Relazione Progetto LPR anno 2008/2009

Andrea Lottarini Alberto Bandettini

26 giugno 2009

Indice

1 Prefazione

L'obbiettivo del progetto finale era realizzare un sistema P2P conforme alle specifiche di BitTorrent per scopi didattici utilizzando un paradigma di programmazione orientato agli oggetti (linguaggio Java). Possiamo affermare di aver raggiunto questo obbiettivo realizzando un sistema funzionale ed efficiente, conforme alle specifiche date e con features/politiche aggiuntive derivate direttamente da quelle del protocollo e dei client BitTorrent attuali. Grazie ad una buona fase iniziale di progettazione delle varie classi la stesura del codice non è risultata eccessivamente complessa, come framework abbiamo utilizzato NetBeans IDE sia per la strutturazione del codice java che dei vari diagrammi UML; dopo una prima stesura del codice effettuata a due mani abbiamo utilizzato Subversion per la gestione delle varie revisioni del codice e la loro condivisione. Abbiamo sfruttato tutte le nostre conoscenze acquisite durante il corso per l'implementazione delle varie parti del progetto avendo cura di sfruttare appieno il paradigma di programmazione ad oggetti e le potenzialità del linguaggio Java. Gran parte dei concetti e delle direttive presenti nella bozza di progetto sono stati rielaborati accuratamente in sede di progetto, la scelta delle politche da utilizzare è derivata in gran parte dalle reference aggiuntive presenti nella bozza mentre per la realizzazione della GUI abbiamo preso a riferimento client BitTorrent attuali (Azureus nel nostro caso).
Altre features aggiuntive sono state implementate per rendere il nostro software utilizzabile in un contesto reale, in particolare abbiamo posto grande attenzione nel realizzare un interfaccia grafica il più possibile reattiva e comprensibile, abbiamo aggiunto meccanismi per il salvataggio su file (.creek) dei riferimenti allo swarm, meccanismi di terminazione gentile sia del client che del server con salvataggio dello stato, aggiramento di NAT e firewall, risolto problematiche relative al trasferimento di file di grosse dimensioni. Grazie a queste features aggiuntive ci sentiamo di affermare che il nostro programma è utilizzabile anche in un contesto reale.
Tutti i vari aspetti qui trattati verrano adeguatamente spiegati nel resto della relazione, sia come analisi dei singoli package sia come spiegazione delle varie features e politiche implementate.

2 Test del Progetto

3 Suddivisione in Package

La fase di progettazione ha portato ad individuare numerose classi. E‘ risultata fondamentale una corretta suddivisione in package, sia nel lato server che nel lato client dell'applicazione. Ogni package verrà analizzato singolarmente nelle varie sottosezioni.

BitCreekPeer Nel realizzare il peer abbiamo suddiviso il codice in tre package

- gui: contiene tutto il codice necessario per il funzionamento dell'interfaccia grafica. Vi sono degli ulteriori riferimenti alla cartella icone contenente tutte le icone utilizzate dalla GUI.
- peer: contiene tutto il codice relativo alle politiche e ai thread del peer
- condivisi: si tratta di un package condiviso tra peer e server

BitCreekServer Nel realizzare il peer abbiamo suddiviso il codice in due package

- server: contiene tutto il codice necessario per il funzionamento del server
- condivisi: vedi sopra

Condivisi E‘ un package condiviso tra peer e server, principalmente contiene tutte le classi necessarie per il funzionamento dell'RMI, gestione eccezioni e dati scambiati tra peer e server.

3.1 server

Il package *server* contiene le seguenti classi:

- BitCreekServer: Classe principale contenente il main del server
- ImplementazioneRMI: implementazione dei metodi presenti nell'InterfacciaRMI condivisa tra peer e server.
- ListaPeer: ArrayList di NetRecord gestita dai Tracker del server (relativa a singolo Swarm).
- MetaInfo: HashSet di Descrittori
- NumPeer: Classe che tiene traccia di Seeder e Leecher (relativa a singolo Swarm).
- ServerListener: Thread di gestione del meccanismo di aggiramento del NAT.
- ThreadSaver: Thread di gestione del salvataggio su file della lista degli swarm presenti.

