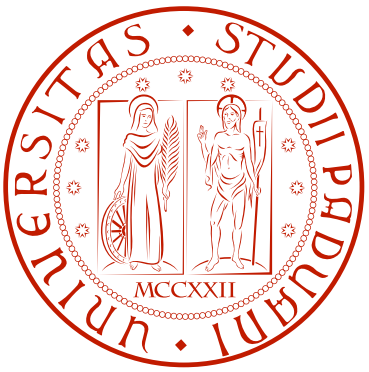
Università degli studi di Padova
Dipartimento di Ingegneria dell'Informazione



Corso di Laurea Triennale in Ingegneria dell'Informazione

PariKad: TCP Firewall Check, Findbuddy, Callback

Relatore: **Chiar.mo Prof. Enoch Peserico Stecchini Negri De Salvi**

Laureando: **Francesco Agnolazza**

Anno Accademico 2011-2012

Alla comunità.

Sommario

Il lavoro di questa tesi si inserisce all'interno del progetto software *PariPari* ed in particolare riguarda lo sviluppo di *PariKad*, parte integrante del plug-in *PariMulo*.

PariKad è un'implementazione in *Java* della rete distribuita *Kad*, una delle *DHT* ad oggi più diffuse che conta ogni giorno milioni di utenti connessi simultaneamente.

Trattandosi di software *Peer-To-Peer* dedicato al *filesharing*, è di primaria importanza il fatto che ogni utente possa connettersi al maggior numero di fonti possibile per poter scaricare i file desiderati il più velocemente possibile.

Sfortunatamente, spesso accade che per la poca praticità dell'utente con tali programmi o per impossibilità dovute a particolari configurazioni della rete locale, alcuni client risultino essere non raggiungibili direttamente.

Ecco che allora, in letteratura, si sono studiati particolari sistemi per far sì che questi client possano essere contattati comunque, anche in sistemi distribuiti come Kad, dove non esiste un server in grado di comunicare con tutti gli utenti.

L'intento di questo lavoro, dunque, è quello di fornire una panoramica generale sul funzionamento di tali meccanismi e di descrivere come sono stati implementati all'interno del plug-in PariMulo gli algoritmi adottati per la loro realizzazione: TCP Firewall Check, FindBuddy e Callback.

Indice

Capitolo 1

Introduzione

In questo capitolo vengono trattati gli argomenti generali in cui si inserisce il lavoro di questa tesi.

1.1 PariPari

PariPari è un progetto software open source scritto in *Java*, nato nel 2006, realizzato interamente da studenti laureandi e dottorandi del Dipartimento di Ingegneria dell'Informazione dell'Università di Padova. Si tratta di una rete multifunzionale *peer-to-peer* che offre al suo interno molte delle applicazioni e dei servizi Internet più diffusi al giorno d'oggi come *file sharing*, *VoIP*, *IM*[1], *web serving*, *distributed storage*, *DHT*[2], ecc.

L'idea originale che sta alla base del progetto è quella di creare un unico software in grado di gestire i programmi che utilizzano la rete, in modo da ottimizzare le risorse a disposizione della macchina su cui viene eseguito, come il consumo della banda e l'utilizzo della CPU.

Per capire meglio, si prenda ad esempio in considerazione l'utilizzo in concomitanza di un client *BitTorrent*[3] e di *Skype*: in genere il client BitTorrent satura la banda, con l'effetto collaterale di rendere impraticabili eventuali

[1] **I**nstant **M**essaging
[2] **D**istributed **H**ash **T**able
[3] Protocollo di file sharing.

chiamate attraverso Skype. Una tale situazione con un software organizzato come PariPari non si verificherà più, in quanto *PariCore*[4] gestirà in modo intelligente le risorse disponibili (in questo caso ridimensionando le velocità di download e upload del client BitTorrent per la durata della chiamata, rendendo così disponibile al client VoIP un flusso continuo di dati).

Grazie all'approccio architetturale scelto per la sua realizzazione, ovvero l'utilizzo di una struttura modulare a plug-in, PariPari è facilmente estendibile con l'aggiunta di nuovi servizi e applicativi.

Inoltre, PariPari è stato pensato per essere eseguito semplicemente con un click da una pagina web. Grazie alla portabilità di Java, esso risulta utilizzabile in qualunque computer che abbia a disposizione un browser, una *JVM*[5] e una connessione ad Internet (condizione pressoché soddisfatta per qualunque PC esistente), eliminando quindi la necessità di una procedura di installazione.

1.2 PariMulo

PariMulo è il plug-in di PariPari per cui è stato svolto il lavoro di questa tesi.

PariMulo fa parte dell'area file sharing del progetto e in particolare esso è un client *eMule*[6]. Inizialmente PariMulo supportava solo la rete *eDonkey*, ma di recente è stata implementata anche la rete *Kad*. I meccanismi che verranno descritti nei prossimi capitoli sono stati sviluppati al fine di completare il supporto della rete Kad e rendere PariMulo completamente compatibile con gli altri client.

A causa della scarsa documentazione reperibile in letteratura sull'implementazione dei protocolli eDonkey e Kad, è stato svolto un importante lavoro di *reverse engineering* sul codice sorgente di eMule, ritenuto l'unica fonte affidabile di informazione. I risultati ottenuti sono da considerarsi comunque buoni. Rispetto ad eMule, allo stato attuale dell'arte, le funzioni princi-

[4] Il modulo centrale di PariPari.

[5] *Java Virtual Machine*

[6] Software open source per filesharing

pali (come download, upload, preview, ecc.) sono pienamente supportate e in alcuni casi migliorate. Altre invece sono parzialmente realizzate o in via di sviluppo (come il *publishing* dei file nella rete Kad) e saranno presto disponibili.

1.3 Le reti eDonkey e Kad

La principale differenza tra la rete eDonkey e la rete Kad è la topologia.

La prima è una rete di tipo centralizzato: presuppone cioè l'esistenza di alcuni server centrali che hanno il compito di mantenere le informazioni degli utenti (indirizzi IP, file condivisi, ecc.) e di metterli in contatto tra di loro in funzione alle richieste.

La seconda, al contrario, è di tipo decentralizzato (o totalmente distribuito): tutti gli utenti partecipano alla realizzazione della rete e agiscono allo stesso tempo sia come client che come server.

Ogni topologia ha i suoi pro e i suoi contro. Ad esempio, in un una rete centralizzata, è molto più semplice e veloce ottenere informazioni riguardo una risorsa (si interpella solo il server) mentre la rete stessa è molto meno scalabile e *fault tolerant*. Infatti all'aumentare degli utenti il server ha sempre più difficoltà nel gestire tutte le richieste e, nel caso di guasto, migliaia di utenti non potrebbero più connettersi, isolando intere porzioni di rete o frazionando la rete in tante sottoreti più piccole.

