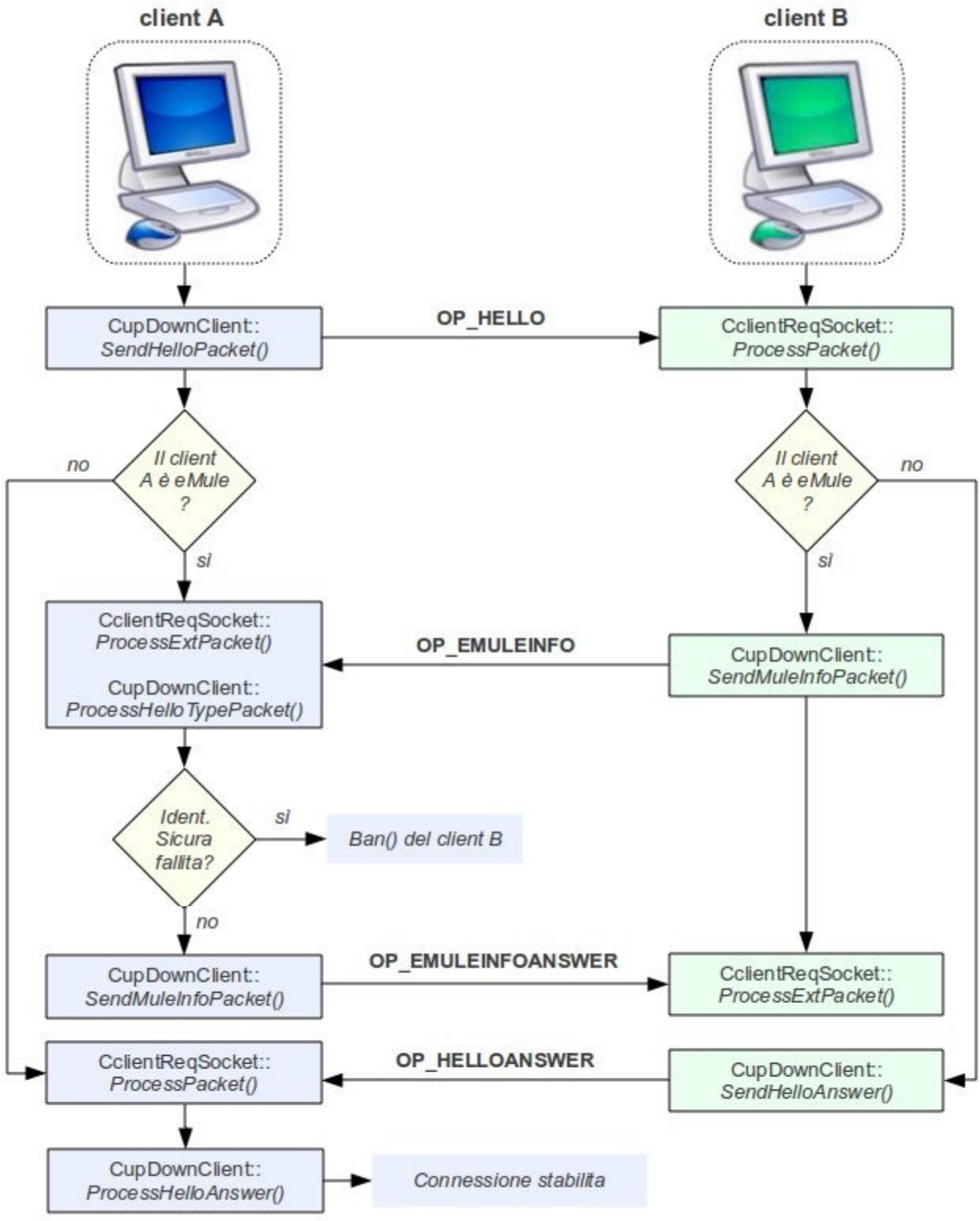

Figura 8: connessione tra client

Richiesta di un file

In questo paragrafo studieremo nella sua interezza il processo di richiesta di un file. Ovviamente siamo nell'ipotesi che il client abbia già stabilito una connessione con la sua fonte, e che questa sia stata individuata come risultato di una ricerca presso un server. Chiameremo il client richiedente A, e la fonte contattata B.

La richiesta di un file prevede lo scambio di numerosi messaggi tra i client, soprattutto se entrambi supportano il protocollo esteso di **eMule**. Gran parte dei pacchetti in cui viaggiano vengono creati e spediti (ma anche processati) all'interno del metodo `SendFileRequest()`, definito nel sorgente **`DownloadClient.cpp`**.

Richiesta base

Il pacchetto OP_REQUESTFILENAME viene generato ad ogni richiesta, e contiene:

- il *FileHash* del file richiesto;
- (opzionale) lo *stato*, ovvero una serie di informazioni generali sul client (ad esempio la versione del protocollo, il supporto alla comunicazione UDP, allo scambio fonti, ...);
- (opzionale) il numero aggiornato di fonti per quel file.

Le parti opzionali dipendono dal supporto o meno al protocollo esteso.

Una volta ricevuto il pacchetto, il client B verifica che il file richiesto sia ancora tra i suoi condivisi:

- in caso negativo informa A inviandogli il pacchetto OP_FILEREQANSNOFIL. Prima di chiudere la connessione, A proverà a chiedere a B se può essere una sua fonte per altri file;
- in caso positivo B invia il pacchetto OP_REQFILENAMEANSWER, in cui comunica il *FileHash* e il nome con cui ha salvato il file in locale. Non appena A lo riceve invierà un nuovo messaggio con la richiesta di upload.

Riassumiamo questo scambio di messaggi nella *`Figura 9`*.

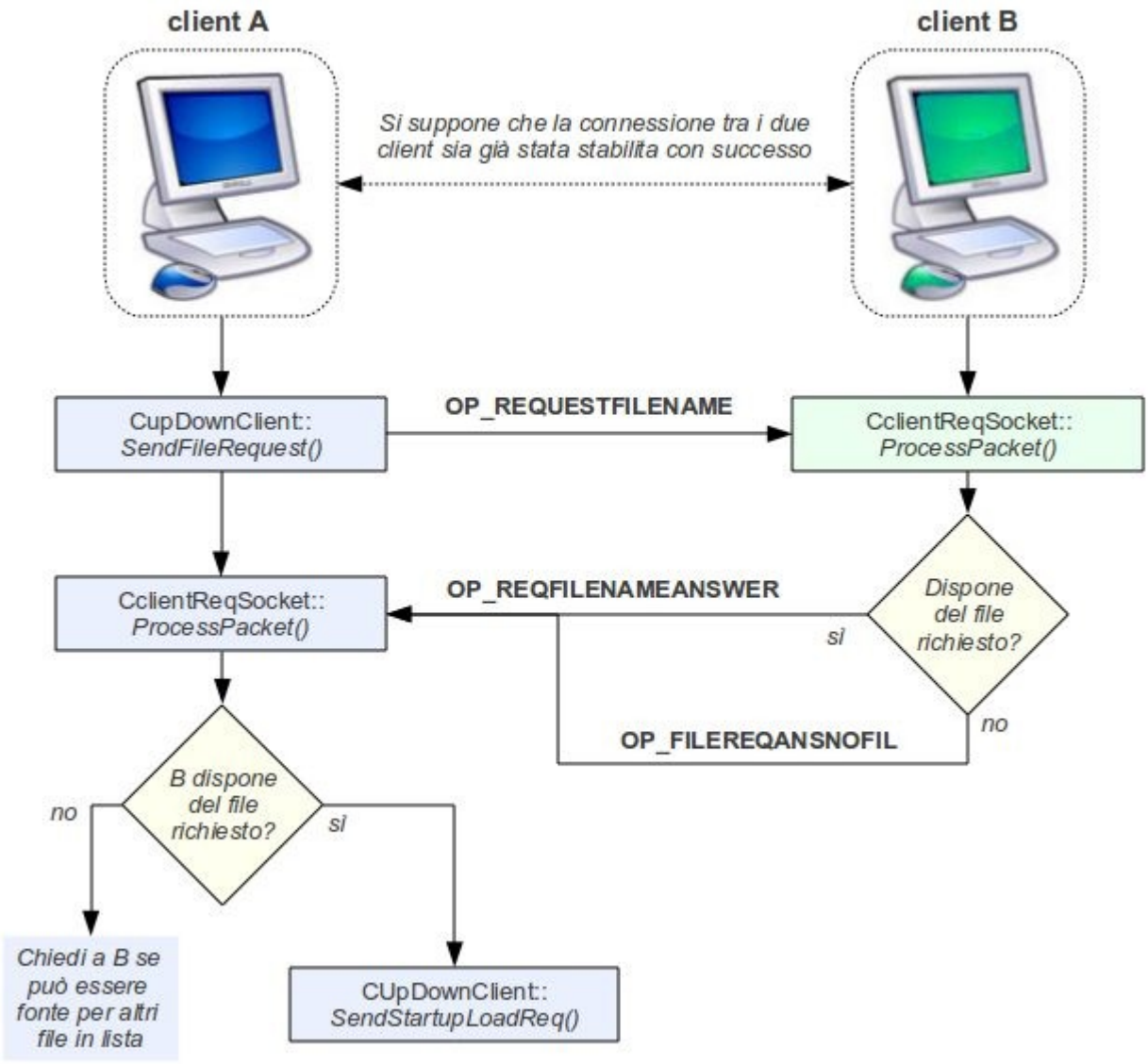

Figura 9: richiesta base

Richiesta per file con più parti

Se il file richiesto è formato da più di una parte, A invia un secondo pacchetto (OP_SETREQFILEID) in cui sostanzialmente riformula la richiesta passando solo il *FileHash*. Ciò che importa infatti non è il suo contenuto, ma il fatto che sblocca l'invio di un nuovo pacchetto di risposta che ottimizza il trasferimento di file multi-parte.

Quando B riceve OP_SETREQFILEID si aprono ancora due alternative:

- B non condivide più il file ed informa A ritornandogli il pacchetto OP_FILEREQANSNOFIL;

- B condivide ancora il file e risponde ad A con il pacchetto OP_FILESTATUS, che include una serie di informazioni aggiuntive che appartengono al protocollo esteso (le stesse che prima abbiamo definito *stato*). Quando A lo riceve invierà un nuovo messaggio a B con la richiesta di upload.

Va sottolineato che l'invio del pacchetto OP_SETREQFILEID non è alternativo a OP_REQUESTFILENAME, ma quando il file è composto da più parti vengono spediti entrambi. Il motivo è garantire la compatibilità con il protocollo standard **eD2k**: dato che il client **eDonkey** tradizionale non gestisce lo *stato*, vengono distinti i due casi.

Illustriamo in *Figura 10* questo secondo tipo di scambio di messaggi.