- TrackerTCP: Thread di gestione della ricerca Peer.
- TrackerUDP: Thread di gestione del meccanismo di Keep Alive.
- Trimmer: Classi che implementa le funzionalità di rimozione Peer che non inviano messaggi di KeepAlive.

e sfrutta le classi presenti in *condivisi*:

- Descrittore: Classe che racchiude tutte le informazioni relative allo swarm.
- ErrorException: Classe che estende le normali eccezioni in Java, utilizzata per un trattamento uniforme di tutte le eccezioni che vanno notificate all'utente.
- InterfacciaRMI: Interfaccia condivisa con il peer per il meccanismo RMI
- InterfacciaCallback: Interfaccia condivisa con il peer per il meccanismo delle CallBack
- NetRecord: Classe che definisce le informazioni necessarie alla lista Peer
- Porte: Classe wrapper utilizzata per definire le informazioni che un peer deve ricevere a seguito dell'invio di un descrittore appena creato; contiene le porte del trackerUDP e TCP associati allo swarm appena creato.

3.2 peer

Il package *gui* contiene le seguenti classi:

- BitCreekGui: Classe principale del package, contenente il main dell'applicazione lato Peer.
- FunctionPanel: Classe che implementa il grafico delle connessioni.
- ModelloTabellaCerca
- ModelloTabellaMieiCreek
- ModelloTabellaPubblicati
- RigaTabellaCerca
- RigaTabellaMieiCreek
- RigaTabellaPubblicati

Il package *peer* contiene le seguenti classi:

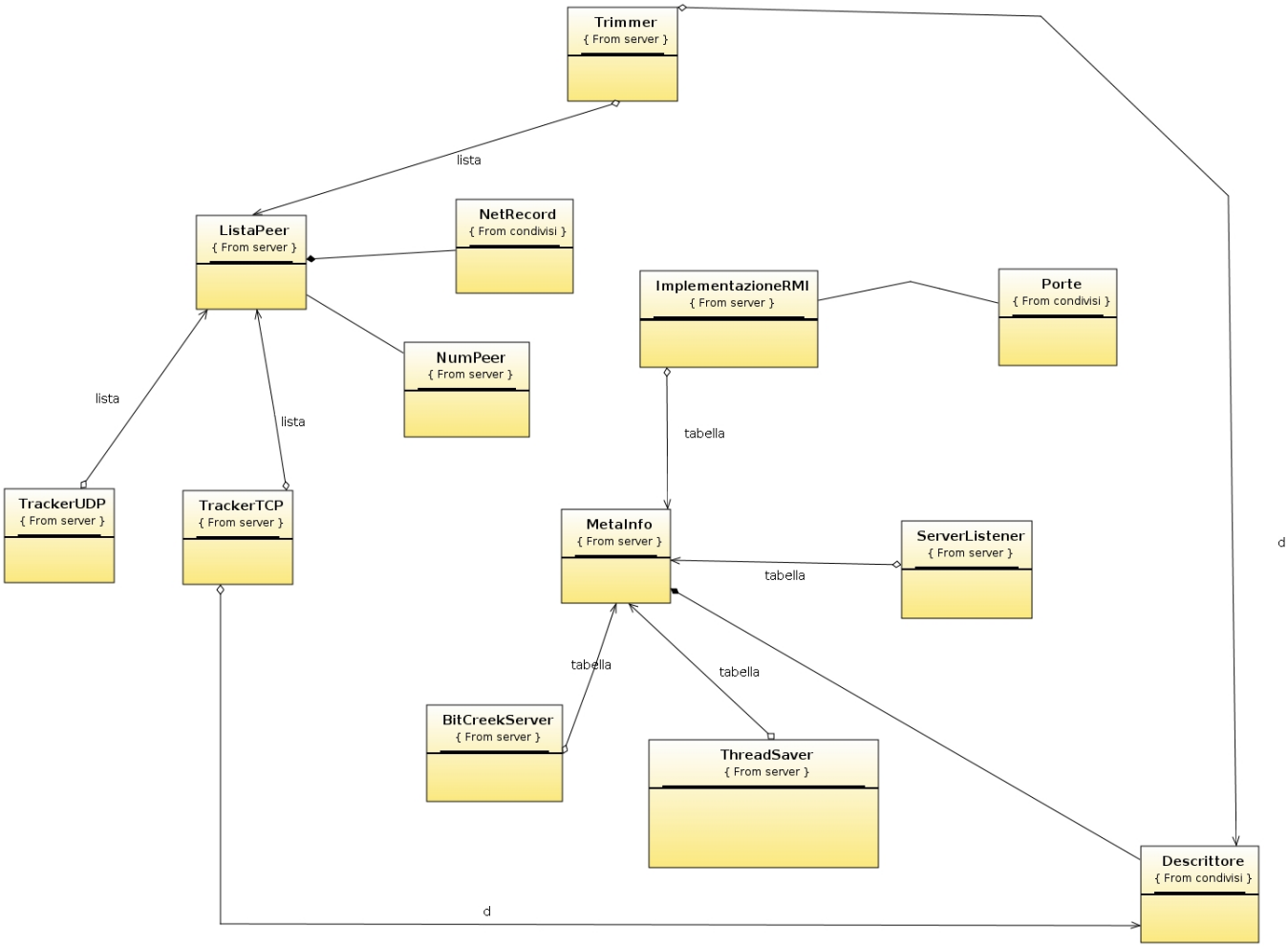

Figura 1: Diagramma delle classi del Server.