D'altro canto, in una rete decentralizzata, le operazioni di *join*, ricerca e condivisione dei file non sono banali e richiedono che la rete sia organizzata in un modo particolare (come sarà accennato nella prossima sezione). Tuttavia, data l'assenza dei server, una rete decentralizzata non soffre delle criticità espresse prima, ovvero scarsa scalabilità e tolleranza ai guasti.

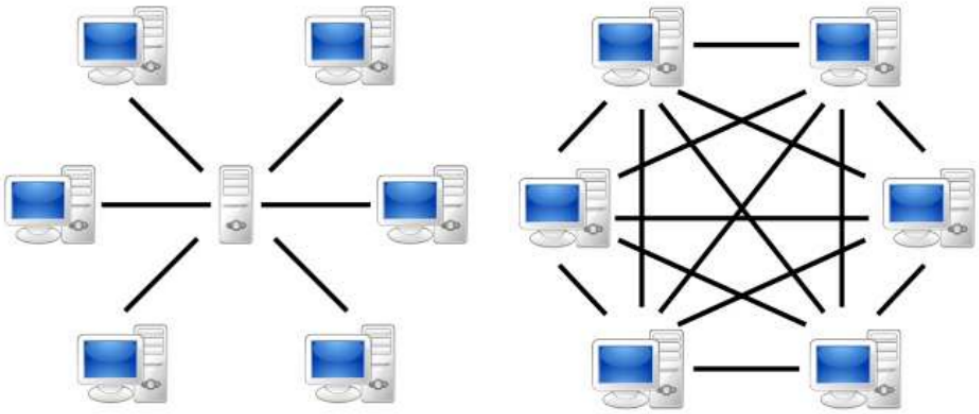


Figura 1.1: esempi di rete centralizzata (eDonkey) e totalmente distribuita (Kad)

1.3.1 Kad Overview

In questo paragrafo si forniscono alcune nozioni basilari e molto generiche sul funzionamento della rete Kad[7], utili per comprendere meglio alcuni passaggi dei prossimi capitoli.

La rete Kad si basa sull'algoritmo *Kademlia*[1], una DHT sviluppata da due ricercatori della New York University nel 2002, che definisce la struttura della rete e come i nodi che la compongono devono scambiarsi le informazioni tra di loro.

Per evitare ambiguità nel resto della tesi si introducono ora alcune definizioni:

- **nodo/peer:** singolo elemento della rete inteso come nodo paritario.
- **risorsa:** file, *note*[8], *keyword*[9] ecc;
- **client:** l'applicazione che permette di sfruttare le potenzialità della rete;

[7] Per maggiori dettagli consultare [3], [4].
[8] Commento
[9] Chiave di ricerca

- **utente:** la persona fisica che utilizza il client;

- **hash ID/Kad ID:** numero binario di 128 bit che identifica univocamente nodi[10] e risorse all'interno della rete. Può essere usato nelle accezioni di:

  file hash: identifica un file e viene calcolato secondo la funzione hash *MD4*[11];

  client ID/user hash: identifica un client e viene generato casualmente all'avvio del client.

Spazio 128-bit e Distanza

L'uso dell'Hash ID porta a definire uno spazio a 128-bit (chiamato anche *128-bit Key Space*) nel quale vivono i nodi (peer e risorse). Come conseguenza, nella rete Kad possono esserci fino ad un massimo di 2^{128} ID diversi. Tale numero, che apparentemente sembra una limitazione per la rete, in realtà è sufficientemente grande per rendere trascurabile[12] la probabilità che due entità assumano lo stesso ID nella rete.

Inoltre in questo spazio è stata definita una metrica per stabilire quanto due ID sono "vicini" tra loro. La distanza, scorrelata dalla distanza geografica, viene calcolata applicando la funzione logica *XOR* tra gli ID dei due elementi presi in considerazione.

Per esempio, nell'ipotesi esemplificativa di uno spazio a 4-bit, se l'ID di un nodo A è 1000 e l'ID di uno nodo B è 1011, la loro distanza è: *A XOR B = 0011*.

Il motivo per cui viene introdotto il concetto di distanza è legato alla struttura della DHT. In particolare, ogni nodo-peer diviene il responsabile della gestione delle risorse che sono vicine ad esso. In questo modo vengono

[10] **N.B.** Peer e risorse vengono identificate nello stesso modo e quindi non sono distinguibili a priori.

[11] Funzione non invertibile che prende in input un insieme di bit (può essere una stringa come un file) e restituisce in output un numero binario di lunghezza fissa 128 bit.

[12] Per il calcolo esatto, si veda [3] par. 3.1

garantiti il mantenimento e la ricerca efficiente delle risorse.

Lookup e Ricerca

Il *lookup* è quella procedura che localizza nella rete i peer responsabili di una data risorsa: dato un Kad-ID, il lookup è lo strumento utilizzato per ottenere le informazioni (IP, porta UDP/TCP) sui peer più vicini a quell'ID.

Si noti che nel processo di lookup si utilizza solo ed esclusivamente l'ID della risorsa e quello che viene restituito è una lista di contatti. Tutto ciò che riguarda il tipo e le altre informazioni legate alla risorsa (quali possono essere ad esempio per un file: possessore, dimensione, formato, ecc.) vengono ricavate nel momento in cui si esegue una **ricerca**, che viene effettuata inviando richieste ai contatti che si trovano nella *tolerance zone*[13].

Infatti, una volta ottenuti i contatti dalla funzione di lookup, si procede inoltrando le richieste di ricerca (es. `PacketUDPKad2SearchKeyRequest(...)`) che possono essere di svariati tipi a seconda dell'obiettivo (keyword, source, file, publishing, note, findbuddy, ecc.).

Publishing

Il *publishing* è quel meccanismo che permette di pubblicare le informazioni relative ai file condivisi da un utente.

Essendo Kad una rete *serverless*[14], queste informazioni non possono risiedere su un server come accade con la rete eDonkey, ma dovranno necessariamente essere mantenute dagli altri utenti che in quel momento sono connessi. Questo significa che ogni utente manterrà in un'apposita struttura dati le informazioni relative ad una certa risorsa/file posseduto da qualche altro peer.

Questa DHT è poi progettata in modo tale da garantire una certa ridondanza nelle informazioni memorizzate, cosicché anche se un nodo è discon-

[13] Corrisponde all'insieme dei peer che probabilmente conoscono la risorsa indicata come obiettivo del lookup.

[14] Senza server.