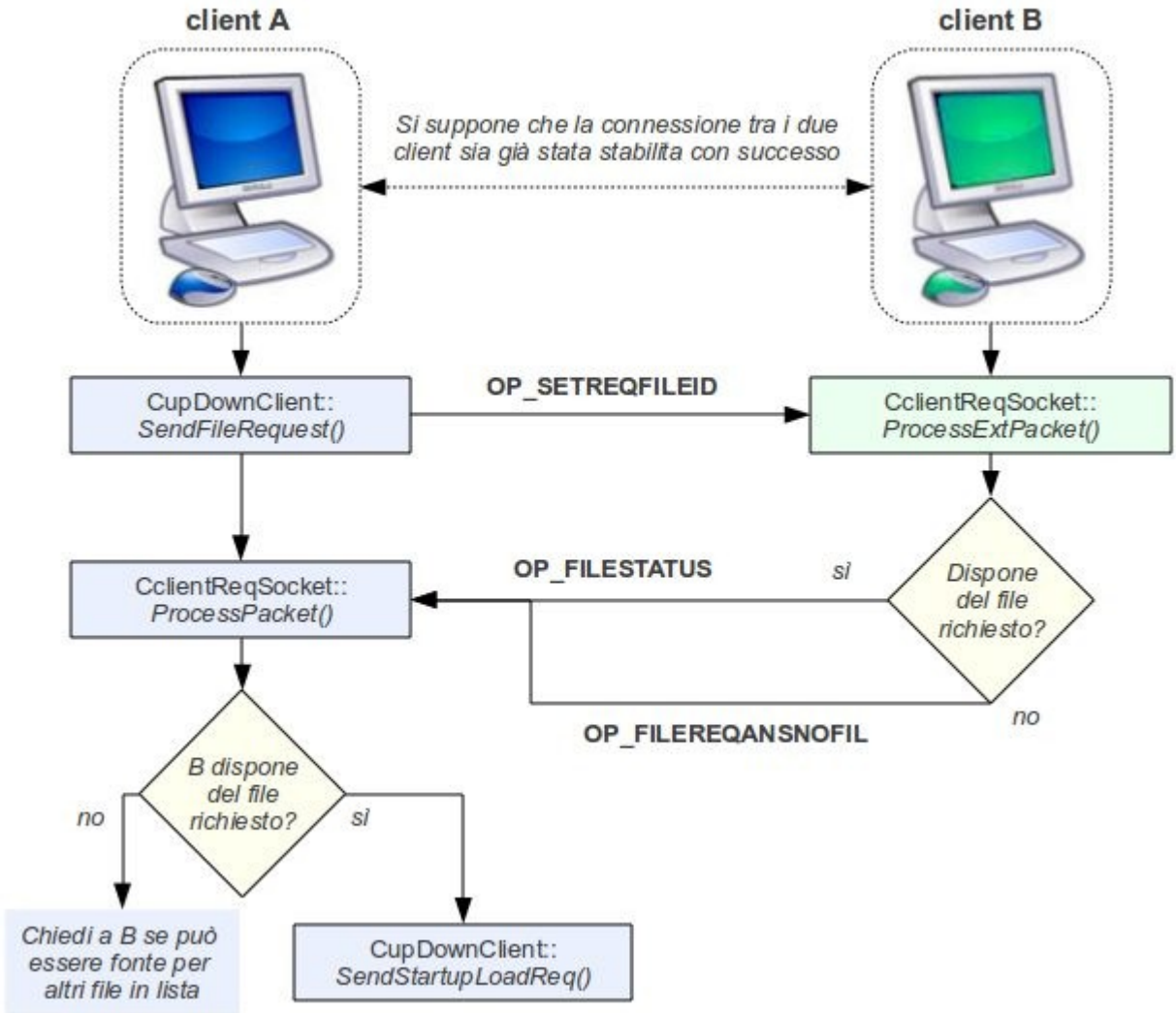

Figura 10: richiesta per file con più parti

Scambio fonti

Gli ultimi pacchetti creati e inviati dal metodo `SendFileRequest()` sono dedicati alla funzionalità di *scambio fonti* tra client. Si tratta di un'ulteriore canale di ricerca fonti che si affianca a quello via server e via **Kad**, e che permette di ottenere una percentuale di fonti contattate su quelle disponibili nelle reti molto prossima al 100%. Bypassare il passaggio dal server si traduce inoltre in un aumento della velocità di reperimento fonti e in un alleggerimento complessivo del carico del sistema.

Esistono due versioni del meccanismo di scambio fonti, ognuna delle quali ha il proprio pacchetto: OP_REQUESTSOURCES e OP_REQUESTSOURCES2. Quest'ultima contiene oltre al *FileHash* del file richiesto una serie di informazioni aggiuntive sulla richiesta.

Quando il client B riceve uno di questi pacchetti controllerà:

1. se è una fonte valida per il file indicato, controllando prima nella lista dei file condivisi e poi in quella dei file in download. In altre parole B si può proporre come fonte sia se dispone del file completo, sia se ne ha già scaricato almeno una parte;
2. se conosce altre fonti per quel file.

In caso di riscontro positivo B creerà il pacchetto OP_ANSWERSOURCES all'interno del metodo `CreateSrcInfoPacket()` implementato in **`BaseClient.cpp`**, scrivendovi:

- il *FileHash* del file richiesto;
- il numero di fonti trovate;
- alcune informazioni per ogni fonte: *Client ID* e porta, IP e porta del server a cui è connessa, *UserHash* (solo per la seconda versione del meccanismo).

Una visione completa del sistema di scambio fonti è illustrata nello schema in *Figura 11*.

Richiesta di upload

La richiesta di upload è la fase che precede quella del trasferimento dati vero e proprio, e prevede l'inserimento in coda del client richiedente.

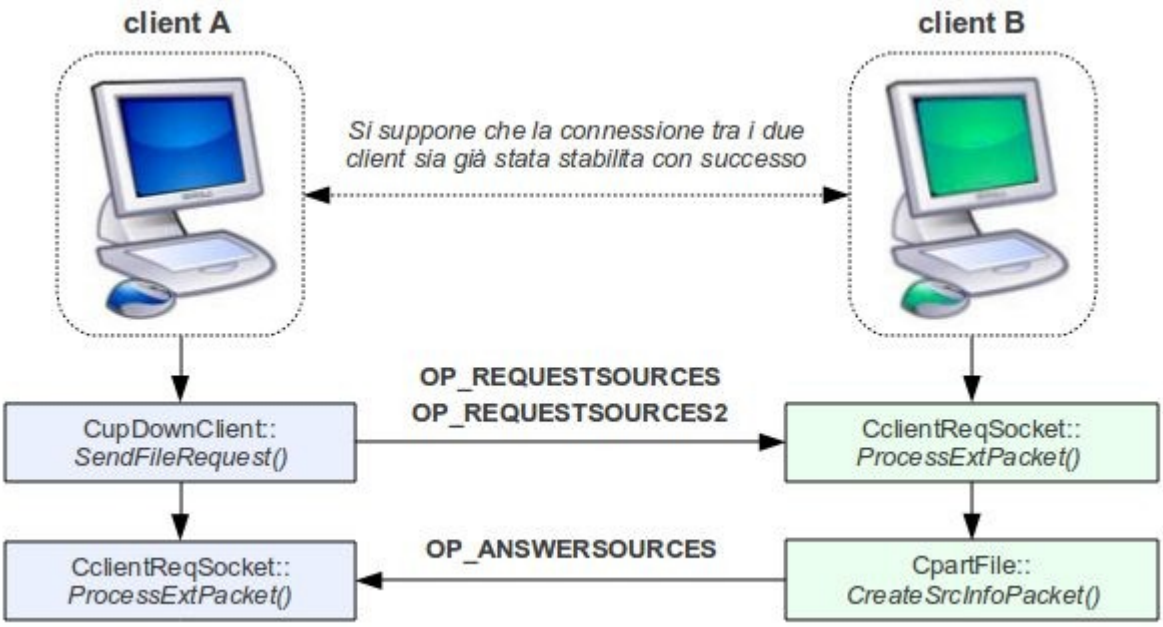

Figura 11: scambio fonti

Il metodo con cui inizia la richiesta di upload è diverso a seconda del file richiesto:

- se ha un'unica parte, la fonte invia il pacchetto OP_REQFILENAMEANSWER che il client processa nel metodo `ProcessFileInfo()`;
- se ha più parti, la fonte invia anche il pacchetto OP_FILESTATUS, che il client A processa grazie al metodo `ProcessFileStatus()` (anch'esso implementato nel sorgente **`DownloadClient.cpp`**).

In realtà ciò che cambia tra i due metodi è ben poco. In entrambi i casi viene anzitutto verificato che il client disponga dell'*Hashset* del file, ovvero l'insieme degli *Hash* delle singole parti di cui è composto (il discorso verrà approfondito in *Appendice*). Se sì richiederà che la fonte avvii l'upload, inviando con `SendStartupLoadReq()` il pacchetto OP_STARTUPLOADREQ; se invece non ne è ancora in possesso spedirà OP_HASHSETREQUEST e il client remoto glielo includerà nel pacchetto OP_HASHSETANSWER.

Prima di passare alla fase di gestione della coda, può essere utile riassumere quanto detto finora nello schema riportato in *Figura 12*.

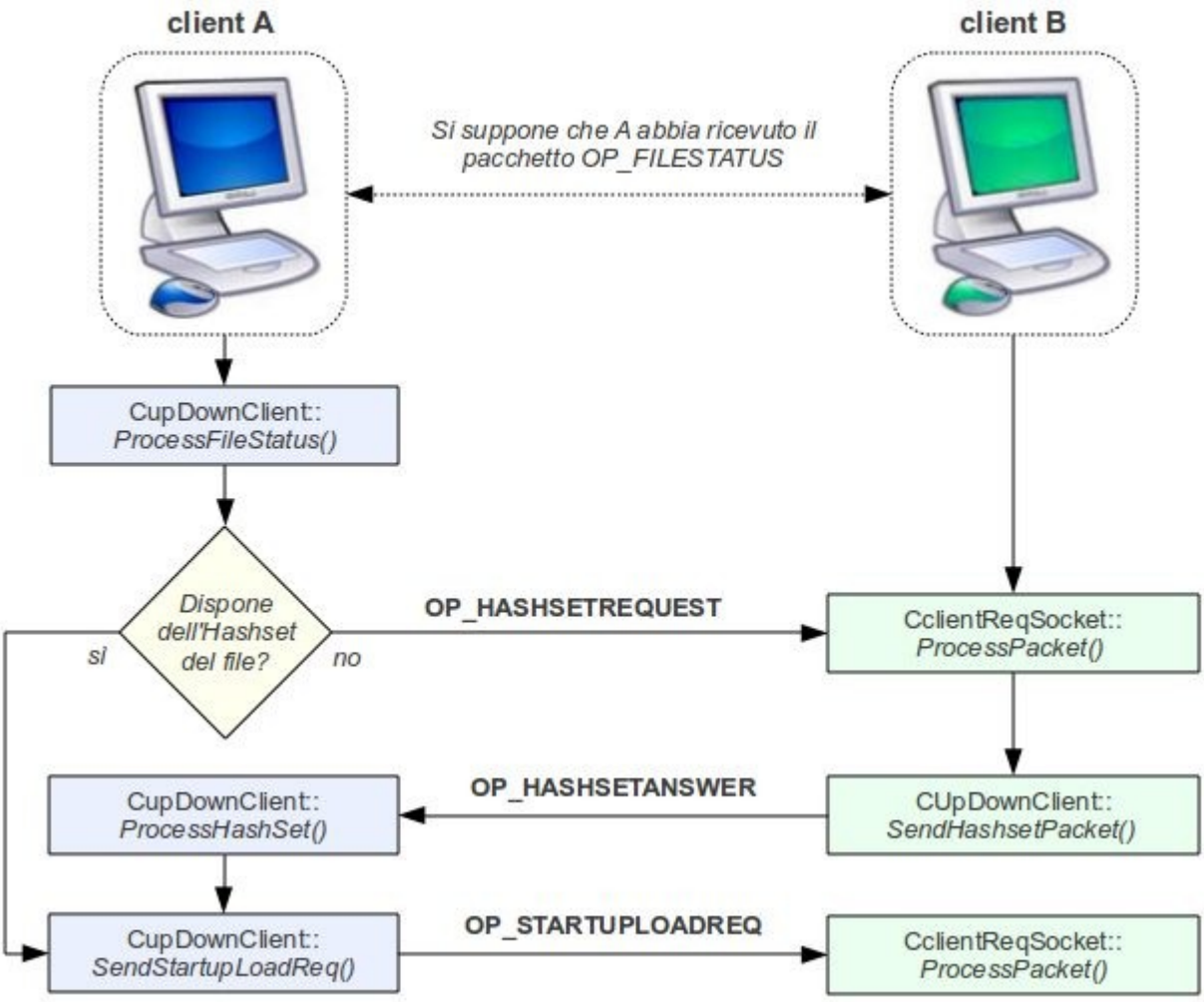

Figura 12: prima fase richiesta di upload

Quando a B arriva il pacchetto di richiesta di avvio upload, se condivide ancora il file richiesto aggiunge A alla sua coda di upload. Tale processo è implementato nel metodo `AddClientToQueue()` definito nel file sorgente **`UploadQueue.cpp`**.

Se la coda non è vuota significa che altri client stanno già scaricando da B. In questo caso invierà ad A il pacchetto OP_QUEUERANKING per riferirgli la sua posizione in coda (*Queue Ranking*). Se invece la coda al momento della richiesta era vuota o se il client ha raggiunto la prima posizione, allora la fonte B gli comunica che è il suo turno di scaricare incapsulando il messaggio in OP_ACCEPTUPLOADREQ.

A questo punto se A è pronto può iniziare a scaricare, altrimenti (ad esempio se ha già ottenuto il file da un'altra fonte) cancellerà il trasferimento.

Le proprietà che caratterizzano una coda sono strettamente legate al sistema dei crediti, e saranno approfondite in *Appendice*. Di seguito riportiamo invece il solito schema riassuntivo.