- BitCreekPeer: Classe principale del package, viene creato da BitCreekGui al momento dell'avvio dell'applicazione.
- Apri: Thread di apertura di un file .creek
- Ascolto: Thread di ascolto di nuove connessioni
- Avvia: Thread di Avvio di un nuovo swarm
- Bitfield: Classe che definisce un messaggio di risposta a livello di HandShake applicativo.
- Cerca: Thread che effettua una ricerca sul server.
- Chunk: Classe che definisce tutti gli attributi di un chunk del file relativo allo Swarm.

- Connessione: Classe che virtualizza la connessione tra 2 peer
- Contact: Classe che definisce il primo Messaggio a livello di Hanshake Applicativo.
- Crea: Thread che implementa la creazione di un nuovo Descrittore da submittare tramite RMI al server.
- Creek: Classe che definisce gli attributi necessari alla gestione di uno swarm (a runtime).
- Downloader: Thread che gestisce lo scaricamento di un file su di una connessione.
- Elimina: Thread che gestisce l'eliminazione di un creek con relativa comunicazione al server.
- Implementazione Callback: gestione della callback relativa alla ricerca.
- KeepAlive: Thread che gestisce l'invio dei messaggi di KeepAlive.
- Messaggio: Classe che definisce i messaggi scambiati tra per lo scaricamento di un file.
- PIO: Classe che definisce gli attributi di un Chunk da scaricare.
- Riavvia: Thread invocato al per la gestione di Swarm interrotti.
- Uploader: Thread che gestisce l'upload su una connessione.
- UploadManager: Thread relativo ad ogni swarm per la gestione delle politiche di upload.

Il peer utilizza le classi contenute nel package condivisi, gia illustrato nella sezione precedente.

diagramma UML delle classi:

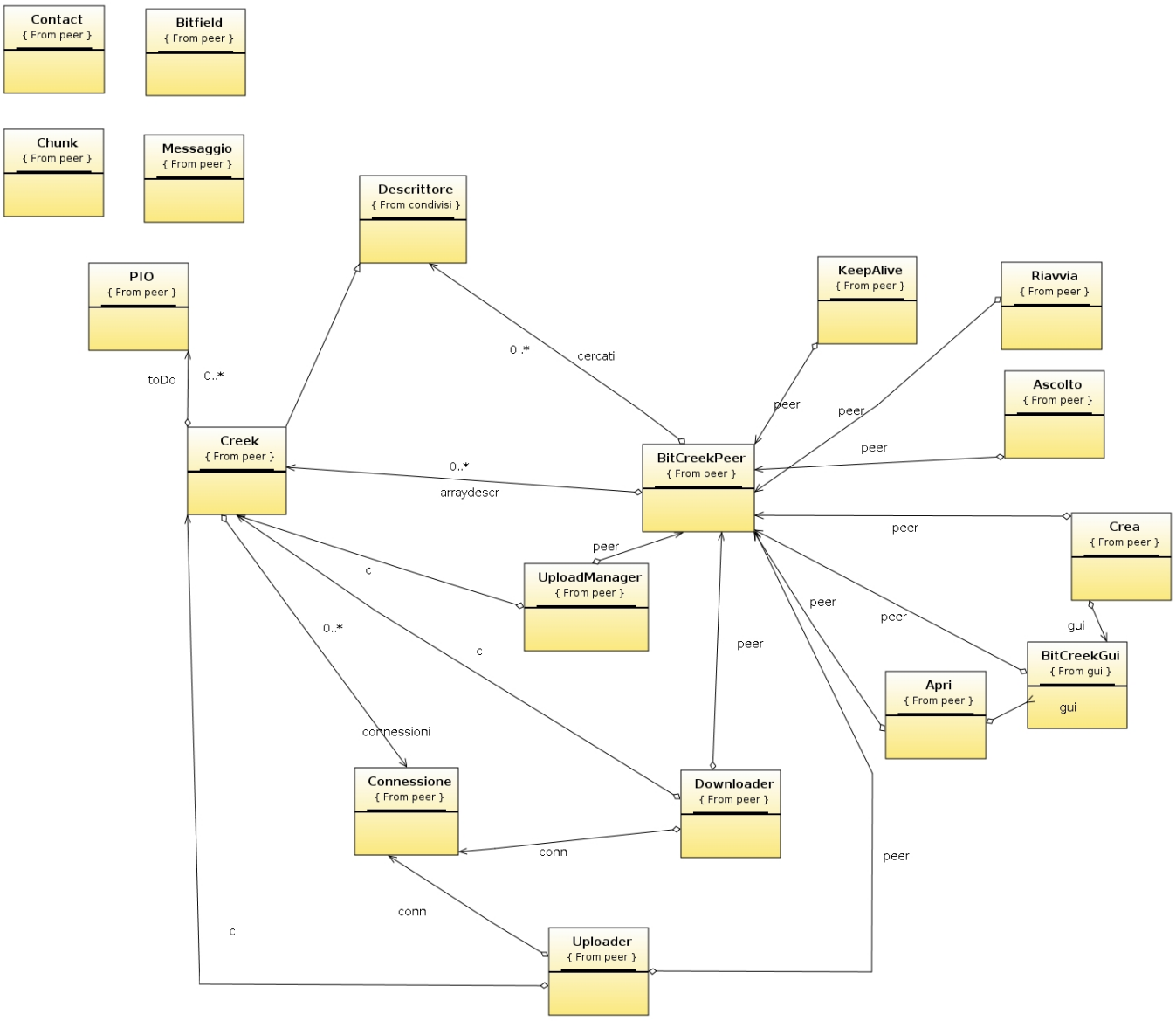

Figura 2: Diagramma delle classi del Peer.

4 Implementazione e Funzionamento

4.1 Server

Analizziamo come prima cosa il funzionamento dell'intero server andando poi a focalizzare l'attenzione sugli aspetti implementativi più importanti. Il main della parte server dell'applicazione si trova nella classe BitCreekServer all'interno del package *server*, all'avvio viene istanziato un nuovo oggetto BitCreekServer e contestualmente alla sua creazione (quindi nel costruttore) viene ripristinato uno stato consistente a partire da un file di configurazione. Successivamente vengono avviati i thread ServerListener e ThreadSaver ed infine avviato RMI. Si noti che le porte utilizzate come ServerSocket per la connessione dei peer e come porta RMI sono fissate a programma questo perché abbiamo ritenuto che un server abbia piena disponibilità delle proprie porte, diversamente nel lato client abbiamo creato un complesso sistema di gestione del NAT interattivo e a runtime per la scelta delle porte. Nel file di configurazione viene salvato lo stato di ogni swarm di cui il server tiene traccia, al momento della creazione del server vengono ripristinati (per ogni swarm) tre thread, un thread tracker di keepalive su UDP, un thread tracker per gestire i nuovi peer che si aggiungono allo swarm e infine un thread Trimmer. Il thread ServerListener si occupa di gestire le richieste di connessione al server da parte dei peer e fornisce meccanismi per la gestione del NAT. Il thread ThreadSaver invece effettua un periodico salvataggio dello stato del server su file di configurazione, questa costituisce una feature aggiuntiva che abbiamo deciso di implementare nel nostro progetto che si rileva particolarmente utile nel nostro caso dato che non abbiamo replicazione del server e quindi un improvviso guasto causerebbe la perdita di tutti i dati relativi agli swarm. Dal diagramma delle classi 1 si nota come La classe Descrittore sia il punto di centralizzazione di tutto il server. Questa classe definisce tutti gli attributi di uno swarm che è bene esplicitare:

Listing 1: *Descrittore*.

```
public class Descrittore implements Serializable {

    /* Variabili d'istanza */
    /** Identificatore dello swarm */
    private int id;
    /* campi riguardanti il file descritto */
    /** Nome del file */
    private String nomefile;
    /** Dimensione del file */
    private long dimensione;
    /** Stringa hash del file */
    private byte[] hash;
    /* campi utili a chi usa il descrittore */
    /** Porta del tracker TCP */
    private int portaTCP;
```

```
    /** Porta del tracker UDP */
    private int portaUDP;
    /** Interfaccia per la callback */
    private InterfacciaCallback stubcb;
    /** Numero di seeder del file */
    private int numSeeders;
    /** Numero di leecher del file */
    private int numLeechers;

...
```

Di fondamentale importanza risultano i campi:

- int id: E' l'identificativo unico dello swarm, viene creato al momento della creazione, tramite RMI, del descrittore realtivo.
- byte[] hash: E' la codifica in SHA del file relativo allo swarm, viene utilizzato al momento della creazione per verificare che non vi siano 2 file uguali associati a diversi swarm.
- int portaTCP, portaUDP: sono le porte di ascolto dei tracker inizializzate al momento della creazione del descrittore. Al momento del riavvio del server i tracker verranno riavviati sulle porte gia utilizzate di modo che un peer può sempre riunirsi allo swarm.
- int numSeeders, numLeechers: sono statistiche sul numero di seeder e leecher che partecipano allo swarm, vengono tenute costantemente aggiornate dal thread Trimmer.

4.1.1 interazioni RMI

La struttura dati viene inizializzata tramite chiamate RMI da parte dei peer che vogliono condividere un nuovo file.

Listing 2: *inviaDescr*.

```
/**
     * Metodo esportato dal server al peer per l'invio di un
       descrittore
     * @param d descrittore da pubblicare
     * @param ip ip client che pubblica
     * @param porta porta in ascolto del peer che pubblica
     * @return porte TCP e UDP dei tracker
     * @throws java.rmi.RemoteException
     */
    public Porte inviaDescr(Descrittore d, InetAddress ip, int porta
       ) throws RemoteException;
```

Come prima cosa viene effettuato un controllo, tramite lo SHA, che il file inviato non sia gia associato ad un altro swarm, in tal caso il peer viene solamente aggiunto e non riceve la callback per le ricerche (si noti come l'implementazione di questo meccanismo è del tutto trasparente al peer). Questo meccanismo risulta di fondamentale importanza in quanto permette di sfruttare appieno la rete P2P. Nel caso questo controllo viene superato il nuovo descrittore viene aggiunto alle *metainfo* (implementate tramite HashSet), vengono inizializzati i tracker TCP e UDP relativi, il thread Trimmer e le relative strutture dati.

L'altra possibile interazione tramite RMI è:

Listing 3: *ricerca*.

```
/**
     * Metodo esportato dal server al peer per effettuare una
       ricerca
     * @param nomefile nome del file da cercare
     * @param ind ip del peer che sta cercando
     * @return lista di descrittori
     * @throws java.rmi.RemoteException
     */
    public ArrayList<Descrittore> ricerca(String nomefile,
       InetAddress ind) throws RemoteException;
```

Questa chiamata provoca una scansione delle MetaInfo del server e ritorna una lista di descrittori. Le funzionalità di confronto lessicali sono implementate utilizzando la librerie per le espressioni regolari consigliata.

4.1.2 ListaPeer

Rimane da chiarire le funzionalità di questa struttura condivisa che rappresenta un notevole punto di contralizzazione per il server. Per ogni swarm di cui il server tiene traccia viene istanziata una *ListaPeer* contenente i riferimento ad ogni peer che partecipa allo swarm. Questa lista viene gestita concorrentemente da 3 Thread:

- Thread TrackerTCP: Questo thread si occupa di ricevere su connessioni SSL sicure le richieste di unione allo swarm da parte dei peer. Al momento della ricezione di una richiesta il Thread aggiunge una nuove entry alla ListaPeer e invia al peer una lista di NetRecord, classe condivisa che contiene tutte le credenziali del peer.
- Thread TrackerUDP: Questo thread si occupa di ricevere i messaggi di keep alive dai peer su connessioni UDP, al momento della ricezione di un messaggio di keep alive viene scandita la ListaPeer e invocato il metodo touch() sul NetRecord corrispondente.
- Thread Trimmer: Questo thread, di tipo TimerTask, viene eseguito a intervalli regolari e si occupa di eliminare dalla ListaPeer i NetRecord relatvi a peer che non hanno più inviato messaggi di keep Alive.

Tutti i metodi per l'accesso e la modifica alla ListaPeer sono quindi Synchronized per garantirne la correttezza ed evitare Race Condition.