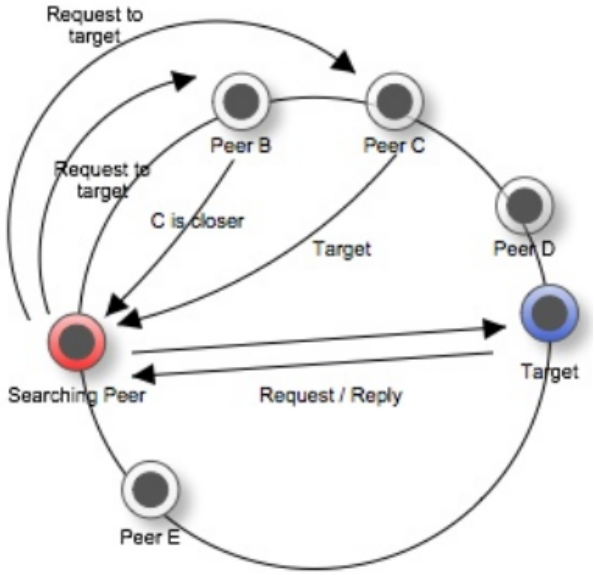


Figura 1.2: Lookup Iterativo: il Searching Peer (A) avvia il lookup andando a chiedere al peer B se è responsabile della risorsa cercata o se tra i suoi contatti ce né qualcuno con distanza minore. Il peer B risponde notificando che C è più vicino, per cui invia ad A le informazioni necessarie per contattare C. C viene contattato da A e risponde che Target è il più vicino. A quel punto A contatta direttamente Target per le richieste.

nesso, le sue informazioni sono ugualmente reperibili da un suo vicino che ne conserva una copia.

Precisamente, ogni qualvolta si desidera condividere un file, si chiama la funzione publishing, che a sua volta invoca il lookup con il Kad ID della risorsa come target. Successivamente, ai contatti restituiti come risultato dal lookup vengono inviati i pacchetti `PacketUDPKadPublishKeyRequest()`, `PacketUDPKadPublishSourceRequest()` ecc. contenenti tutte le informazioni del file inserite come TAG. Questi dettagli saranno poi salvati da ognuno di essi.

Come si vedrà nel Capitolo 5 relativo alla Callback, il publishing sarà di fondamentale importanza per comunicare al resto della rete come contattare peer che sono in una condizione di non-raggiungibilità ad esempio a causa di un *NAT*[15].

[15] ***N**etwork **A**ddress **T**ranslator* vedi sezione 2.1

Capitolo 2

Fasi della Callback

In questo capitolo verrà illustrata l'idea che sta alla base del meccanismo della Callback sia in eDonkey (per completezza) che in Kad. Inoltre verranno presentati gli elementi utilizzati dal TCP Firewall Check e dal Findbuddy, introdotti in PariMulo al fine di poter realizzare la Callback nella rete Kad.

2.1 Principio di funzionamento della Callback

In una rete peer-to-peer dedicata al file sharing come eDonkey e Kad dove i file sono distribuiti tra tutti i componenti della rete, è di fondamentale importanza il fatto che ogni utente possa connettersi al maggior numero di fonti possibile per poter scaricare i file desiderati il più velocemente possibile. Ciò significa che, in uno scenario ideale, ogni utente connesso dovrebbe poter raggiungere e dovrebbe poter essere raggiunto da chiunque. Questo però si sa non essere vero.

Il problema

Al giorno d'oggi a causa della larga diffusione di pc, laptop, tablet e dispositivi mobili la maggior parte dei tipici utenti che fanno uso del file sharing si trova a possedere in casa più device in grado di sfruttare una connessione ad Internet. Per connetterle contemporaneamente allora, viene utilizzato

un *router* che nella maggior parte dei casi implementa anche il NAT, ossia maschera gli indirizzi IP privati della LAN agli altri computer presenti nella WAN, permettendo così la gestione di più accessi contemporanei alla rete con uno stesso IP pubblico (SNAT[1]).

Di default, se non opportunamente configurato, un router non esegue il *port forwarding*[2] dei pacchetti TCP/UDP. Questo fatto, per un software come PariMulo, si traduce in una condizione di non raggiungibilità che ha degli effetti negativi per il resto della rete. Per esempio viene negata la possibilità di scaricare risorse provenienti da un utente con queste limitazioni. D'ora in avanti questo tipo di utenti verranno chiamati *firewalled*[3].

La Callback

Anche gli utenti *firewalled* possono essere contattati grazie alla Callback. Il meccanismo della Callback ha il compito di ribaltare in un certo senso i ruoli: se qualcuno vuole contattare un peer che non può essere contattato (firewalled), sarà quest'ultimo a contattare il primo. Come? La chiave sta nell'avere un terzo "elemento della rete[4]" che abbia già stabilito una connessione con il peer firewalled e sia disposto ad inoltrargli le richieste.

In generale per predisporsi alla Callback si possono distinguere tre fasi:

- **Fase 1:** capire se il client che si sta utilizzando è o meno raggiungibile dell'esterno (condizione di porte chiuse o firewalled).

- **Fase 2:** nel caso in cui dalla *Fase 1* sia emerso uno stato di non raggiungibilità, preoccuparsi di trovare un *relay* tra i due peer in grado di inoltrare le richieste per effettuare la Callback.

[1] **S**ource NAT

[2] Tecnica che permette di instradare tutti i pacchetti ricevuti su una particolare porta di un IP pubblico verso un terminale specifico della rete interna.

[3] Si noti che l'uso del termine *firewalled* non è propriamente corretto, non essendoci in realtà alcun firewall.

[4] Un server (rete *eDonkey*) o un altro peer (rete *Kad*).

- **Fase 3:** informare il resto della rete su chi è il relay.

Si vedrà ora come le tre fasi sono implementate nelle due reti.

2.1.1 Callback in eDonkey

Nei sistemi *centralizzati* come eDonkey viene sfruttata la presenza dei server che agiscono come relay. Ovviamente, affinché la callback possa avvenire, il peer che richiede la callback e il peer firewalled a cui essa è rivolta devono essere connessi allo stesso server.

Preparazione

La **Fase 1** avviene durante il processo di *login* del client. Infatti durante questo task il server al quale ci si sta connettendo assegna al client un *High ID* o un *Low ID* in funzione della raggiungibilità.

Per quanto riguarda la **Fase 2** sarà il server stesso ad occuparsi di inoltrare le richieste in caso di Low ID.

La **Fase 3** in eDonkey non è propriamente presente perché il server nelle risposte delle interrogazioni fornisce le liste dei peer con i rispettivi ID. In questo modo i client riescono a dedurre se l'utente è direttamente raggiungibile o se è necessario eseguire una Callback, a seconda che l'ID sia *High* o *Low*.