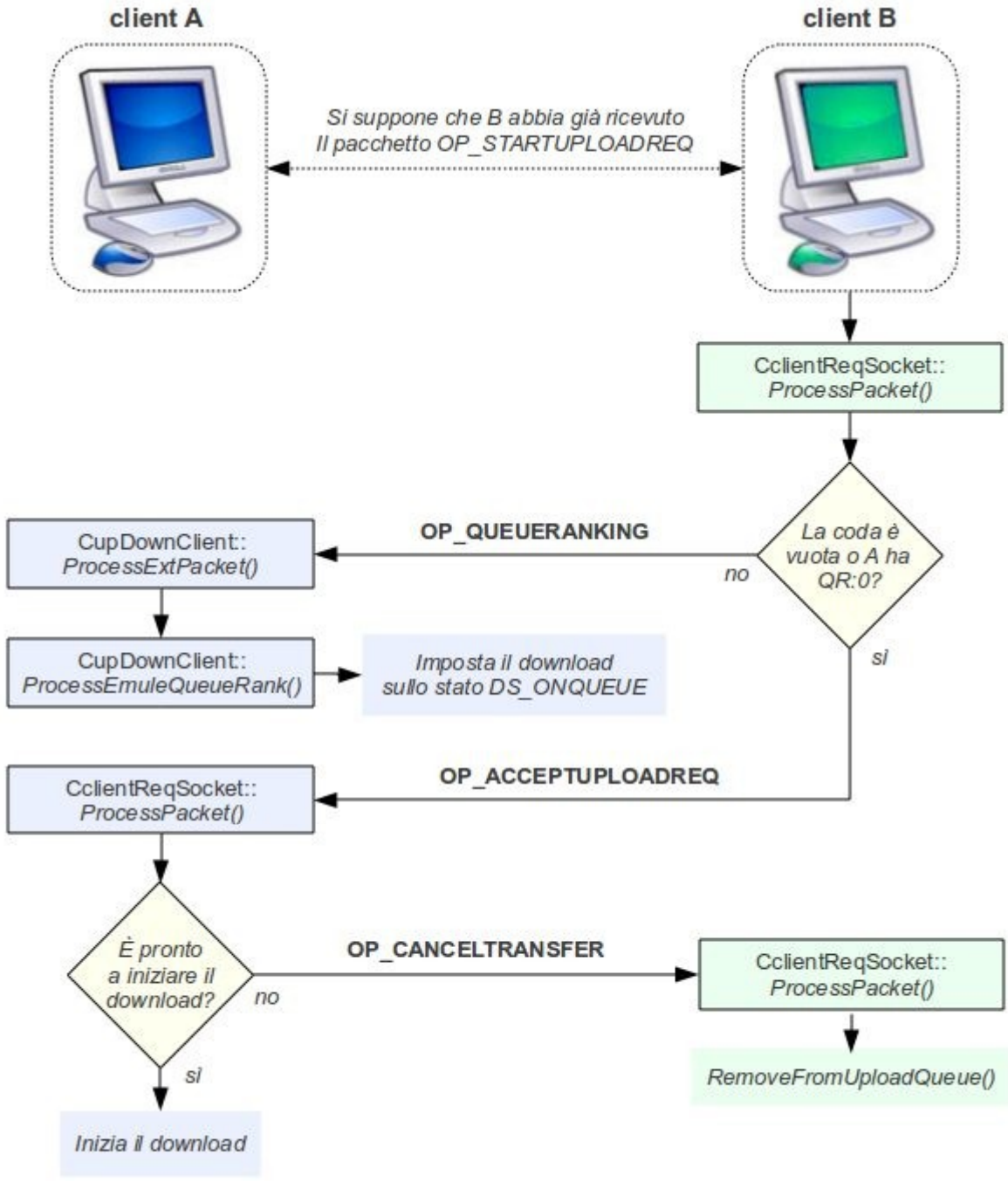

Figura 13: seconda fase richiesta di upload

Trasferimento file

Quando il client A riceve un pacchetto OP_ACCEPTUPLOADREQ da una fonte, se è ancora intenzionato a scaricare quel file invierà una richiesta per tre blocchi. Questi non vengono richiesti in ordine lineare (dal primo all'ultimo), ma a seconda della disponibilità istantanea. Ogni client dà priorità alle parti più rare – quelle con meno fonti – per bilanciare/velocizzare la diffusione globale, e minimizzare la possibilità di avere file incompleti in condivisione. Si possono impostare altre priorità, ad esempio scaricare prima le parti utili per avere un'anteprima del file (tipicamente quelle iniziali e finali) o quelle quasi completate nelle precedenti sessioni di download.

La richiesta viaggia all'interno del pacchetto OP_REQUESTPARTS, che contiene gli offset di inizio e di fine dei tre blocchi interessati. Quando la fonte riceve il pacchetto mette i blocchi indicati in una coda di richiesta.

Non appena si libera uno slot di upload per uno dei client in coda, la fonte B recupera i blocchi in coda di upload per quel client e glieli invia. In particolare, se l'altro client supporta la compressione dei dati glieli invierà nel pacchetto OP_COMPRESSEDPART, altrimenti in OP_SENDINGPART.

Tutta la fase di invio dati è orchestrata dalla fonte, non dal client, attraverso il metodo `CreateNextBlockPackage()` implementato in **`UploadClient.cpp`**. Il client ricevente si limita invece a processare i pacchetti dati nell'ordine in cui arrivano invocando la funzione `ProcessBlockPacket()` (in **`DownloadClient.cpp`**).

Una visione più completa dell'intero procedimento è illustrata in *`Figura 14`*.

Nelle prime versioni di **eMule** i blocchi ricevuti venivano scritti su disco solo una volta scaricati completamente; se quindi il download si interrompeva prima, quanto scaricato fino a quel momento veniva perso. Per ridurre lo spreco di banda è stato introdotto l'utilizzo di un buffer da riempire e svuotare sul disco regolarmente, indipendentemente dalla quantità di dati scaricati. Il buffer non è condiviso ma viene allocato per ogni singolo file, e viene svuotato ogni 60 secondi (configurabili).

Un'altra importante considerazione sulla fase di trasferimento dati è che il limite di traffico in upload di ogni client non dovrebbe essere superiore all'80% della propria capacità, o non ci sarebbe abbastanza banda per la gestione dell'*overhead.*

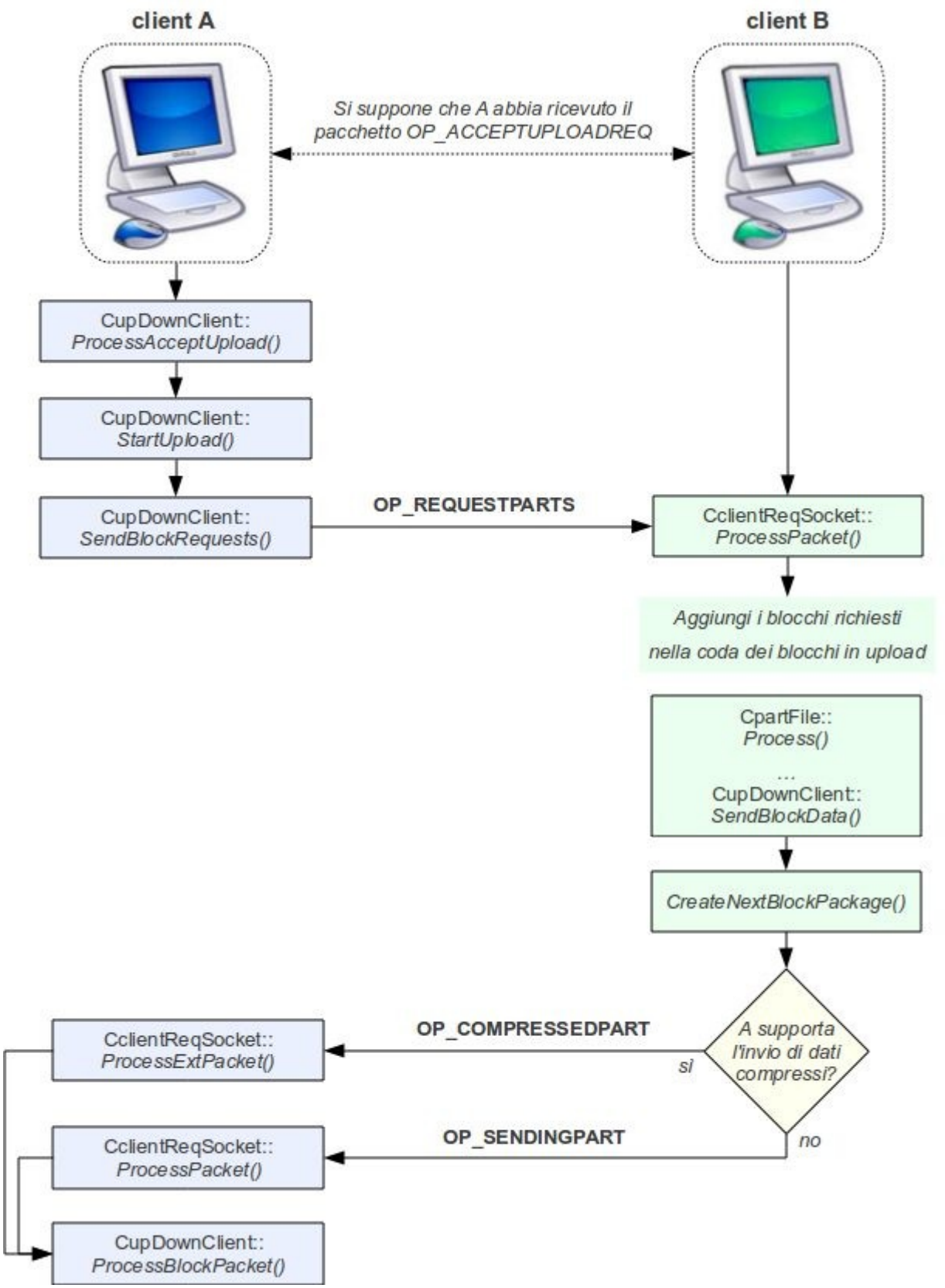

Figura 14: trasferimento file

A questo proposito **eMule** mette a disposizione un meccanismo noto come *Upload Speed Sense* (*USS*), che se abilitato lascia al client il compito di regolare automaticamente tale limite. Il criterio secondo cui agisce l'*USS* è usare tutto l'upload disponibile evitando congestioni e massimizzando la velocità di download e riducendo i tempi di attesa in coda (accumulando più crediti).

Aggiornamento dello stato delle fonti

Durante lo scaricamento di un file il client mantiene in esecuzione un'istanza del metodo `CPartFile::Process()` per ogni parte di file in download. Il suo scopo è aggiornare periodicamente lo stato delle fonti, e non appena si disconnette da queste inviare (ogni circa 20 minuti) il pacchetto UDP OP_REASKFILEPING.

La risposta della fonte può essere di tre tipi:

- OP_FILENOTFOUND, se il file richiesto non è più tra quelli condivisi;
- OP_REASKACK, indica che il client è ancora nella coda della fonte, e viene fornita anche la sua posizione nella stessa;
- OP_QUEUEFULL, se la fonte condivide ancora quel file ma ha troppi client in coda per aggiungere anche quello che ha inviato la richiesta.

Si consideri che in nessuno dei tre pacchetti di risposta è riportato l'identificativo del file a cui fanno riferimento.

Nella pagina seguente in *Figura 15* è riportato il diagramma di flusso che riassume quanto appena detto.

L'intervallo di tempo entro cui la richiesta OP_REASKFILEPING viene reiterata è regolabile dall'interfaccia utente. Accorciando il periodo i contatti diventano più frequenti, ma se da una parte questo aumenta le possibilità di ottenere uno slot dalla fonte, dall'altra può essere interpretato come un comportamento aggressivo, perché ha un serio impatto sul numero di connessioni e sulla banda messa a disposizione dalle fonti. Per questo motivo i client che inviano richieste troppo frequenti possono essere inseriti in una *blacklist*, ovvero una lista di client diffidati a cui non viene più accordato il permesso di stabilire connessioni col client che li ha segnalati. In genere le *blacklist* generate automaticamente vengono azzerate ad ogni riavvio di **eMule**.