4.1.3 Features aggiuntive e chiarimenti

Nel realizzare il server abbiamo fatto alcune assunzioni ulteriori rispetto a quelle presenti nella bozza di progetto:

- Descrittori con zeri fonti: Differentemente da quanto specificato nella bozza di progetto i file con fonti non al momento disponibili non vengono rimosse dal server, la stato dei loro descrittori viene ugualmente salvato sul server ma la ricerca di un file con zero fonti non restituisce risultati, con questo meccanismo è possibile che lo swarm venga riattivato al momento che si ripresentano alcuni dei seeder o un peer pubblichi nuovamente il file relativo allo swarm.
- Salvataggio periodico dello stato: Come precedentemente spiegato questa funzionalità è stata aggiunta come prevenzione contro fallimenti improvvisi del server che, in mancanza di un meccanismo di salvataggio periodico, risulterebbero fatali per il buon funzionamento dell'applicazione.
- Controllo sulla pubblicazione: Per evitare che un file venga pubblicato più volte abbiamo effettuato un controllo sia sullo SHA del file che sulla dimensione, in questo modo la probabilità che due file diversi risultino uguali è assolutamente trascurabile.

4.2 GUI

Il main della parte server in questo caso si trova nella classe *BitCreekGui*. Al momento della creazione di una nuova istanza di BitCreekGui vengono eseguite in sequneza le seguenti azioni:

1. initComponents(): Funzione che inizializza tutta la parte grafica del peer.
2. initProtocol(): Funzione che inizializza la parte di protocollo (o logica) del peer, al suo interno viene istanziato un nuovo BitCreekPeer che mantiene un riferimento alla BitCreekGui per l'aggiornamento grafico in seguito ad eventi rilevati dalla logica.
3. inizializzazione del ListenerTabelle.
4. inizializzazione del ListenerGrafico.

Al momento del'avvio la GUI si presenta come in figura:
In questa sezione daremo una spiegazione delle varie politiche e delle scelte implementative fatte, per una descrizione dell'interazione utente-GUI si legga la sezione