Funzionamento

Supponiamo che un peer A con *High ID* voglia scaricare un file condiviso da un peer B con *Low ID* (vedere Figura 2.1). Il peer A deve inviare al server un messaggio con la richiesta di Callback verso il peer B - *Callback req.* Il server a sua volta inoltra a B la richiesta - *Callback req forwarded* (ciò è possibile perché B è già connesso al server). A questo punto B, ricevuto il messaggio dal server, si attiva per connettersi ad A e soddisfare le sue richieste - *Callback TCP conn.*

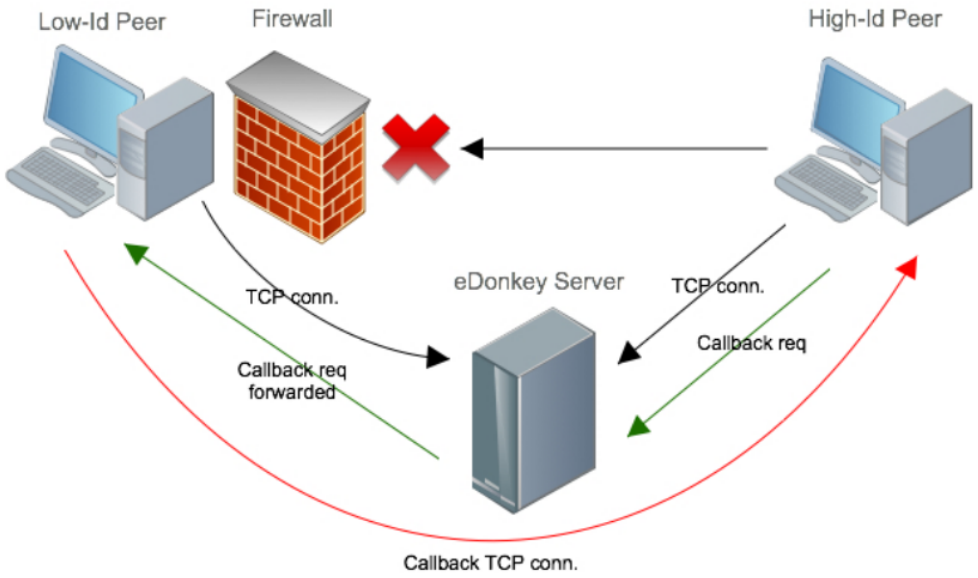


Figura 2.1: esempio di Callback in eDonkey

2.1.2 Callback in Kad

Essendo Kad una rete serverless, il lavoro che in eDonkey viene interamente svolto dal server a cui un peer è connesso (test raggiungibilità e inoltro richieste in caso di non-raggiungibilità), in Kad dev'essere eseguito inevitabilmente dai peer che in quel momento popolano la rete.

Preparazione

Nella **Fase 1** avviene il *firewall check*: il client comincia ad inviare via UDP delle richieste chiamate *Firewall Check Request* ad alcuni peer della rete che, dopo aver risposto via UDP, tenteranno di instaurare una connessione TCP con esso. Se il tentativo va a buon fine viene inviato via TCP un pacchetto di *acknowledgment* che conferma la raggiungibilità del client. Nel caso in cui invece dal client non venga ricevuto alcun pacchetto di acknowledgment, PariMulo determina lo stato di non-raggiungibilità. Se dopo un determinato periodo di tempo lo stato del client è ancora di non-raggiungibilità viene avviata la *Fase 2* detta anche di *find buddy*.

Nella **Fase 2**, non essendoci il server, si cerca un peer[5] della rete disposto ad inoltrare le richieste dirette al client *firewalled*.

Come da protocollo le richieste (*Findbuddy Request*) vengono mandate via UDP ad alcuni utenti connessi, i quali, se dispongono dei requisiti necessari (devono cioè essere in uno stato *non-firewalled* e non devono essere già il buddy di nessun altro), rispondono candidandosi automaticamente ad aspiranti buddy. Il client PariMulo allora tenterà di stabilire una connessione TCP con uno di essi. Se la connessione va a buon fine, da quel momento in avanti quel peer sarà il buddy e sarà il responsabile dell'inoltro delle richieste (Callback) dirette al client provenienti dal resto della rete.

Per la **Fase 3** una volta ottenuto il buddy si procederà con il *publishing* dei file condivisi, inserendo tra i *TAG* anche IP e porta TCP del buddy.

Riassumendo:

1. si individua se il client PariMulo è raggiungibile dall'esterno;
2. in caso negativo, il client comincia la ricerca di un buddy valido, inviando via UDP i *Findbuddy Request*;
3. se un peer risponde, si prova ad instaurare con esso una connessione TCP: nel caso in cui vada a buon fine, questo sarà il buddy del client PariMulo;
4. infine, attraverso il publishing, si pubblicano i file inserendo nei TAG IP e porta TCP del buddy.

Funzionamento

Il funzionamento è pressoché identico a quello della rete eDonkey, dove però il lavoro del server viene svolto dal buddy.

Si osservi la figura 2.2: un peer C interessato ad una risorsa posseduta dal client A (*Peer Firewalled*) invia una *Kad Callback Request* al buddy

[5]Nel resto di questo lavoro, questo peer verrà chiamato con il nome di **buddy**

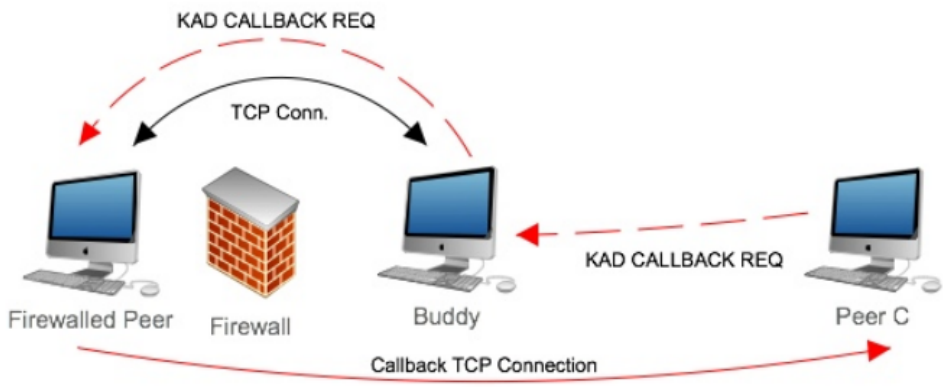


Figura 2.2: esempio di Callback in Kad

B. Il buddy B inoltrerà al client A la richiesta e a questo punto sarà il client A a contattare via TCP il peer C per l'upload del file - *Callback TCP connection*.

2.2 Nuovi elementi introdotti

Ora che è più chiaro il principio che sta alla base del funzionamento della Callback per la rete Kad e si conoscono le procedure necessarie per poterla attuare (TCP Firewall Check e Findbuddy), si introducono gli elementi usati in PariMulo per gestire tali processi.

Per il TCP Firewall Check e il Findbuddy si è deciso di adottare la stessa soluzione utilizzata dal software eMule.

L'idea di fondo è quella di realizzare una sorta di "macchina a stati" (Figura 2.3): il client associa ad ogni peer uno *stato* e, in funzione del verificarsi o meno di certe condizioni, lo *stato* può cambiare dando luogo ad eventi correlati al peer stesso.