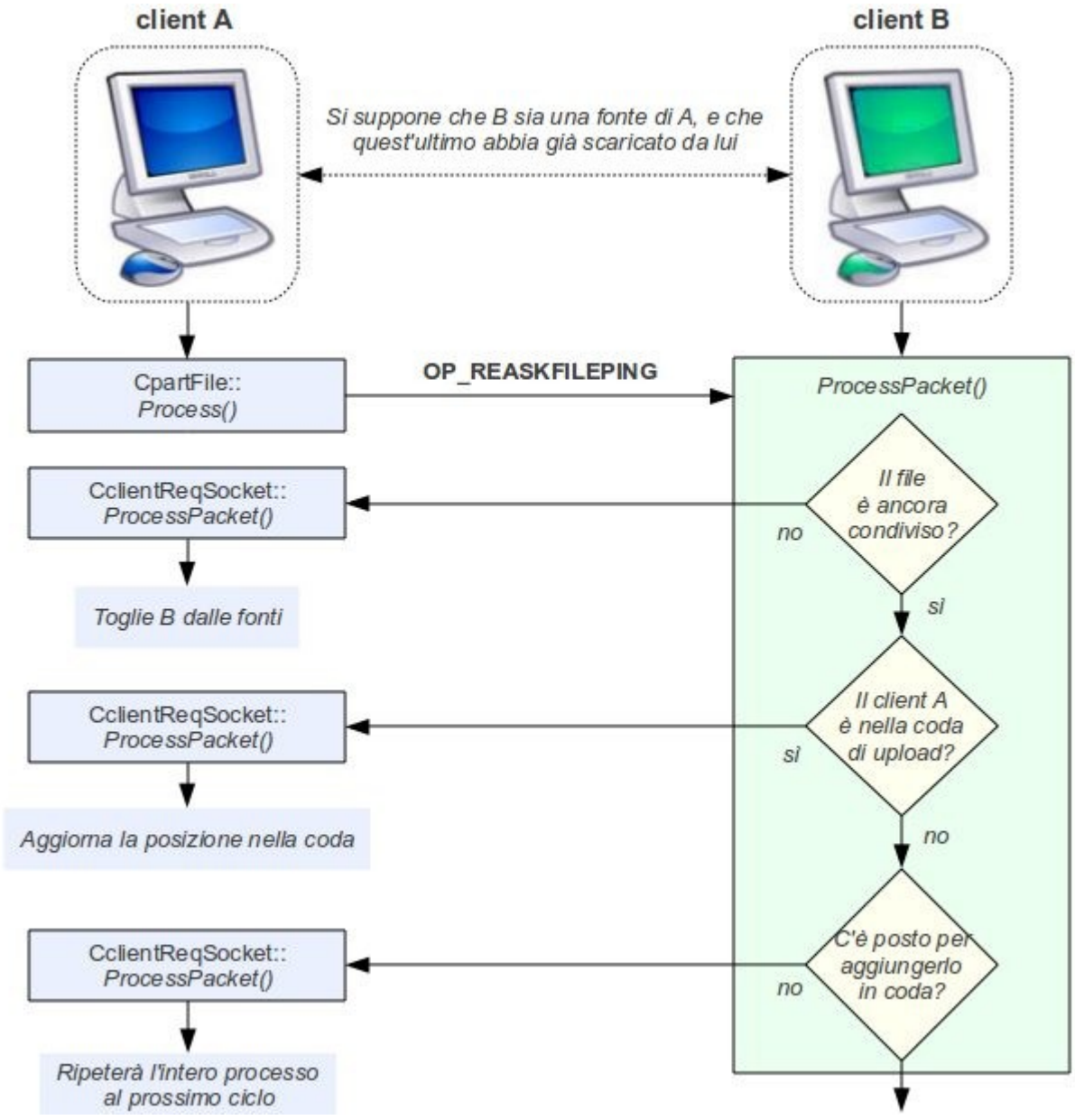

Figura 15: aggiornamento dello stato delle fonti

Identificazione Sicura degli Utenti

L'**Identificazione Sicura degli Utenti** (**SUI**, *Secure User Identification*) avviene attraverso l'utilizzo di firme digitali, un'importante applicazione delle tecnologie di crittografia asimmetrica. Il suo utilizzo principale in **eMule** è la certificazione della proprietà dei crediti di un client presso altri client, per evitare che impostori assumano l'identità ed ereditino i crediti accumulati da altri.

Si tratta di una funzionalità opzionale, attivata di default ma disattivabile in qualsiasi momento.

Cenni firme digitali

La **firma digitale** (*digital signature*) di un messaggio rende possibile la verifica dell'autenticità del mittente e dell'integrità del messaggio stesso.

Il sistema richiede la creazione di una coppia di chiavi: quella **pubblica** (**PK**, Public Key) utilizzata per la verifica della firma, e quella **privata** (**SK**, Secret Key) nota solo al mittente, che la usa per firmare il messaggio. L'algoritmo che genera le chiavi è implementato in modo da rendere la chiave pubblica di un utente la sola in grado di decifrare correttamente i documenti cifrati con la sua chiave privata.

Fasi del processo di firma e verifica di un messaggio:

1. l'utente A calcola l'hash del messaggio con un algoritmo pubblico, così da ottenere il *message digest* (o impronta digitale). L'algoritmo deve garantire due proprietà: che sia impossibile risalire al documento di partenza, e che sia molto improbabile ottenere lo stesso digest da due documenti diversi;
2. l'utente A utilizza la sua chiave privata per cifrare il *message digest*, ottenendo la firma digitale;
3. l'utente A invia il messaggio e la firma digitale all'utente B;
4. l'utente B decifra la firma digitale ricevuta utilizzando la chiave pubblica di A, riottenendo così il *message digest*;

5. l'utente B applica al messaggio la stessa funzione di hash usata da A, e confronta il *message digest* ottenuto con quello calcolato nella fase precedente;
6. se i due valori coincidono, allora sia l'autenticità che l'integrità del messaggio sono confermati.

Si noti che chiunque può verificare l'autenticità di un messaggio firmato. Lo scopo infatti è garantirne l'integrità, non la confidenzialità.

Fasi SUI in eMule

In questo paragrafo forniremo una descrizione semplificata e ad alto livello di come avviene il processo di identificazione sicura in **eMule.**

Creazione chiave privata

Al primo avvio del client **eMule**, se l'opzione *SUI* è abilitata, verrà creata una chiave privata con un algoritmo **RSA** a 384 bit. Questo valore sarà memorizzato nel file `cryptkey.dat` e non dovrà più essere cambiato, o il client non sarà più in grado di dimostrare la proprietà dei crediti accumulati (vedremo poi perché).

Connessione tra client

Il client A e il client B instaurano con successo una connessione, scambiandosi i rispettivi pacchetti di informazione (OP_EMULEINFO) in cui si dicono che supportano entrambi l'Identificazione Sicura.

Scambio informazioni utili alla SUI

Il client A e il client B si invieranno a vicenda:

- una chiave pubblica (creata insieme alla chiave privata);
- un valore casuale che chiameremo X per il client A, ed Y per il client B.

Ciascun client memorizzerà nel proprio file `clients.met` l'*UserHash* e la chiave pubblica dell'altro, così che non debba essere richiesta al prossimo incontro. I valori casuali X ed Y non saranno invece salvati perché vengono rigenerati ad ogni connessione.

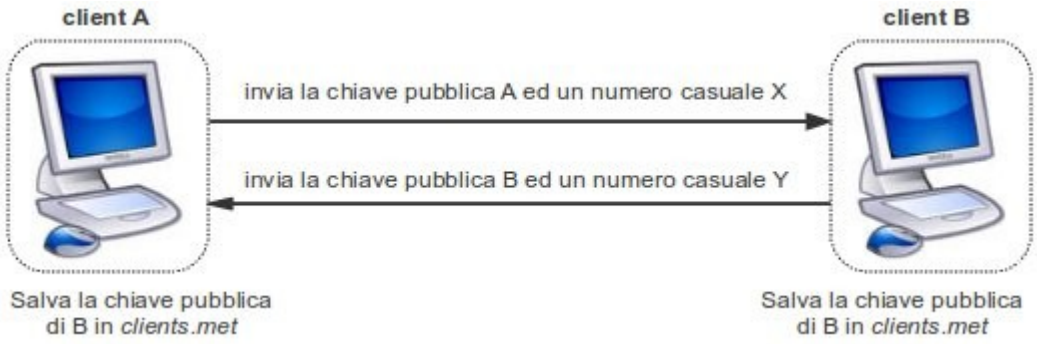

Figura 16: scambio informazioni utili alla SUI

Generazione e invio della firma

Per identificarsi in modo sicuro il client A:

1. genera una firma digitale ottenuta crittando con la sua chiave privata un messaggio composto da: chiave pubblica di B, valore casuale Y (spedito da B);
2. invia la firma digitale al client B.

Si noti che la firma rimane valida finché è valido Y, cioè fino a quando non si instaura una nuova connessione (quindi finché A non cambia IP o finché B non chiude il client).

Ovviamente il client B farà lo stesso per identificarsi presso A.

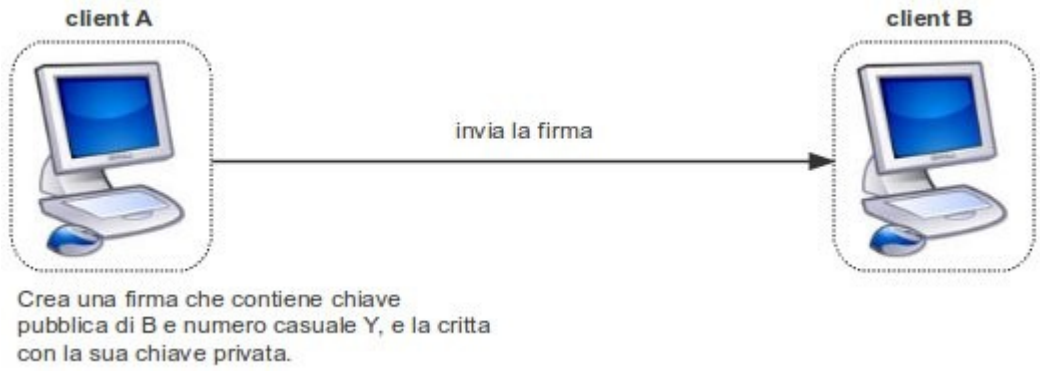

Figura 17: generazione e invio firma in SUI

Verifica della firma

Una volta ricevuta la firma digitale da A, il client B:

1. la decritta con la chiave pubblica di A;
2. verifica che contenga la chiave pubblica di B e il valore Y. Se sì A è autenticato, altrimenti no.

Prima di passare a considerare il codice, una breve ma fondamentale nota. Dalla descrizione di queste cinque fasi appare evidente perché è importante che il file `cryptkey.dat` non venga perso o danneggiato, tuttavia può essere poco chiaro il motivo per cui se ciò accade bisogna cancellare anche il file `preferences.dat`. In realtà la spiegazione è semplice: dato che in `clients.met` i client memorizzano la coppia di valori "chiave pubblica - *UserHash*" dei client con cui entrano in contatto, se uno di loro crea una nuova PK questa non combacerà più con la vecchia accoppiata all'identificativo, e la *SUI* fallirà sempre (impedendo al client di accumulare crediti in futuro).

SUI nel codice

Nel codice di **eMule** il processo di Identificazione Sicura coinvolge principalmente tre sorgenti:

- **`BaseClient.cpp`**, che contiene i metodi necessari per la preparazione dei pacchetti da scambiare tra i client;
- **`ListenSocket.cpp`**, in cui sono implementati i metodi che gestiscono i pacchetti ricevuti, e in generale quelli che prevedono l'ascolto delle attività sulla socket dei client;
- **`ClientCredits.cpp`**, che contiene i metodi per la creazione e la verifica della firma (da ora chiamata anche *signature*);

Vengono inoltre utilizzati molti metodi della libreria libera **Crypto++**, che mette a disposizione un'implementazione in C++ dei principali algoritmi crittografici. In particolare utilizza quelli per la creazione di chiavi crittografiche **RSA** e per la generazione e la verifica di firme digitali.

In *`Figura 18`* viene presentato lo schema guida dell'intero processo *SUI*, riportando come al solito sia i metodi che i pacchetti coinvolti.

Nei prossimi paragrafi ripercorreremo il flusso del diagramma in figura facendo continui riferimenti al codice dei sorgenti interessati.