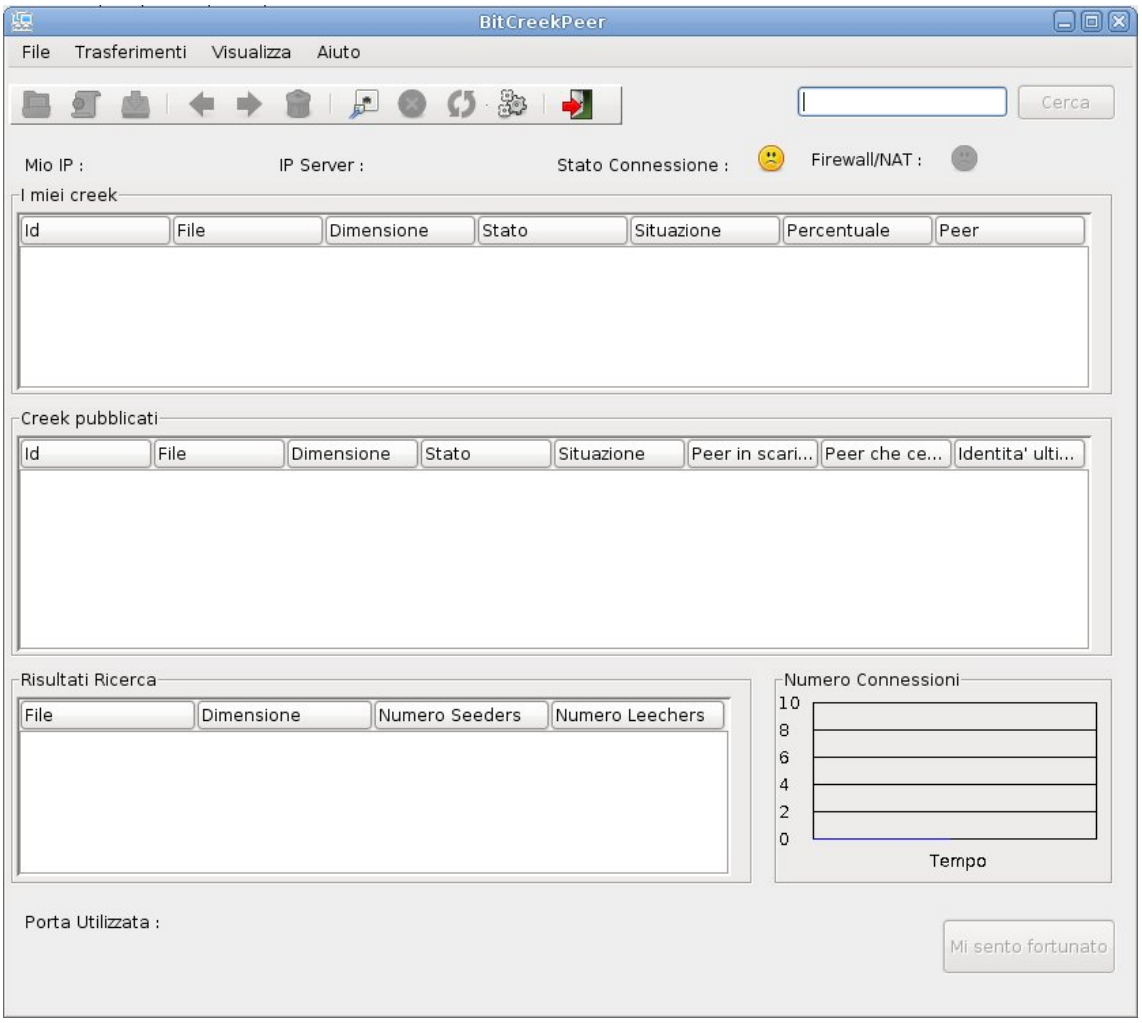

Figura 3: snapshot della GUI.

successiva. Nella realizzazione della Gui abbiamo posto grande attenzione all'utilizzo delle primitive adeguate per avere un'interfaccia grafica responsiva. In particolare abbiamo utilizzato i meccanismi della libreria *Javax.Swing* per la gestione dell'EventQueue, in particolare nessuna delle operazioni invocabili dalla Gui gira sul Thread Dispatcher ma viene assegnato ad un Thread apposito della logica che al termine dell'elaborazione invoca un apposito metodo della Gui per l'aggiornamento della grafica con il metodo *InvokeLater*(). Con questi meccanismi manteniamo l'interfaccia grafica sempre responsiva anche a seguito di elaborazioni consistenti da parte della logica. In particolare per le funzionalità come la creazione dei Descrittori e la loro ricerca tramite RMI, molto onerose anche a causa dell'interazione client-server, l'interfaccia grafica non presenta freeze. Unica eccezione a questa politica è la funzione di connessione, abbiamo infatti ritenuto che nel caso della connessione fosse più naturale avere un freeze dell'interfaccia fino a che la procedura di connessione al server non è completata. Altro aspetto

che abbiamo considerato accuratamente è l'abilitazione/disabilitazione dei vari bottoni, evitando quindi che un utente inesperto possa effettuare operazioni prive di senso ed eventualmente dannose per la consistenza dei dati.

4.2.1 Implementazione della GUI

La GUI utilizza tre modelli con altrettanti pannelli:

- modello MieiCreek: visualizza lo stato degli swarm a cui sto partecipando.
- modello CreekPubblicati: visualizza lo stato degli swarm in cui sono seeder, il peer che ha pubblicato il file ha anche visibilità dei peer che effettuano la ricerca.
- modello RisultatiRicerca: vengono visualizzati i risultati della ricerca di un file, la ricerca si attiva sia dalla textbox in alto a sinistra che tramite il pulsante Mi sento fortunato in basso a destra. Questa funzionalità è stata implementata durante il testing dell'applicazione ed è rimasta inalterata nella versione finale del progetto.

Listener della GUI La GUI utilizza due listener, entrambi implementati come SwingTimer. Questi due Thread effettuano una scansione di arrayDescr¡Creek¿ e arrayCercati¡Descrittore¿ nella logica del peer e aggiornano le relative tabelle dell'interfaccia grafica. In fase di progetto abbiamo deciso di utilizzare i meccanismi degli SwingTimer perché consentono una stesura del codice coerente oltre ad essere preferibile, nel nostro caso, a degli oggetti *Observable*. Degli oggetti *Observable* avrebbero causato un Overhead eccessivo per il peer e assolutamente ingiustificato. Nel nostro caso abbiamo scelto un delay tra le varie invocazioni del timer di 0.5 secondi nell'aggiornamento delle tabelle e di 1.2 secondi per l'aggiornamento del grafico delle connessioni, la scelta dei valori è un buon compromesso tra la responsività dell'interfaccia grafica e l'overhead introdotto dai Thread.

4.3 Peer

Questo Package contiene le varie classi che compongono la logica dell'applicazione e definiscono il protocollo di interazione tra peer. Al momento dell'avvio (initProtocol) da parte della GUI viene istanziato un oggetto di tipo BitCreekPeer, il suo stato viene ripristinato dalla precendente esecuzione usando dei file di configurazione presenti in una cartella apposita. Come prima cosa l'utente deve connettersi da interfaccia grafica tramite l'apposito tasto, durante la connessione viene invocata la procedura di test del NAT, se la procedura da esito positivo il peer risulta connesso al server. A questo punto vengono avviati i vari thread di supporto:

- Thread KeepAlive
- Thread Listener

A questo punto le varie funzionalità della GUI sono attive ed è possibile:

1. creare un nuovo descrittore.
2. creare un file .creek.
3. aprire un file .creek e avviarlo.
4. avviare un file tra quelli cercati.
5. eliminare file (sia in stato di seeder che di leecher).
6. disconnettersi dalla rete.
7. chiudere l'applicazione (ovviamente).