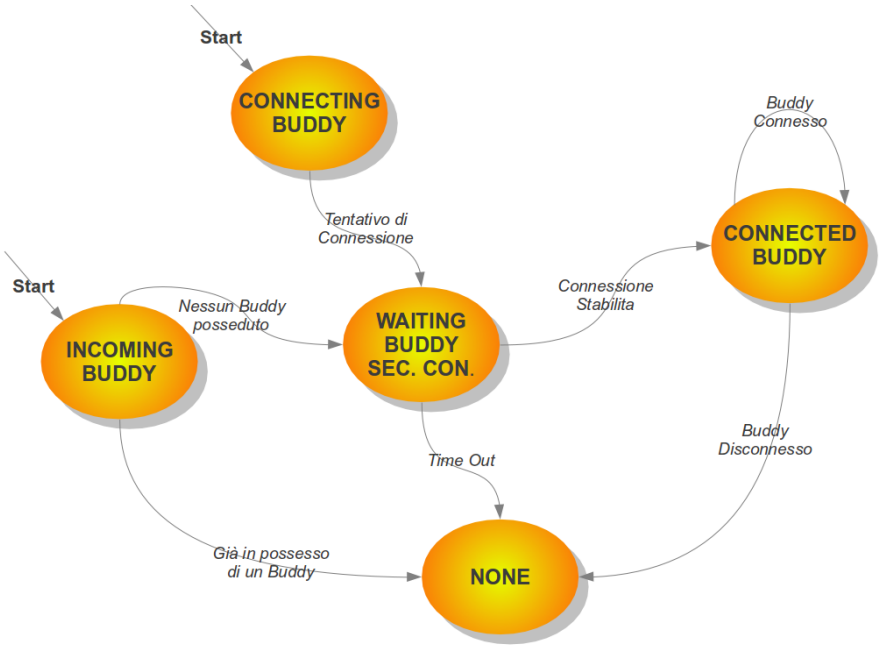


Figura 2.3: macchina a stati per il Findbuddy

2.2.1 KadState

Per introdurre in PariMulo il concetto di "stato del peer" di cui sopra si è inserito nella classe `Peer` del pacchetto `Entities` il campo *KadState*.

Come tipo di dato il KadState è un *enum*[6], che al momento conta 8 valori possibili.

Lo stato non viene assegnato indistintamente a tutti i peer noti al client ma come sarà chiaro più avanti, viene attribuito solamente a quelli che contribuiscono al TCP Firewall Check o contribuiscono al Findbuddy.

C'è poi una struttura dati dedicata in cui verranno inseriti tali peer, la *kadList*[7], che si occuperà di processarli ciclicamente, andando ad impostare il nuovo stato in base allo stato attuale e all'evento verificatosi.

[6] *Enumerated type*: è un tipo di dato che consiste in un insieme di valori (tipicamente identificatori) chiamati elementi che si comportano come costanti.

[7] Vedi par. 2.2.2

Gli stati presenti al momento sono:

- `ASKED_TCP_TW_CHECK`: il peer ha chiesto un TCP Firewall Check;
- `WAITING_TCP_FW_CHECK`: il peer sta aspettando un TCP Firewall Check;
- `TCP_FW_CHECKED`: il TCP Firewall Check è stato completato e il peer può essere eliminato dalla lista;
- `INCOMING_BUDDY`: il peer ha bisogno di un buddy;
- `CONNECTING_BUDDY`: il peer è candidato ad essere un buddy;
- `WAITING_BUDDY_SECURED_CONNECTION`: il peer sta completando il processo per stabilire la connessione;
- `CONNECTED_BUDDY`: il peer è connesso;
- `NONE`: il peer può essere eliminato dalla lista;

2.2.2 KadList

La struttura dati principale che è stata introdotta nel codice del plug-in PariMulo con questo lavoro è la `KadList`: una `ConcurrentHashMap<InetAddress, Peer>` definita nella classe *Kad* e utilizzata per la gestione delle richieste di firewall check e find buddy.

Il fatto che sia stata scelta una HashMap *simultanea* deriva dalla necessità di dover garantire la consistenza della lista evitando le `ConcurrentModificationException`[8]. Infatti, in seguito a vari test effettuati, si è potuto constatare che utilizzando una normale HashMap i casi di conflitto sono molto frequenti, in particolar modo durante la ricerca del buddy.

[8] Questo tipo di eccezioni viene lanciato nel momento in cui due o più metodi distinti tentano di accedere contemporaneamente alla lista.

La *kadList* viene popolata dai metodi `process()` dei pacchetti Firewall-Request e FindbuddyRequest/Response tramite `addPeerToKadList(peer)` e viene processata dal metodo `processKadList()` che si trova in *Kad.java*.

Per evitare di sprecare risorse si è optato per creare un nuovo thread - `KadListThread` - che avvia periodicamente il processing della lista solo se essa risulta essere non vuota. Durante il loop, per mezzo di uno *switch* - *case*, viene scandito ed elaborato ogni singolo peer in funzione del suo KadState. Di seguito viene presentata una descrizione delle possibili transizioni:

- `case WAITING_TCP_FW_CHECK`: invia al peer un `PacketTCPKadFirewall CheckAck()` e passa allo stato TCP_FW_CHECKED;
- `case TCP_FW_CHECKED`: peer controllato, passa allo stato NONE;
- `case INCOMING_BUDDY`: se il client è già connesso ad un buddy, passa allo stato NONE. Altrimenti passa allo stato WAITING_BUDDY_SECURED_CONNECTION;
- `case CONNECTING_BUDDY`: tentativo di connessione al peer (candidato buddy), successivamente passa a WAITING_BUDDY_SECURED_CONNECTION;
- `case WAITING_BUDDY_SECURED_CONNECTION`: si attende che venga stabilita la connessione con il peer. In caso affermativo si passa allo stato CONNECTED_BUDDY. In caso di *time out* si cancella il peer dalla lista;
- `case CONNECTED_BUDDY`: se non si è già connessi ad un buddy, si salva questo peer come `myBuddy`. Altrimenti si verifica se la connessione con il buddy è ancora attiva. In caso negativo: se si necessita di un altro buddy si avvia una nuova ricerca, altrimenti si elimina `myBuddy` e non si fa nulla.
- `case NONE`: elimina il peer dalla lista;

Ora che sono note le basi dei meccanismi di firewall check e findbuddy si procederà con l'illustrare la loro implementazione nel plug-in PariMulo entrando nel dettaglio del codice.

Capitolo 3

TCP Firewall Check

Verranno qui introdotti alcuni dettagli di implementazione del meccanismo di TCP Firewall Check.

3.1 Il funzionamento: sender e receiver

Sender

Il *sender* è colui che vuole sapere se la sua porta TCP è aperta e raggiungibile dall'esterno (Peer A nella Figura 3.1).

La verifica dello stato di raggiungibilità avviene ogni qualvolta si avvia il thread Kad e viene ripetuta ad intervalli fissi di un'ora di durata.