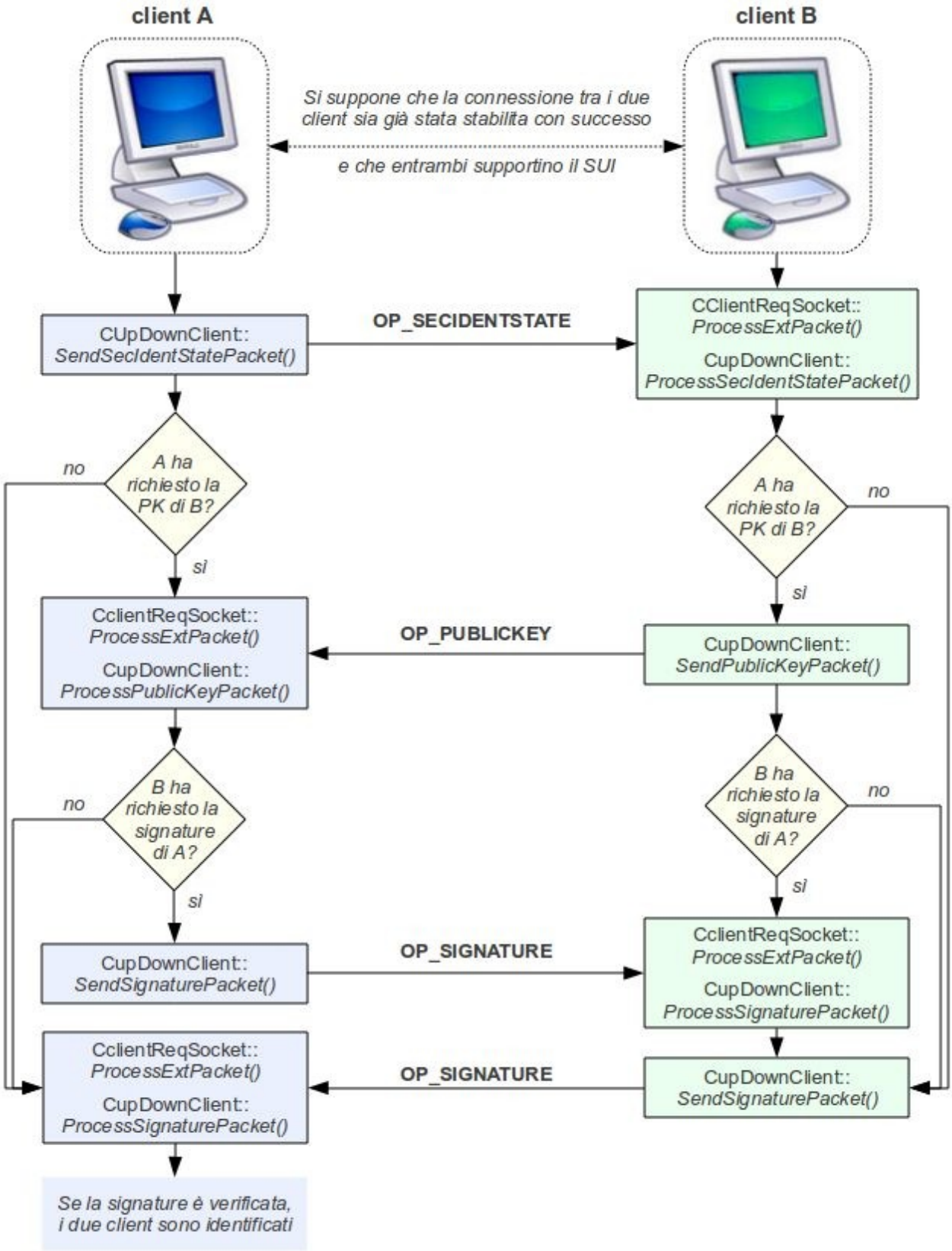

Figura 18: schema riassuntivo *SUI*

Verifica supporto SUI

Nei pacchetti scambiati durante la fase di connessione (OP_HELLOANSWER e OP_EMULEINFOANSWER), i client si comunicano se supportano il meccanismo *SUI*. La verifica viene eseguita dalla funzione `InfoPacketsReceived()`, che in caso positivo mette in moto il processo invocando `SendSecIdentStatePacket()`.

```
// SOURCE: BaseClient.cpp

void CUpDownClient::InfoPacketsReceived()
{
    // indicates that both Information Packets has been received
    // needed for actions, which process data from both packets
    ASSERT ( m_byInfopacketsReceived == IP_BOTH );
    m_byInfopacketsReceived = IP_NONE;

    if (m_bySupportSecIdent){
        SendSecIdentStatePacket();
    }
}
```

Invio del pacchetto OP_SECIDENTSTATE

`SendSecIdentStatePacket()` controlla prima di tutto che il client locale sia in possesso della chiave pubblica di quello remoto, e scrive il risultato della verifica nella variabile `nValue`. I casi sono due:

- se non conosce la sua PK, ne chiederà l'invio insieme alla signature e a un valore casuale da utilizzare come *challenge* (`nValue = IS_KEYANDSIGNEEDED`);
- se invece la conosce già richiederà solo l'invio della signature e del *challenge* (`nValue = IS_SIGNATURENEEDED`).

I valori passati alla variabile `nValue` sono definiti nella libreria **`ClientStateDefs.h`**.

```
//ClientStateDefs.h

enum ESecureIdentState{
    IS_UNAVAILABLE      = 0,
    IS_ALLREQUESTSSEND  = 0,
    IS_SIGNATURENEEDED  = 1,
    IS_KEYANDSIGNEEDED  = 2,
};
```

La richiesta viene incapsulata in OP_SECIDENTSTATE, che comprende sia lo stato *SUI* (il contenuto della variabile `nValue`) che un *challenge* di 4 byte. Il pacchetto viene infine inviato al client remoto.

Di seguito riportiamo il codice:

```
//BaseClient.cpp

void CUpDownClient::SendSecIdentStatePacket()
{
    // check if we need public key and signature
    uint8 nValue = 0;
    if (credits){
        if (theApp.clientcredits->CryptoAvailable()){
            if (credits->GetSecIDKeyLen() == 0) // se non ha la chiave pubblica..
                nValue = IS_KEYANDSIGNEEDED; // ..la richiede insieme alla signature
            else if (m_dwLastSignatureIP != GetIP())
                nValue = IS_SIGNATURENEEDED;
        }
        if (nValue == 0){
            //if (thePrefs.GetLogSecureIdent())
            //  AddDebugLogLine(false, _T("Not sending SecIdentState Packet,
                        //      because State is Zero"));
            return;
        }
        // crypt: send random data to sign
        uint32 dwRandom = rand()+1;
        credits->m_dwCryptRndChallengeFor = dwRandom;
        // creazione del pacchetto OP_SECIDENTSTATE
        Packet* packet = new Packet(OP_SECIDENTSTATE,5,OP_EMULEPROT);
        theStats.AddUpDataOverheadOther(packet->size);
        packet->pBuffer[0] = nValue;
        PokeUInt32(packet->pBuffer+1, dwRandom);
        if (thePrefs.GetDebugClientTCPLevel() > 0)
            DebugSend("OP__SecIdentState", this);
        SendPacket(packet, true);
    }
    else
        ASSERT ( false );
}
```

Ricezione del pacchetto OP_SECIDENTSTATE

Il metodo `ProcessExtPacket()` rimane in perenne ascolto sulla socket aperta con il client, e si attiva alla ricezione di un pacchetto gestendolo a seconda del suo *operation code* (passato come parametro). Gli *operation code* e i relativi significati sono riportati nella libreria **`opcodes.h`**; ad esempio quello di OP_SECIDENTSTATE è `0x87`.

Quando il client remoto riceve il pacchetto OP_SECIDENTSTATE lo processerà dunque con un ramo del metodo `ProcessExtPacket()`, e a seconda del valore dello *stato* di Identificazione Sicura (recuperato dalla variabile `m_SecureIdentState`) predisporrà l'invio della chiave pubblica e della *signature*, o solo di quest'ultima.

```
//ListenSocket.cpp

bool CClientReqSocket::ProcessExtPacket(const BYTE* packet,
                                        uint32 size,
                                        UINT opcode, UINT uRawSize)
{

// [...]

switch(opcode)
{

// [...]

   case OP_SECIDENTSTATE:
   {
      if (thePrefs.GetDebugClientTCPLevel() > 0)
         DebugRecv("OP_SecIdentState", client);
      theStats.AddDownDataOverheadOther(uRawSize);

      client->ProcessSecIdentStatePacket(packet, size);
      if (client->GetSecureIdentState() == IS_SIGNATURENEEDED)
         client->SendSignaturePacket();
      else if (client->GetSecureIdentState() == IS_KEYANDSIGNEEDED)
      {
         client->SendPublicKeyPacket();
         client->SendSignaturePacket();
      }
      break;
   }

// [...]
```

Alcuni commenti al codice:

- il metodo `ProcessSecIdentStatePacket()` è implementato nel file **`BaseClient.cpp`** e prende come parametro il puntatore al pacchetto e la sua dimensione. Ciò che fa è salvare il contenuto di OP_SECIDENTSTATE (stato SUI e *challenge*) rispettivamente nelle variabili `m_SecureIdentState` e `m_dwCryptRndChallengeFrom`;

- i metodi `SendPublicKeyPacket()` e `SendSignaturePacket()` si occupano rispettivamente dell'invio del pacchetto con la chiave pubblica e di quello con la *signature*.

Invio del pacchetto OP_PUBLICKEY

Se il client locale aveva chiesto la chiave pubblica a quello remoto, questi invocherà il metodo `SendPublicKeyPacket()` per inviargli il pacchetto OP_PUBLICKEY che contiene la chiave stessa e la sua dimensione.

```
//BaseClient.cpp

void CUpDownClient::SendPublicKeyPacket()
{
   // send our public key to the client who requested it
   if (socket == NULL || credits == NULL ||
               m_SecureIdentState != IS_KEYANDSIGNEEDED)
   {
      ASSERT ( false );
      return;
   }

   if (!theApp.clientcredits->CryptoAvailable())
      return;

   Packet* packet = new Packet(OP_PUBLICKEY, theApp.clientcredits->GetPubKeyLen()
                                                                          +
1,OP_EMULEPROT);
   theStats.AddUpDataOverheadOther(packet->size);
   memcpy(packet->pBuffer+1,theApp.clientcredits->GetPublicKey(),
          theApp.clientcredits->GetPubKeyLen());

   packet->pBuffer[0] = theApp.clientcredits->GetPubKeyLen();
   if (thePrefs.GetDebugClientTCPLevel() > 0)
      DebugSend("OP__PublicKey", this);

   SendPacket(packet, true);

   m_SecureIdentState = IS_SIGNATURENEEDED;
}
```

Si noti che nella penultima riga viene aggiornato il valore della variabile che rappresenta lo *stato* dell'*Identificazione Sicura* (da `IS_KEYANDSIGNEEDED` a `IS_SIGNATURENEEDED`), il che significa che da questo momento è richiesta solo la *signature*.

Ricezione del pacchetto OP_PUBLICKEY

La ricezione del pacchetto OP_PUBLICKEY è ovviamente presa in carico da un blocco dello **`switch`** del metodo `ProcessExtPacket()`, che ne rimanda la gestione alla funzione `ProcessPublicKeyPacket()`.

```
//ListenSocket.cpp

bool CClientReqSocket::ProcessExtPacket(const BYTE* packet,
                                        uint32 size,
                                        UINT opcode, UINT uRawSize)
{

// [...]

switch(opcode)
{

// [...]

   case OP_PUBLICKEY:
   {
      if (thePrefs.GetDebugClientTCPLevel() > 0)
         DebugRecv("OP_PublicKey", client);

      theStats.AddDownDataOverheadOther(uRawSize);

      client->ProcessPublicKeyPacket(packet, size);
      break;
   }

// [...]
```

La prima operazione compiuta dal metodo `ProcessPublicKeyPacket()` è aggiornare le informazioni del client remoto scrivendo la chiave pubblica che gli ha appena comunicato. Per essere più precisi, il client locale si limita ad aggiungerlo alla lista dei client noti (con la funzione `AddTrackClient()` implementata in **`ClientCredit.cpp`**), così che possa poi essere salvato nel file `clients.dat` dal metodo `SaveList()`.

La seconda operazione compiuta da `ProcessPublicKeyPacket()` è inviare al client remoto - se richiesto - il pacchetto con la *signature*.

```
//BaseClient.cpp

void CUpDownClient::ProcessPublicKeyPacket(const uchar* pachPacket, uint32 nSize)
{
   theApp.clientlist->AddTrackClient(this);

   // [...] verifica di alcune condizioni [...]

   // the function will handle everything (mulitple key etc)
   if (credits->SetSecureIdent(pachPacket+1, pachPacket[0])){
      // if this client wants a signature, now we can send him one
      if (m_SecureIdentState == IS_SIGNATURENEEDED){
         SendSignaturePacket();
      }
      else if(m_SecureIdentState == IS_KEYANDSIGNEEDED)
      {
         // something is wrong
         // [...] scrivi l'errore nel log di sistema [...]
      }
   }
   else
   {
      if (thePrefs.GetLogSecureIdent())
         AddDebugLogLine(false, _T("Failed to use new received public key"));
   }
}
```

Invio del pacchetto OP_SIGNATURE

L'invio del pacchetto OP_SIGNATURE può avvenire in due direzioni:

- dal client locale al client remoto, solo se il client locale ha richiesto la chiave pubblica dell'altro;
- dal client remoto al client locale, sempre.

In entrambi i casi viene invocato il metodo `SendSignaturePacket()` definito in **`BaseClient.cpp`**, che prevede i seguenti passaggi:

1. la verifica di una serie di condizioni iniziali (corretto stato del *SUI*, chiave pubblica e *challenge* dell'altro client disponibile);
2. la verifica del *Client ID*, che condizionerà il tipo di valore da assegnare alle due variabili che consentono anche ai client con *ID Basso* di identificarsi in modo sicuro (`ChallengeIP` e `byChaIPKind`);
3. il riempimento di un buffer con il contenuto di alcune variabili, che sarà poi

cifrato nella fase successiva;

4. creazione della *signature* (vedi paragrafo successivo);
5. preparazione e invio del pacchetto, che contiene la *signature*, la sua lunghezza e il valore della variabile `byChaIPKind`.

```
//BaseClient.cpp

void CUpDownClient::SendSignaturePacket()
{
   // [...] verifica condizioni iniziali [...]

   uint8 byChaIPKind = 0; // indica il tipo di Challenge IP usato
   uint32 ChallengeIP = 0;
   // V2 fa riferimento alla seconda versione del metodo, che se usata
   // permette il supporto del SUI anche per gli ID Bassi
   if (bUseV2)
   {
      if (theApp.serverconnect->GetClientID() == 0 ||
                  theApp.serverconnect->IsLowID())
       {
         // se il client locale ha ID Basso non riuscirà a ottenere il suo IP
         // pubblico, quindi userà quello del client remoto
         ChallengeIP = GetIP();
         byChaIPKind = CRYPT_CIP_REMOTECLIENT;
      }
      else
      {
         // in caso contrario userà il suo Client ID
         ChallengeIP = theApp.serverconnect->GetClientID();
         byChaIPKind  = CRYPT_CIP_LOCALCLIENT;
      }
   }

   uchar achBuffer[250];
   uint8 siglen = theApp.clientcredits->CreateSignature(credits,
                        achBuffer, 250, ChallengeIP, byChaIPKind );
   if (siglen == 0){
      ASSERT ( false );
      return;
   }
   Packet* packet = new Packet(OP_SIGNATURE,siglen + 1 +
                                          ( (bUseV2)? 1:0 ),OP_EMULEPROT);
   theStats.AddUpDataOverheadOther(packet->size);
   memcpy(packet->pBuffer+1,achBuffer, siglen);
   packet->pBuffer[0] = siglen;
   if (bUseV2)
      packet->pBuffer[1+siglen] = byChaIPKind;
   if (thePrefs.GetDebugClientTCPLevel() > 0)
      DebugSend("OP__Signature", this);
   SendPacket(packet, true);
   m_SecureIdentState = IS_ALLREQUESTSSEND;
}
```

Creazione della signature

In questo paragrafo torneremo sui dettagli della creazione della *signature*, che abbiamo già dato per inviata nella fase precedente.

Il metodo utilizzato è il `CreateSignature()` (in **`ClientCredits.cpp`**), che fa uso tra le altre della libreria *Crypto++*.

Riportiamo la firma del metodo e analizziamo alcuni dei suoi parametri:

```
uint8 CClientCreditsList::CreateSignature(CClientCredits* pTarget,
                                          uchar* pachOutput, uint8 nMaxSize,
                                         uint32 ChallengeIP, uint8 byChaIPKind,
                                        CryptoPP::RSASSA_PKCS1v15_SHA_Signer* sigkey)
```

Il primo parametro è un puntatore all'istanza della classe `CClientCredit`, quella del client destinatario della *signature*. La classe è così strutturata:

```
//ClientCredits.h

struct CreditStruct{
    uchar       abyKey[16]; // UserHash
    uint32      nUploadedLo;
    uint32      nDownloadedLo; // tot dati scaricati dal client remoto
    uint32      nLastSeen;
    uint32      nUploadedHi;     // insieme a nUploadedLo è il numero a 64 bit che
                                 // indica il tot dei dati inviati al client remoto
    uint32      nDownloadedHi; // insieme a nDownloadedLo è il numero a 64 bit che
                               // indica il tot dei dati scaricati dal client remoto
    uint16      nReserved3;
    uint8       nKeySize;
    uchar       abySecureIdent[MAXPUBKEYSIZE];  // chiave pubblica
};
```

Il secondo parametro di `CreateSignature()` è il puntatore al buffer in cui sarà inserita la firma una volta creata, e da cui sarà presa da `SendSignaturePacket()` per riempire il pacchetto OP_SIGNATURE.

Gli altri parametri sono già stati discussi o facilmente comprensibili, tranne l'ultimo, che è il puntatore a una classe definita nelle librerie di *Crypto++*: `CryptoPP::RSASSA_PKCS1v15_SHA_Signer`. Si tratta dello schema di firme digitali PKCS v2.0, che fa riferimento all'algoritmo crittografico asimmetrico RSA

utilizzato per generare la *signature*. Questo metodo utilizza la chiave privata del client recuperata dal file `cryptkey.dat` dalla funzione `InitializeCrypting()`.

Di seguito riportiamo il codice di creazione della *signature*.

```
//ClientCredits.cpp

uint8 CClientCreditsList::CreateSignature(CClientCredits* pTarget,
                                          uchar* pachOutput, uint8 nMaxSize,
                                         uint32 ChallengeIP, uint8 byChaIPKind,
                                        CryptoPP::RSASSA_PKCS1v15_SHA_Signer* sigkey)
{
    // sigkey param is used for debug only
    if (sigkey == NULL)
        // se il sigkey non è passato come parametro, usa la SK del client
        sigkey = m_pSignkey;

    // [...]

    try{

        SecByteBlock sbbSignature(sigkey->SignatureLength());
        AutoSeededRandomPool rng;
        byte abyBuffer[MAXPUBKEYSIZE+9]; // MAXPUBKEYSIZE è una costante pari a 80
        // assegna a keylen la lunghezza della chiave pubblica del client remoto
        uint32 keylen = pTarget->GetSecIDKeyLen();
        // scrive in abyBuffer la chiave pubblica del client remoto
        memcpy(abyBuffer,pTarget->GetSecureIdent(),keylen);
        // 4 additional bytes random data send from this client
        // recupera il valore del challenge inviato da OP_SECIDENTSTATE
        uint32 challenge = pTarget->m_dwCryptRndChallengeFrom;
        ASSERT ( challenge != 0 );

        // aggiunge in abyBuffer il valore del challenge
        PokeUInt32(abyBuffer+keylen, challenge);
        // [...] inserisce in abyBuffer il valore del ChallengeIP e del suo tipo
         // [...] chiama il metodo di Crypto++ per crittare abyBuffer con RSA,
        // usando la chiave privata del client locale
        sigkey->SignMessage(rng,abyBuffer,keylen+4+ChIpLen,sbbSignature.begin());

        // [...] memorizza la signature nel buffer pachOutput [...]
    }
    // restituisce la lunghezza della signature
    return nResult;
}
```

Ricezione del pacchetto OP_SIGNATURE

Come nei casi precedenti, la ricezione del pacchetto OP_SIGNATURE è gestita da un blocco del metodo `ProcessExtPacket()`, che a sua volta invoca il metodo

`ProcessPublicKeyPacket()` implementato in **`BaseClient.cpp`**.

```
//ListenSocket.cpp

bool CClientReqSocket::ProcessExtPacket(const BYTE* packet,
                                        uint32 size,
                                        UINT opcode, UINT uRawSize)
{

// [...]

switch(opcode)
{

// [...]

   case OP_SIGNATURE:
    {
    if (thePrefs.GetDebugClientTCPLevel() > 0)
        DebugRecv("OP_Signature", client);
    theStats.AddDownDataOverheadOther(uRawSize);

    client->ProcessSignaturePacket(packet, size);
    break;
    }

// [...]
```

Verifica della signature

La verifica della *signature* rappresenta l'ultima fase del sistema di Identificazione Sicura degli Utenti, ed è implementata in `ProcessSignaturePacket()`. Una volta passati come parametri il puntatore al pacchetto e la sua dimensione, il metodo verifica una serie di condizioni iniziali:

- che il pacchetto sia di tipo e dimensioni corrette;
- che il client remoto referenziato esista;
- che entrambi i client supportino l'Identificazione Sicura;
- che il client remoto non abbia già inviato un'altra signature;
- che la chiave pubblica del client remoto sia nota.

Se tutte le verifiche hanno successo viene lanciata la funzione `VerifyIdent()` della classe `RSASSA_PKCS1v15_SHA_Verifier` definita nelle librerie *Crypto++*, che permette di creare oggetti di tipo verificatore per una *signature* generata con

algoritmo RSA. Se la verifica va a buon fine allora il client remoto è autenticato in modo sicuro ed i suoi crediti accumulati presso il client locale vengono automaticamente riconosciuti. In caso contrario il processo *SUI* fallisce e l'evento viene notificato nel log dell'applicazione.

```
//BaseClient.cpp

void CUpDownClient::ProcessSignaturePacket(const uchar* pachPacket, uint32 nSize)
{
    // here we spread the good guys from the bad ones ;)
    // [...] verifica condizioni iniziali

    if (theApp.clientcredits->VerifyIdent(credits, pachPacket+1, pachPacket[0],
                                          GetIP(), byChaIPKind ) )
     {
        // client verificato
        if (GetFriend() != NULL && GetFriend()->IsTryingToConnect())
            GetFriend()->UpdateFriendConnectionState(FCR_USERHASHVERIFIED);
    }
    else
    {
        // client non verificato
        if (GetFriend() != NULL && GetFriend()->IsTryingToConnect())
            GetFriend()->UpdateFriendConnectionState(FCR_SECUREIDENTFAILED);
        if (thePrefs.GetLogSecureIdent())
            AddDebugLogLine(false,
                    _T("'%s' has failed the secure identification, V2 State: %i"),
                    GetUserName(), byChaIPKind);
    }
    // [...]
}
```

Concludiamo il paragrafo e il capitolo studiando più da vicino il comportamento del metodo `VerifyIdent()` richiamato da `ProcessSignaturePacket()`.

La funzione principale del metodo è la `VerifyMessage()`, che confronta:

- un buffer che sia stato riempito esattamente come nel metodo `CreateSignature()` partendo dagli stessi dati (alcuni dei quali, come il *challenge*, sono noti solo a lui e al client a cui li aveva inviati);
- il buffer ottenuto decrittando la *signature* utilizzando la chiave pubblica del client che gliel'aveva inviata. Il fatto che il nome dato al buffer (`abyBuffer`) sia uguale in entrambi i metodi è un'ulteriore sottolineatura del concetto.

Se i due buffer sono uguali allora l'utente è stato identificato in modo sicuro.

```
//ClientCredits.cpp

bool CClientCreditsList::VerifyIdent(CClientCredits* pTarget,
                                     const uchar* pachSignature, uint8 nInputSize,
                                     uint32 dwForIP, uint8 byChaIPKind)
{
   // [...] verifica condizioni iniziali [...]
   bool bResult;
   try{
      // definisce la sorgente che contiene la PK
      StringSource ss_Pubkey((byte*)pTarget->GetSecureIdent(),
                             pTarget->GetSecIDKeyLen(),true,0);
      // crea una variabile di tipo verificatore a cui passa la sorgente ss_Pubkey
      RSASSA_PKCS1v15_SHA_Verifier pubkey(ss_Pubkey);
      // 4 additional bytes random data send from this client +5 bytes v2
      byte abyBuffer[MAXPUBKEYSIZE+9];
      // inserisce il valore della chiave pubblica nell'abyBuffer
      memcpy(abyBuffer,m_abyMyPublicKey,m_nMyPublicKeyLen);
      uint32 challenge = pTarget->m_dwCryptRndChallengeFor;
      ASSERT ( challenge != 0 );
      // inserisce il valore del challenge nell'abyBuffer
      PokeUInt32(abyBuffer+m_nMyPublicKeyLen, challenge);

      // [...] recupera le informazioni del challenge IP e del suo tipo,
      // e le inserisce in abyBuffer [...]

      // verifica che il valore della signature decrittata con la PK sia
      // uguale ad abyBuffer
      bResult = pubkey.VerifyMessage(abyBuffer, m_nMyPublicKeyLen+4+nChIpSize,
                                     pachSignature, nInputSize);
   }
   // [...] gestione delle eccezioni nella try [...]

   if (!bResult){
      // se la signature non è verificata, il SUI fallisce..
      if (pTarget->IdentState == IS_IDNEEDED)
      pTarget->IdentState = IS_IDFAILED;
   }
   else{
      // ..altrimenti va a buon fine
      pTarget->Verified(dwForIP);
   }
   return bResult;
}
```

Appendice A: I File

I file della rete **eDonkey** sono identificati in modo univoco da una chiave chiamata *FileHash*, che dipende esclusivamente dal contenuto del file e non dal nome con cui è salvato in locale. Ciò consente ai client di trovare tutte le fonti di un certo file indipendentemente dal nome con cui l'hanno memorizzato sul loro hard disk.

FileHash e segmentazione dei file sono alla base sia del sistema di download multi-fonte che della rilevazione e correzione degli errori.

La rete supporta la condivisione di file di qualsiasi tipo, con una dimensione massima di 256 GB.

Suddivisione file

I file vengono divisi in **parti** da 9,28 MB, chiamati **chunk** (o *part*), a loro volta suddivisi in **blocchi** da 180 KB. Ad esempio un file di 100 MB sarà formato da:

- 11 chunk, di cui 10 di dimensioni standard e l'ultimo di 7,2 MB;
- 556 blocchi, di cui 555 di dimensioni standard e l'ultimo di 100 KB.

Per ogni chunk viene calcolato il suo valore di hash (*Part Hash*) utilizzando l'algoritmo MD4, e l'insieme di questi hash (*Hashset*) concorrerà a formare il *FileHash* del file da usare nella rete **eDonkey**.

Come è rappresentato un download

Per ogni download **eMule** crea 3 file temporanei nella cartella `emule/temp`, a cui viene assegnato un nome numerico che dipende dall'ordine con cui è stato aggiunto nella lista dei file da scaricare: il primo si chiamerà `001`, il secondo `002` e così via (quando uno di questi viene completato, al download successivo viene assegnato il numero vacante).

Assumendo che il nome del file sia *xxx*, avremo:

- *xxx*`.part`, in cui vengono memorizzati i chunk finora scaricati. Poiché questi

non vengono scaricati in modo lineare dall'inizio alla fine ma possono essere una parte qualsiasi del file completo, il file temporaneo sarà salvato con le stesse dimensioni di quest'ultimo anche se è stato scaricato un solo blocco;

- *xxx*`.part.met`, il file di controllo per il download, che contiene: l'hash, il nome, la dimensione, i chunk scaricati. Queste quattro informazioni sono le prime ad essere scaricate da **eMule**;
- *xxx*`.part.met.bak`, una copia di backup del file precedente, pronta all'utilizzo in caso di corruzione.