Le richieste (`PacketUDPKadFirewalled2Request()`) vengono mandate dal sender ai peer che gli inviano o i `PacketUDPKad2HelloResponse()` o i `PacketUDPKad2HelloRequest()` e contengono (Tabella 3.1):

- la porta TCP del sender;
- il Kad ID del sender;
- un bit che specifica alcune opzioni di connessione[1]

[1] Al momento non supportato da PariMulo e quindi irrilevante

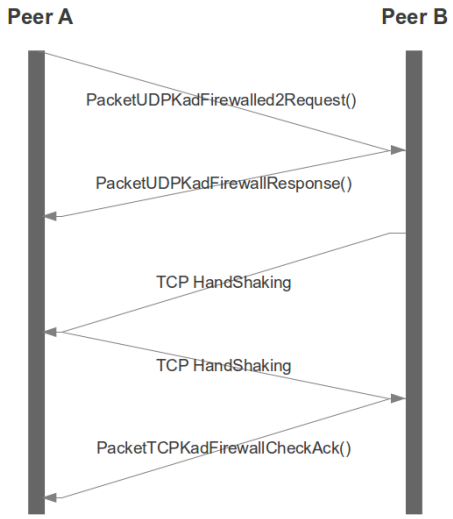


Figura 3.1: sequenza pacchetti nel TCP Firewall Check

Campo	Size [B]	Contenuto
`Protocol`	1	0xE4 (Kad protocol)
`Type`	1	0x53 (TCP Firewall Check REQ OPcode)
`Port`	2	*Sender* TCP Port
`Hash ID`	16	*Sender* Kad ID
`Conn.Opt`	1	Boolean

Tabella 3.1: PacketUDPKadFirewalled2Request()

Poiché di *Kad HelloRequest/Response* se ne ricevono in continuazione per l'intera durata dell'esecuzione di PariMulo, si è introdotto un contatore che ha la funzione di tenere monitorato il numero di *Firewall Response* ricevute. In questo modo, una volta ottenuto un numero[2] sufficiente di risposte, si blocca l'invio delle *Firewall Request*.

Pertanto, nel momento in cui si riceve una risposta (`PacketUDPKadFirewalledResponse()`), si va ad incrementare con il metodo `incRecheckIP()` il contatore delle risposte ottenute. Queste corrispondono ai peer della rete che hanno ricevuto la richiesta di firewall check e si sono resi disponibili ad effettuare il test.

Una volta ricevute le 4 risposte si interrompe l'invio delle *firewall check request* così da evitare di sprecare inutilmente la banda disponibile.

Arrivato a questo punto, il richiedente deve solo attendere di essere contattato via TCP dai client che hanno risposto e aspettare i pacchetti di acknowledgment (`PacketTCPKadFirewallCheckAck()`).

Nel caso il client sia raggiungibile quello che avviene è il *TCP handshacking* e successivamente la ricezione del suddetto pacchetto. Nel metodo `process()` di tale pacchetto viene chiamato il metodo `incFirewalled()` che aumenta di un'unità la costante `firewalled`, inizialmente impostata a 0. Quando il sender riceve correttamente due `PacketTCPKadFirewallCheckAck()`, cioè `firewalled` raggiunge il valore 2, stabilisce che è raggiungibile dall'esterno. In caso contrario, rimane in una condizione di non raggiungibilità.

Receiver

Il *receiver* è colui che riceve e gestisce il `PacketUDPKadFirewalled2Request()` (Peer B nella Figura 3.1).

Nel metodo `process()` del pacchetto come prima cosa viene creato un oggetto di tipo *Peer* con i dati del *sender* (ricavabili dalla ricezione del `PacketUDPKadFirewalled2Request()`): IP, porta TCP, porta UDP, Kad

[2]In eMule e di conseguenza anche in PariMulo, questo numero è stabilito dalla costante KAD_FIREWALL_CHECKS che è impostata a 4

Campo	Size [B]	Contenuto
`Protocol`	1	0xE4 (Kad protocol)
`Type`	1	0x58 (TCP Firewall Check RES OPcode)
`IP`	4	*Sender* IP

Tabella 3.2: PacketUDPKadFirewalledResponse()

Campo	Size [B]	Contenuto
`Protocol`	1	0xC5 (eDonkey protocol)
`Packet Size`	4	0x00000001
`Type`	1	0xA8 (TCP Firewall Ack OPcode)

Tabella 3.3: PacketTCPKadFirewallCheckAck()

ID. All'oggetto Peer viene poi assegnato il *KadState* ASKED_TCP_FW_CHECK e viene inserito nella *KadList*.

Successivamente, il receiver invia al sender la risposta `PacketUDPKadFirewallResponse()`, via UDP, formata semplicemente dall'IP del sender (Tabella 3.2). Infine tenta di stabilire la connessione TCP con `peer.connect()`.

Nel metodo `process()` di `PacketTCPHelloResponse()` c'è una funzione (`checkKadList(peer)`) che controlla se l'utente che ha inviato la *TCP Hello Response* è presente nella *KadList*. Se la ricerca ha successo e il KadState dell'elemento restituito è ASKED_TCP_FW_CHECK, allora significa che il peer ha risposto, è raggiungibile e si può procedere con il test. Di conseguenza lo stato del peer nella lista viene cambiato in WAITING_TCP_FIREWALL_CHECK e, quando `processList()` arriva a processare tale elemento, viene inviato il pacchetto di acknowledgment (Tabella 3.3).

Spedito il pacchetto, la connessione con l'utente viene chiusa.

Capitolo 4

FindBuddy

Ora verrà descritto come in PariKad sono state implementate le procedure della **fase 2** di ricerca del buddy.

4.1 Ricerca e gestione del buddy

L'unico modo che un client *firewalled* ha per comunicare con un peer utilizzando il protocollo TCP è quello di essere lui stesso ad inizializzare la connessione. Se così non fosse, la connessione non può essere instaurata tra i due utenti. A tal scopo, il protocollo prevede che prima vengano inviate una serie di richieste - *FindBuddyRequest* - ad alcuni peer della rete, successivamente, i peer che inviano la risposta - *FindBuddyResponse* - vengono inseriti nella *kadList* e contattati uno ad uno. Il primo con il quale il tentativo di connessione TCP va a buon fine diventerà il buddy del client e il client, di conseguenza, il suo[1].

Per poter rispondere alle FindBuddyRequest, i peer devono soddisfare i seguenti requisiti:

1. non essere nello stato di raggiungibilità *firewalled*;
2. non avere un buddy già assegnato;

[1]Come in eMule, il termine *buddy* è usato in modo relativo: i peer che richiedono il servizio, indicano con il termine buddy chi lo offre e viceversa.