Gestione delle corruzioni

ICH

L'**Intelligent Corruption Handling** (**ICH**) è un sistema per la rilevazione di porzioni corrotte di un file. Ogni volta che viene scaricato un chunk di un file, **eMule** ne calcola l'hash: se questo corrisponde al *PartHash* riportato nel file `xxx.part.met` allora il download prosegue con il chunk successivo, altrimenti lo scaricherà di nuovo. Si noti che in caso si successo della verifica il chunk viene subito messo in condivisione con il resto della rete.

Per evitare di riscaricare ogni volta l'intero chunk in caso di errore, l'ICH scarica solo il primo blocco e ripete la verifica dell'hash: se è corretto passa al chunk successivo, altrimenti riscarica il secondo blocco e ripete il controllo, e così via finché non si ottiene il giusto valore del *PartHash*. Il limite dell'algoritmo ICH è che nel caso peggiore bisognerà ripetere il download di tutta la parte corrotta.

AICH

L'**Advanced Intelligent Corruption Handling** (**AICH**) rappresenta la naturale evoluzione dell'ICH, perché effettua i controlli di integrità sui singoli blocchi e non sull'intero chunk. In questo modo si aumenta notevolmente l'efficienza della gestione delle corruzioni, perché si riducono al minimo i “re-scaricamenti” e l'*overhead* da essi generato.

L'AICH richiede come prerequisito il calcolo di un valore di hash per ogni blocco (*BlockHash*) usando l'algoritmo di hashing SHA1.

Per comprendere meglio il funzionamento dell'algoritmo introduciamo il concetto di albero di hash (*HashTree*), una rappresentazione grafica della composizione di un file. La *Figura 19* mostra l'*HashTree* di un file composto da 4 chunk, ognuno dei quali contiene 53 blocchi per un totale di 212 *BlockHash*. Ognuno di essi dà a sua volta vita ad un *HashTree* per altri 7 livelli, finché non si raggiunge il *RootHash*, che è l'identificativo dell'intero file.

Un altro nome con cui è noto quest'albero è *AICH HashSet*. I colori dei cerchi indicano le dipendenze matematiche che vanno dal *BlockHash* più piccolo (in basso) fino al *RootHash* (in alto), il che significa che se è corretto lui è verificato l'intero albero.

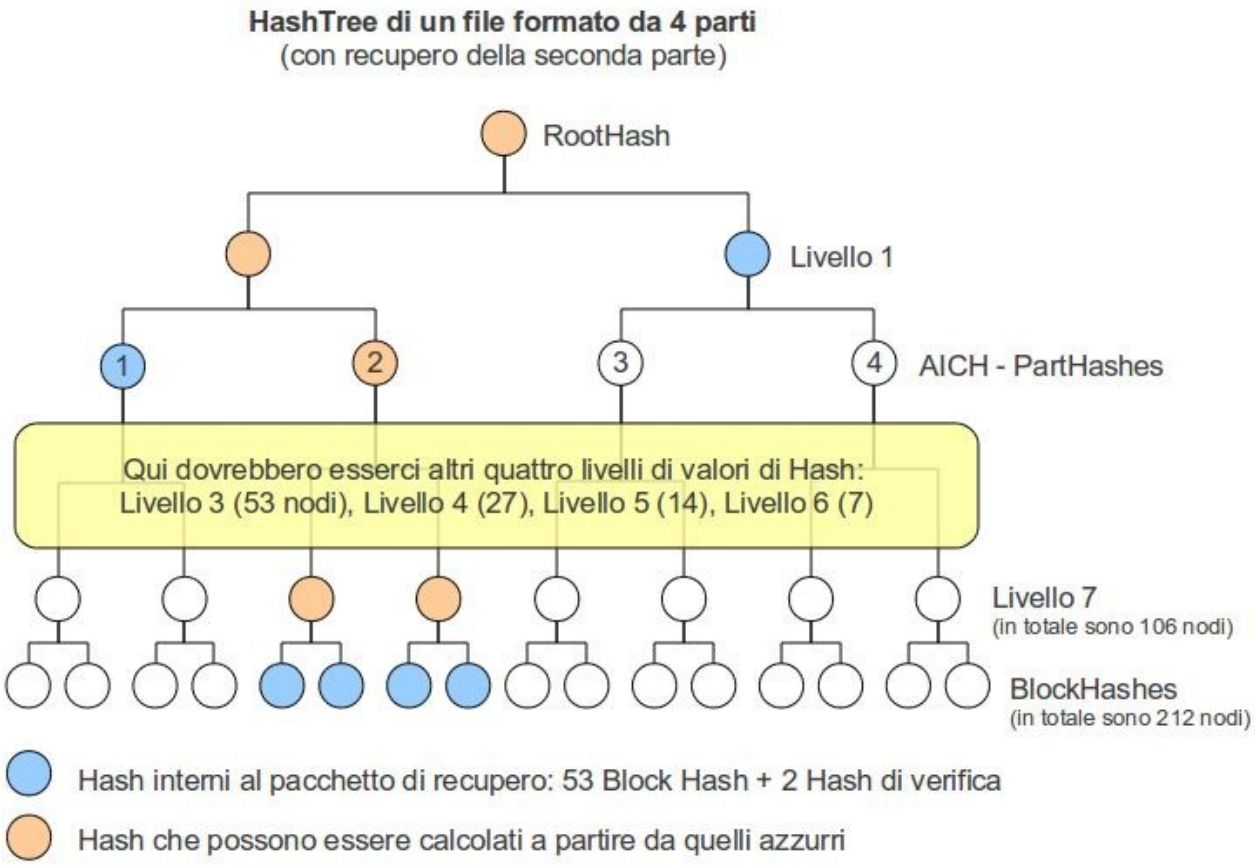

Figura 19: funzionamento AICH

eMule crea un *AICH HashSet* per ogni file condiviso, e lo memorizza nel file `known_64.met` che si trova di default nella cartella `emule\config`.

Vediamo ora nel dettaglio come funziona il recupero dei dati corrotti.

Ogni volta che **eMule** rileva che la parte di un file è corrotta richiede un *pacchetto di*

recupero da un client casuale che dispone dell'*AICH Hashset* completo del file. Questo pacchetto contiene tutti i 53 *BlockHash* della parte corrotta, ed un numero di hash di verifica dell'intero albero così calcolato:

$2^x \geq$ [n°parti] , dove x è il numero degli hash di verifica

Dopo aver ricevuto il pacchetto di recupero **eMule** analizza gli hash di verifica per il *RootHash* verificato. Se c'è corrispondenza passerà a confrontare tutti e 53 i blocchi della parte corrotta con quelli riportati nel pacchetto. L'AICH conserverà tutti i blocchi per cui c'è corrispondenza mentre riscaricherà solo quelli corrotti.

Da quanto detto appare evidente che il *RootHash* contenuto nel pacchetto di recupero deve essere assolutamente fidato. La cosa migliore sarebbe scaricare il file da un link **eD2k** con il *RootHash* integrato, perché assumendo che la fonte di quel link sia di fiducia lo sarà anche il suo hash. Se invece non viene fornito insieme al link **eMule** si preoccuperà di verificare l'esattezza di quello che riceve dalle fonti del file. In particolare considererà valido un *RootHash* solo se lo riceverà identico da almeno 10 fonti diverse, e se almeno il 92% di tutte le fonti concorderanno su quel valore. Si noti inoltre che se il *RootHash* è fornito col link allora il suo valore sarà salvato sul disco, altrimenti sarà valido solo per la durata di una sessione.

Nel caso in cui il sistema AICH non possa essere utilizzato, ad esempio perché non esiste un *AICH HashSet* valido per un certo file, si può sempre ricorrere al metodo ICH per recuperare la parte corrotta.

Collezioni

Nella versione 0.46b di **eMule** è stato introdotto il supporto alle collezioni, ovvero file indice che contengono i collegamenti a tutti i file che ne fanno parte, solitamente correlati tra loro da un qualche criterio di omogeneità.

Appendice B: Gestione delle fonti

La ricerca delle fonti è un'attività di enorme importanza per un'applicazione P2P di *file sharing*, motivo per cui **eMule** adotta più di un sistema per metterla in atto:

- attraverso la rete **eD2k**, interrogando a intervalli regolari sia il server a cui si è connesso sia gli altri presenti nella lista in `Server.met`;
- attraverso la rete **Kad**;
- attraverso il sistema dello *Scambio fonti*: quando **eMule** trova una fonte per un download le chiede di inviargli la lista degli utenti a cui lei è a sua volta collegata per scaricare lo stesso file. Si noti che ciò avviene sia se la fonte contattata è completa sia se ha solo alcune parti da condividere;
- attraverso il sistema *Passivo*: il client chiede a tutti i client per cui è fonte se possono essere anch'essi fonte per lui.

Numero delle fonti

In linea teorica più fonti vengono trovate per un certo download e più velocemente viene scaricato quel file. Nella pratica c'è però un limite: maggiore è il numero delle fonti e maggiore è il numero di connessioni che **eMule** si troverà a gestire, e se sono troppe congestionerà il traffico e ostacolerà i download. Nella maggior parte delle connessioni è consigliabile non superare il valore (sperimentale) di 3000 fonti trovate.

Formato delle fonti

Nel client **eMule** il formato delle fonti è **`XX/YY + ZZ (WW)`** , dove:

- **`XX`**, è il numero di fonti disponibili (quelle trovate e da cui è possibile scaricare);
- **`YY`**, è il numero totale di fonti trovate;
- **`ZZ`**, è il numero di fonti marcate come **A4AF** (vedi dopo);
- **`WW`**, è il numero di fonti da cui si sta attualmente scaricando una parte del file.

Per capire il significato delle fonti "**zz**" va prima segnalata una regola importante per la rete **eDonkey**, ovvero che ogni client può essere nella coda di un utente per un solo file alla volta. Quindi ad esempio se un client B vuole scaricare due file posseduti dal client A, entrerà nella coda di uno e sarà messo nello stato **Asked for another file** (**A4AF**) per l'altro.

Di default la gestione delle fonti A4AF è automatica, ma può essere regolata manualmente dall'utente a seconda della priorità con cui si vogliono scaricare i file. Ad esempio è possibile assegnare a un unico download X tutte le fonti marcate come *Asked for another file*, il che significa che **eMule** scorrerà la lista dei K client da cui può scaricare X, e se uno di questi si trova in coda per un altro file lo toglierà da quella e la sposterà su quella di X. L'operazione ha come effetto l'aumento del numero di fonti del file e di conseguenza anche delle possibilità di scaricarlo più velocemente.

Se al contrario l'obiettivo è dare precedenza ad altri file (ad esempio perché più rari), basterà impedire che nelle code degli altri download ci siano fonti A4AF.

La gestione avanzata delle fonti *Asked for another file* rappresenta dunque un'ulteriore sistema per imporre precedenze nei download.

Appendice C: Code e crediti

In **eMule** il download di un file è gestito attraverso il meccanismo delle code per ragioni di efficienza, perché ogni client può sostenere un numero limitato di connessioni incidenti (in entrata e uscita) ed accettarne di nuove in condizioni di sovraccarico non favorisce il sistema.

Ogni client crea automaticamente una coda in cui inserisce tutte le richieste di upload degli altri *peer*. La coda di upload è unica, quindi non dipende dal file per cui si è richiesti come fonte.

Il client che vuole scaricare un file viene dunque inserito nella coda di attesa di qualche altro client, e la posizione che gli viene attribuita in questa lista è chiamata **Queue Rank** (**QR**). Quando raggiunge la posizione zero in una coda (**QR: 0**) accede a uno *slot* di upload della fonte, e inizia a scaricare. Per slot si intende una delle porzioni in cui viene divisa la banda in upload per la trasmissione di un file, quindi il numero di *Kbps* (configurabili) assegnati ad ogni client per il download.

Funzionamento della coda

Si supponga che un client sia fonte per un file richiesto da diversi client in momenti diversi. Se l'avanzamento della coda fosse lineare si avrebbe una politica di tipo **FIFO** (*First In First Out*), ma **eMule** premia chi condivide di più assegnandogli un punteggio che gli permette di risalire la coda più velocemente. Il punteggio va contestualizzato alla coda in cui si trova il client, quindi due punteggi uguali in due code differenti potrebbero avere effetti molto diversi. Per questo motivo **eMule** fornisce solo il **QR** e non il punteggio, perché la posizione in coda è un'informazione non ambigua e non va messa in relazione col contesto.

Il **QR** viene aggiornato ogni 30 minuti, e solo quando arriva a 0 viene assegnato uno slot di upload. Per assicurare che possa scaricare per un tempo sufficiente senza essere scalzato in prima posizione da un altro, gli viene raddoppiato il punteggio per

15 minuti. Infine, quando il client si disconette dalla rete mantiene la sua posizione in coda per altri 20 minuti.

Esistono alcune eccezioni alla normale gestione di una coda:

- *coda piena*, quando la fonte raggiunge il limite massimo di lunghezza della coda e rifiuta tutte le richieste di connessione;
- *Asked for another file* (*A4AF*), di cui si è discusso nel capitolo precedente.