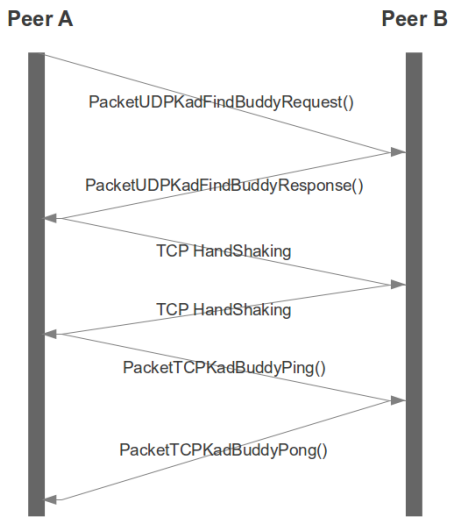


Figura 4.1: sequenza dei pacchetti scambiati nel Findbuddy

Scendendo nel dettaglio del codice, questo è quello che accade:

Sender:

L'avvio della ricerca del buddy viene gestito all'interno del thread della classe Kad. Precisamente, con il metodo `process()` viene continuamente controllato lo stato di raggiungibilità del client. Se, trascorsa una costante di tempo[2] dall'istante in cui viene avviato il thread Kad, lo stato risulta essere ancora *firewalled*, viene chiamato il metodo `KadPrefs.setFindBuddy(true)` che imposta la variabile `findBuddy` in KadPrefs.java a `true`. In questo modo, quando nel metodo `processKadList()` vengono controllate le condizioni dei seguenti metodi booleani

- Kad.isConnected();
- KadPrefs.getFirewalled();

[2]La costante di tempo è espressa dalla variabile `m_tNextFindBuddy_millis` e il valore attualmente è impostato a 5 minuti.

Request	Response
Peer A esegue una ricerca dei peer disponibili a fargli da buddy, inviando richieste buddy via UDP	Peer B riceve una richiesta buddy via UDP da Peer A. Se B ha le porte aperte e non è il buddy di nessun altro, allora Peer B risponde a Peer A
Peer A riceve risposta da Peer B e prova a connettersi via TCP ad esso	Peer B riceve e instaura una connessione TCP con Peer A
Se la connessione va a buon fine, Peer B è il buddy di Peer A	Se la connessione va a buon fine, Peer A è il buddy di Peer B
Peer A testa se la connessione con B è attiva tramite un Ping	Peer B, se è ancora il buddy di A, risponde con un Pong

Tabella 4.1: descrizione della sequenza dei pacchetti scambiati tra un Peer A *firewalled* e un Peer B non *firewalled* di Figura 4.1

Campo	Size [B]	Contenuto
`Protocol`	1	0xE4 (Kad protocol)
`Type`	1	0x51 (Findbuddy opcode)
`Kad ID`	16	Kad ID
`Hash ID`	16	User Hash
`Port`	2	TCP port

Tabella 4.2: Pacchetto PacketUDPKadFindBuddyRequest

- KadPrefs.getFindBuddy();

esse risulteranno essere tutte e tre verificate. Ciò significa che il client è connesso alla rete Kad, è nello stato *firewalled* e non possiede ancora un buddy. Diventa quindi necessario avviare il lookup per trovare i potenziali contatti vicini al client, disposti a fargli da buddy. Questo viene fatto creando un oggetto di tipo ricerca: `KadFindBuddySearch fbsearch = new KadFindBuddySearch(Config.getKadID())` con target il Kad-ID del client e chiamando `fbsearch.perform()` per far partire la ricerca.

Ottenuta la lista dei peer, viene inviato loro il pacchetto `PacketUDPKadFindBuddyRequest()` formato da (Tabella 4.2):

- il KadID del client (Int128);
- lo userHash del client (Int128);
- la porta TCP del client (int (castata a 2 byte));

Una volta ricevute le risposte PacketUDPKadFindBuddyResponse, che sono nella forma (Tabella 4.3):

- il KadID del client(Int 128);
- lo userHash del peer che ha risposto (Int128);
- la porta TCP del peer che ha risposto (int (castata a 2 byte));

Campo	Size [B]	Contenuto
`Protocol`	1	0xE4 (Kad protocol)
`Type`	1	0x5A (Findbuddy opcode)
`Kad ID`	16	Kad ID
`Hash ID`	16	User Hash
`Port`	2	TCP port

Tabella 4.3: Pacchetto PacketUDPKadFindBuddyResponse

si opera nel seguente modo:

1. viene verificato che il KadID contenuto nel pacchetto di risposta sia uguale al KadID del client, per assicurarsi che chi ha risposto sia un peer realmente contattato dal client: `Config.getKadID().equals(this.buddyId)`;

2. se sono uguali, si procede con la creazione di un oggetto Peer (possibile futuro buddy) utilizzando i dati ottenuti dalla ricezione del pacchetto `PacketUDPKadFindBuddyResponse`:

   - IP;
   - userHash;
   - TCP port;
   - UDP port;

3. infine viene chiamato il metodo `FindBuddy.requestBuddy(peer)`. Chiamando questo metodo si assegna il KadState `CONNECTING_BUDDY` al peer: `peer.setKadState(KadState.CONNECTING_BUDDY)` e si aggiunge il peer alla *kadList*: `Kad.addPeerToKadList(peer)`.

Da questo momento in poi la gestione viene affidata al metodo che scandisce la *kadList*. Infatti, se durante la scansione si trova un peer con KadState `CONNECTING_BUDDY`, si tenta di connettersi ad esso, utilizzando il me-

todo `peer.connect(false)`. Se la connessione va a buon fine, si cambia il KadState del peer corrente da `CONNECTING_BUDDY` a `CONNECTED_BUDDY`.

Una volta che il peer ha come KadState `CONNECTED_BUDDY`, lo si salva come buddy `FindBuddy.setMyBuddy(peer)` e gli si imposta il KadState a `NONE` in modo da poterlo eliminare dalla lista all'iterazione successiva.

Ovviamente una volta salvato il primo buddy i peer presenti in quel momento nella *kadList* vengono tutti eliminati e si termina il thread.

Response:

Nel caso invece sia il client PariMulo a ricevere un pacchetto `PacketUDPKadFindBuddyRequest`, se non è nello stato *firewalled* e non possiede già un buddy, crea un oggetto Peer (possibile futuro buddy) utilizzando i dati ottenuti dalla ricezione del pacchetto `PacketUDPKadFindBuddyRequest`:

- IP;
- KadID;
- userHash;
- TCP port;
- UDP port;

in seguito, chiama il metodo `FindBuddy.incomingBuddy(peer)` e invia la risposta `PacketUDPKadFindBuddyResponse(this.userId)` costruita come sopra:

- il KadID (Int 128) di chi ha inviato la *Request*;
- lo userHash del client (Int128);
- la porta TCP del client (int (castata a 2 byte));

Il metodo `FindBuddy.incomingBuddy(peer)` non fa altro che impostare il KadState del peer in `INCOMING_BUDDY` e aggiungere il peer alla *kadList*.

Se nella scansione della *kadList* un peer ha il KadState impostato a `INCOMING_BUDDY`, viene verificato che tra il client e questo peer esista effettivamente una connessione TCP: `peer.isSecured()`. In caso affermativo, viene cambiato il KadState in `CONNECTED_BUDDY` e, come prima, viene salvato questo peer come buddy del client.

Si osservi che una volta salvato un buddy tutti gli altri pacchetti di richiesta `PacketUDPKadFindBuddyRequest` vengono scartati, poiché il protocollo prevede che ogni peer abbia al più un buddy.

Ping - Pong

Per verificare se la connessione tra il peer e il buddy sia effettivamente attiva, sono stati introdotti i pacchetti `PacketTCPKadBuddyPing` e `PacketTCPKadBuddyPong`. Questi pacchetti vengono scambiati via TCP tra il client e il suo buddy ogni 7 minuti circa. Se dovesse trascorrere più tempo del previsto, il metodo `FindBuddy.isBuddyAlive()`, presente nel *case* `CONNECTED_BUDDY` della *kadList*, fallirebbe il test che verifica se sono trascorsi più di 10 minuti dall'ultimo invio/ricezione di un pacchetto *ping-pong*. In questo caso, allora, il client farebbe il reset di tutte le variabili legate al buddy e si porterebbe nelle condizioni di poter avviare una nuova ricerca.

Capitolo 5

Callback

5.1 Publishing

Una volta che un client *firewalled* ha ottenuto un buddy è necessario rendere pubblica questa informazione per permettere a chi volesse contattare il client, di raggiungerlo.

Questo avviene, come accennato precedentemente, attraverso il publishing dei file che si condividono.

I dati riguardanti IP e porta TCP del buddy vengono inseriti nel pacchetto del publishing della sorgente (`PacketUDPKadPublishSourceRequest`). Essi vengono inseriti in particolari *TAG*, specifici per peer *firewalled*. In questo modo un utente che tenta di scaricare un file posseduto da un peer non raggiungibile perché *firewalled*, quando processa i risultati ottenuti dalla ricerca della risorsa, riesce ad estrarre le informazioni utili per contattare il buddy.

In PariMulo, la definizione dei TAG da inserire nel pacchetto avviene nel metodo `kadPublishFile(Shared file)` che si trova nella classe `mulo.file.Shared`. In particolare, i TAG che riguardano il buddy sono:

- `KAD_SERVER_IP` IP del buddy;
- `KAD_SERVER_PORT` porta TCP del buddy;

Campo	Size [B]	Contenuto
`Protocol`	1	0xE4 (Kad protocol)
`Type`	1	0x44 (Publish opcode)
`Kad ID`	16	File ID
`Kad ID`	16	Client ID
`List`	var	TagList

Tabella 5.1: Pacchetto PacketUDPKadPublishSourceRequest

La procedura di publishing su PariKad al momento non è ancora stata automatizzata. Per poter pubblicare un file si deve perciò lanciare dalla console di PariMulo il comando `shared.publish id` dove *id* rappresenta il numero associato al file nella lista dei file condivisi (non il KadID del file).

5.2 Kad Callback

Come si è detto precedentemente, un client scopre se una data risorsa proviene da un peer *firewalled* quando processa i dati ottenuti dalla ricerca della stessa. Infatti, nei pacchetti che contengono i dati relativi al peer che possiede il file è presente un TAG chiamato `KAD_SOURCE_TYPE` che, a seconda del valore assunto, indica lo stato di raggiungibilità del peer possessore della risorsa. Nel caso di peer *firewalled* questo TAG contiene il valore 0x03.

Il client, ora consapevole del fatto che il peer che possiede la risorsa non è raggiungibile direttamente, ricava IP e porta TCP del buddy dai TAG KAD_SERVER_IP e KAD_SERVER_PORT e contatta il buddy, inviandogli, via UDP la richiesta di callback (`PacketUDPKadCallbackRequest(Int128 clientId, Int128 fileId)`) contenente il KadID del peer da contattare e il KadID del file da scaricare.

Il buddy, ricevuta la richiesta, la inoltra via TCP (Tabella 5.2) al peer *firewalled* (suo buddy). Questo fatto è reso possibile dalla connessione preesistente tra i due:

Campo	Size [B]	Contenuto
`Protocol`	1	0xE3 (eDonkey protocol)
`Packet Size`	4	38
`Type`	1	0x99 (Firewall opcode)
`Kad ID`	16	Client ID
`Kad ID`	16	File ID
`IP`	4	IP
`Port`	2	TCP port

Tabella 5.2: Pacchetto PacketTCPKadCallback

```
packet = PacketTCPKadCallback(clientId, fileId, IP, TCPPort);
buddy.getContext().addPacketToSend(packet);
```

Una volta che il pacchetto è giunto a destinazione, il peer controlla se effettivamente possiede il file corrispondente al File ID contenuto nel pacchetto inviatogli dal buddy. In caso affermativo si connette via TCP al client iniziale (prendendo IP e porta TCP dal pacchetto `PacketTCPKadCallback()`), che, a sua volta, gli invierà una richiesta di upload.

Capitolo 6

Conclusioni

Con il lavoro di questa tesi si è contribuito allo sviluppo di PariKad soprattutto in termini di coinvolgimento dei client PariMulo all'interno della rete Kad. Infatti, con l'introduzione della callback ed i meccanismi ad essa connessi, anche gli utenti in particolari condizioni di non raggiungibilità sono diventati attivi nella condivisione dei file, portando benefici all'intera rete.

Inoltre, essendo stati i meccanismi descritti in questa tesi affrontati per la prima volta all'interno del progetto PariKad, ci sarà sicuramente la possibilità di approfondire e migliorare la loro attuale implementazione. Infatti, ora che si conoscono i principi su cui si basano, si sa come interagiscono tra di loro e si ha una loro implementazione funzionante, sarà più semplice studiare meglio e raffinare le tecniche e gli algoritmi utilizzati per realizzarli.

In più, con l'introduzione della *kadList*, si sono aperte le porte al possibile sviluppo di nuove funzionalità, quali l'*UDP Firewall Check* e la *Direct Callback*: se si scopre di avere solamente la porta TCP non raggiungibile, si può eseguire la callback senza utilizzare il buddy, ma sfruttando direttamente la porta UDP aperta, facendo transitare le comunicazioni attraverso di essa.

Altra considerazione riguarda, invece, le modalità di lavoro: per velocizzare lo sviluppo di questa parte del progetto sarebbe opportuno studiare dei sistemi più efficienti per i test pratici, che sono stati di gran lunga la parte

più impegnativa dell'attività in termini di tempo e risorse. Questa difficoltà è legata alla necessità di riuscire a gestire e controllare contemporaneamente almeno tre client (PariMulo/eMule) connessi alla rete e ciò non sempre risulta facile, specialmente se ci si trova a dover lavorare singolarmente.

Bibliografia

[1] **Petar Maymounkov and David Mazières**: *Kademlia: A Peer-to-peer Information System Based on the XOR Metric*, 2002.

[2] **René Brunner**: *A Performance Evaluation of the Kad-Protocol*, 2006.

[3] **Francesco Mattia**: *PariMulo: Kad*, 2011.

[4] **Christian Piccolo**: *PariKad: Kad Routing Table*, 2010.

[5] **Andrew S. Tanenbaum and Maarten Van Steen**: *Distributed System: Principles and Paradigms*, 2nd ed, 2007.