Voto e punteggio

Finora abbiamo parlato di punteggio, ma in realtà dovremmo distinguere tra **Voto** (*Rating*) e **Punteggio** (*Score*).

Voto

Ad ogni client in coda viene assegnato inizialmente un **Voto** pari a 100, che viene continuamente aggiornato con l'utilizzo di diversi *modificatori*. Questi non sono altro che valori da moltiplicare al voto iniziale, e possono essere maggiori di 1 (aumentando il voto), inferiori ad 1 (diminuendolo) o pari ad 1 (lasciandolo invariato).

La formula per calcolare il voto è la seguente:

```
voto = 100 * [Bannato] * [Priorità] * [OldVersion] * [Crediti]
```

Spieghiamo nel dettaglio i fattori che concorrono alla formazione del voto.

In **eMule** è possibile bloccare le connessioni con alcuni utenti, ad esempio perché trasmettono (intenzionalmente o meno) dati corrotti o *fake*, o perché fanno spam. Quest'operazione viene chiamata *ban* dell'utente e consiste nell'inserire l'IP del client da bloccare (sia in upload che in download) in una *blacklist* memorizzata nel file `ipfilter.dat`. Il modificatore `[Bannato]` azzera il voto del client, impedendogli così di raggiungere la cima della coda e quindi scaricare.

La `[Priorità]` del file condiviso indica la frequenza con cui si concede la trasmissione di quel file piuttosto che un altro, e può avere cinque valori:

- Release = 1,8
- Alta = 0,9
- Normale = 0,7

- Bassa = 0,6
- Molto bassa = 0,2

Di default **eMule** assegna la priorità automaticamente basandosi sul numero di fonti e utilizzando solo i valori Alta [+] (0-40 fonti), Media [=] (41-80) e Bassa [-] (>80). Gli altri due valori (Release e Molto bassa) possono essere impostati solo manualmente.

Il modificatore `[OldVersion]` penalizza l'utilizzo delle versioni di **eMule** precedenti alla v.20a, dimezzandone il voto. Queste versioni causavano infatti un carico di rete altissimo, motivo per cui viene incentivato l'aggiornamento ai rilasci successivi.

L'ultimo modificatore è quello dei `[Crediti]`, che hanno un ruolo fondamentale nella formazione del voto e dipendono dal rapporto “dati inviati / dati scaricati”. Può valere da 1 a 10 e premia gli utenti che condividono file nella rete. Si noti che il valore minimo è 1, quindi non avere crediti non abbassa il voto (o chi utilizza **eMule** la prima volta sarebbe continuamente scavalcato); il loro effetto è solo migliorativo sul *Rating* complessivo. Saranno approfonditi in dettaglio nell'ultimo paragrafo del capitolo.

Proponiamo due esempi riepilogativi.

Caso A. Il voto di un client **eMule** 0.50a che richiede un file ad alta priorità da una fonte da cui non è stato bannato e presso cui ha ottenuto un modificatore per i crediti pari a 4 è:

```
voto = 100 * [Bannato] * [Priorità] * [OldVersion] * [Crediti]
     = 100 * 1 * 0,9 * 1 * 4 = 360
```

Caso B. Il voto di un client **eMule** 0.19a che richiede un file di priorità release da una fonte da cui non è stato bannato e presso cui ha ottenuto un modificatore per i crediti pari a 5 è:

```
voto = 100 * [Bannato] * [Priorità] * [OldVersion] * [Crediti]
     = 100 * 1 * 1,8 * 0.5 * 5 = 450
```

Punteggio

Il **Punteggio** è calcolato applicando un ultimo modificatore al *Voto* appena ottenuto: il *tempo di attesa in coda*, cioè l'intervallo di tempo espresso in secondi da cui il client sta aspettando il suo turno per scaricare. Ogni secondo corrisponde a 1 punto, quindi ad esempio 10 minuti in coda equivalgono a 600 punti.

La formula per calcolare il punteggio è la seguente:

```
punteggio = Voto * [tempo di attesa in coda] / 100
```

Riprendiamo gli esempi di prima.

Se il client descritto nel **Caso A** è in attesa da 14 minuti, il suo punteggio sarà:

```
punteggio = Voto * [tempo di attesa in coda] / 100
          = 360 * 840 / 100 = 3024
```

Se il client descritto nel **Caso B** è in attesa da 6 minuti, il suo punteggio sarà:

```
punteggio = Voto * [tempo di attesa in coda] / 100
          = 450 * 240 / 100 = 1080
```

Crediti

I **crediti** sono un importante modificatore calcolato da un client per favorire o penalizzare gli altri client che vogliono scaricare da lui. Non si tratta di un modificatore globale, ma può essere usato solo nei confronti del client che li ha concessi. Il principio è che maggiori sono i crediti accumulati da un client presso una fonte, e più velocemente questo scalerà la sua coda.

Esistono due formule per calcolarli, e viene scelta quella che dà il risultato minore:

- **Formula 1**
  `crediti = ([byte ricevuti] * 2) / [byte inviati]`
- **Formula 2**
  `crediti = √([Megabyte ricevuti] + 2)`

Ad esempio se una fonte ha ricevuto 10 MB da un client e gliene ha inviati 2 MB, si ottiene:

Formula 1: `crediti = (10485760 byte * 2) / 2097152 byte =` **10**

Formula 2: `crediti = sqrt(4 MB) =` **2**

Quindi il numero di crediti acquisiti dal client presso quella fonte sarà pari a **2**.

Nel calcolo delle formule ci sono due condizioni particolari da tenere in considerazione:

- se la quantità di dati inviati è inferiore a 1 MB, allora il valore dei crediti sarà 1;
- se la quantità di dati scaricati è pari a 0 MB, allora il valore dei crediti sarà 10;
- il valore finale deve essere compreso tra 1 e 10. Risultati superiori o inferiori a queste soglie vengono semplicemente troncati, senza alcuna operazione di normalizzazione.

Un client con un numero di crediti alto potrà superare più rapidamente anche quei client che sono in coda da più tempo di lui. Se infatti ad esempio fosse in coda da 1 ora (quindi 3600 minuti) per un file con priorità pari a 1 e vantasse 10 crediti presso quella fonte, allora otterrebbe un punteggio di 36000 (un tempo di attesa di 10 ore se non avesse avuto crediti).

I crediti accumulati non dipendono dal numero di file messi in condivisione in rete, ma dalla quantità di dati effettivamente inviata. Ovviamente più file si condividono e più si hanno possibilità che qualcuno li chieda, ma è solo a quel punto che si comincerebbero ad accumulare crediti.

Per evitare manomissioni del loro valore a vantaggio dell'utente (frodando il sistema virtuoso di premiazione di **eMule**) è stato scelto di salvare in locale solo i crediti dovuti agli altri client e non quelli vantati. Il loro valore è memorizzato nel file `clients.met` sotto forma di record, per ognuno dei quali è riportata la coppia "*UserHash* dell'utente – numero di crediti che ha accumulato presso il client locale". Hanno validità per 150 giorni (circa 5 mesi), dopodiché vengono azzerati.

Slot amico

Lo **slot amico** è uno slot di upload assegnato ad un client che si trova nella lista dei suoi *amici* (client inseriti nell'omonima lista preferenziale). Ogni volta che un client ha

questo slot abilitato e proverà a scaricare dall'amico, gli sarà data massima priorità nella coda.

Lo slot amico può essere concesso a un solo amico per volta, quindi eventuali slot amici già concessi saranno disabilitati automaticamente. Il suo effetto si limita alla velocità di risalita della coda e non alla velocità di scaricamento, che non supererà limiti validi per tutti gli altri upload della fonte.

Appendice D: Creazione coppia di chiavi

In **eMule** per certificare la proprietà dei crediti di un client è previsto un sistema di Identificazione Sicura Utente, che si basa sull'utilizzo di firme digitali.

Le firme digitali (*digital signature*) richiedono la creazione di una coppia di chiavi: quella *pubblica* (**PK**, Public Key) utilizzata per la verifica della firma, e quella *privata* (**SK**, Secret Key) nota solo al mittente, che la usa per firmare il messaggio.

L'algoritmo che genera le chiavi è implementato in modo da rendere la chiave pubblica di un utente la sola in grado di decifrare correttamente i documenti cifrati con la sua chiave privata.

Creazione chiavi in eMule

In questo capitolo mostreremo come avviene la creazione della coppia di chiavi in **eMule**, facendo riferimento ai metodi nell'unico sorgente interessato: **`ClientCredits.cpp`**.

Primo avvio eMule

Al primo avvio di **eMule** viene lanciato il metodo `CClientCreditsList()` che a sua volta invoca la funzione di inizializzazione `InitalizeCrypting()`.

```
//ClientCredits.cpp

CClientCreditsList::CClientCreditsList()
{
   m_nLastSaved = ::GetTickCount();
   LoadList();
   InitalizeCrypting();
}
```

Verifica esistenza chiavi

Le informazioni sulle chiavi sono memorizzate nel file `cryptkey.dat`. La prima operazione del metodo `InitalizeCrypting()` è verificare se il file esiste già, e se sì

se è vuoto o no. Se esiste già si può passare alla fase di caricamento delle chiavi (descritta più avanti), altrimenti viene richiamata la funzione `CreateKeyPair()`.

```
// ClientCredits.cpp

void CClientCreditsList::InitalizeCrypting()
{
   // [...] verifica condizioni iniziali

   // check if keyfile is there
   bool bCreateNewKey = false;
   HANDLE hKeyFile = ::CreateFile(thePrefs.GetMuleDirectory(EMULE_CONFIGDIR) +
                           _T("cryptkey.dat"), GENERIC_READ, FILE_SHARE_READ, NULL,
                           OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);

   if (hKeyFile != INVALID_HANDLE_VALUE) // se non ci sono stati errori
                                         // nell'apertura...
   {
      if (::GetFileSize(hKeyFile, NULL) == 0) // ...ma il file è vuoto...
         bCreateNewKey = true; // ...ricorda di creare le chiavi
      ::CloseHandle(hKeyFile);
   }
   else // se invece ci sono stati errori nell'apertura...
      bCreateNewKey = true; // ...ricorda di creare le chiavi
   if (bCreateNewKey)
      CreateKeyPair();

// [...]
```

Creazione chiavi

Il metodo `CreateKeyPair()` genera la coppia di chiavi utilizzando un algoritmo RSA a 384 bit, e la memorizza nel file `cryptkey.dat`.

```
// ClientCredits.cpp

bool CClientCreditsList::CreateKeyPair()
{
   try{
      AutoSeededRandomPool rng;
      InvertibleRSAFunction privkey;
      privkey.Initialize(rng,RSAKEYSIZE);
      Base64Encoder privkeysink(new
                  FileSink(CStringA(thePrefs.GetMuleDirectory(EMULE_CONFIGDIR) +
                                                         _T("cryptkey.dat"))));
      privkey.DEREncode(privkeysink);
      privkeysink.MessageEnd();
      // [...] scrive nel log di sistema e ritorna il risultato
      // a InitalizeCrypting() [...]
}
```

Caricamento chiavi

Arrivati qui si ha la certezza che le chiavi esistano, o perché già esistenti o perché appena create.

Il metodo `InitalizeCrypting()` le recupera dal file `cryptkey.dat` memorizzando:

- la SK nella variabile `m_pSignKey`, di tipo `RSASSA_PKCS1v15_SHA_Signer` (la classe delle librerie *Crypto++* usata per firmare le *signature* in RSA);
- la chiave pubblica nella variabile `m_abyMyPublicKey`, ottenuta dal verificatore (classe `RSASSA_PKCS1v15_SHA_Verifier`) della chiave privata.

```
// ClientCredits.cpp

   // load key
   try{
      // load private key
      FileSource filesource(CStringA(thePrefs.GetMuleDirectory(EMULE_CONFIGDIR)
                                  + _T("cryptkey.dat")), true, new Base64Decoder);
      m_pSignkey = new RSASSA_PKCS1v15_SHA_Signer(filesource);
      // calculate and store public key
      RSASSA_PKCS1v15_SHA_Verifier pubkey(*m_pSignkey);
      ArraySink asink(m_abyMyPublicKey, 80);
      pubkey.DEREncode(asink);
      m_nMyPublicKeyLen = (uint8)asink.TotalPutLength();
      asink.MessageEnd();
   }

   // [...] gestione delle eccezioni nella try [...]
}
```

Fonti

eMule project
http://www.emule-project.net

eMule su SourceForce
http://sourceforge.net/projects/emule

Peer-to-Peer Computing: The Evolution of a Disruptive Technology
Ramesh Subramanian , Brian D. Goodman
2005

Application Layer Traffic Optimization in the eMule System
Lijie Sheng, Jianfeng Song, Xuewen Dong, Kun Xie
2010

Peer Sharing Behaviour in the eDonkey Network, and Implications for the Design of Server-less File Sharing Systems
S. B. Handurukande, A.-M. Kermarrec, F. Le Fessant, L. Massoulié, S. Patarin
2006

The eMule Protocol Specification
Yoram Kulbak, Danny Bickson
2005