Si osservi il diagramma dei casi d'uso.

4.3.1 arrayDescr

Questa classe rappresenta il punto centrale di tutta l'applicazione. E' implementata come ArrayList⟨*Creek*⟩ e contiene tutte le informazioni relative ai vari swarm a cui il peer partecipa, ogni singolo oggetto di tipo Creek infatti rappresenta un Descrittore a runtime e quindi racchiude tutte le informazioni necessarie a runtime per la gestione di uno swarm. Data la sua importanza riportiamo qui le sue varibili di istanza:

Listing 4: *Creek*.

```
/**
 * Classe che definisce la struttura dati
 * del client di supporto al download/upload
 * in uno swarm
 * @author Bandettini Alberto
 * @author Lottarini Andrea
 * @version BitCreekPeer 1.0
 */
public class Creek extends Descrittore implements Serializable {

    /** Contatatore */
    private static int countNext = 0;

    /* Variabili d'istanza */
    /** Stato del creek : puo essere LEECHER o SEEDER */
    private boolean stato;
    /** Situazione del creek : puo' essere NOTSTARTED o STARTED */
    private boolean situazione;
    /** Politica di download : puo' essere INIT, RAREST o ENDGAME */
    private int statoDownload;
```

```
    /** Percentuale di completezza del file */
    private int percentuale;
    /** Flag che indica se il file e' stato pubblicato dal peer */
    private boolean pubblicato;
    /** Numero di peer connessi al peer per questo file */
    private int peer;
    /** Numero di peer che hanno crecato questo file */
    private int peercercano;
    /** IdentitÃ  dell' ultimo peer che ha cercato questo file */
    private InetAddress ind;
    /** Array di falg che indicano i pezzi posseduti dal peer */
    private boolean[] have;
    /** Array di PIO che indicano i chunk liberi o occupati */
    private ArrayList<PIO> toDo;
    /** Array di connessioni per questo file */
    private ArrayList<Connessione> connessioni;
    /** File del creek */
    private File file;
    /** File Random associato al creek */
    private RandomAccessFile raf;
    /** Numero pezzi scaricati */
    private int scaricati;
    /** Array degli indici dei pezzi scaricati */
    private int[] scaricatiId;
```

4.3.2 Analisi delle varie classi

Andiamo ad analizzare le varie classi in dettaglio:

Le seguenti classi implementano tramite appositi Thread le funzionalità invocate dalla GUI:

- Apri
- Avvia
- Cerca
- Crea
- Riavvia

La caratteristica comune di questi Thread è di operare principalmente su arraydescr del peer per aggiungere/avviare/eliminare nuovi creek su cui lavorare. Merita approfondire le operazioni svolte dal thread Avvia in quanto sono significative per comprendere l'avvio di uno scaricamento da parte del peer:

1. Creazione dell'oggetto creek a partire dal descrittore ricevuto dal server.
2. Controllo di presenza (si verifica che lo swarm non sia gia stato avviato)
3. Settaggio del Creek e aggiunta ad arrayDescr, in questa fase vengono inzializzate tutte le parti del creek NON serializzabili e relative al download, in particolare viene aperto il RandomAccessFile e inizializzata la lista toDo relativa ai chunk da scaricare.
4. Contatto il server in SSL e recupero la listaPeer.
5. Per ogni NetRecord contenuto nella ListaPeer tento una connessione (Handshake applicativo), se ha successo creo un nuovo oggetto Connessione e lo associo ad un Thread Downloader
6. Avvio l'UploadManager

In fase di progetto abbiamo scartato l'ipotesi di introdurre un'ulteriore livello di generalizzazione estendendo la classe Creek in quanto non avrebbe introdotto un ulteriore livello di astrazione ma esclusivamente aggiunto dettagli implementativi alla classe.

Questo stesso tipo di interazione è riscontrabili in tutti gli altri Thread sopra elencati con lievi differenze, per esempio nel caso di Riavvia i descrittori non vengono selezionati a partire dalla funzionalità di ricerca ma caricati da file system.

4.3.3 Politiche e Strutture Dati

A questo punto è doveroso dare una panoramica delle politiche e stutture dati utilizzate per la gestione di uno swarm all'interno del peer. La trattazione non può essere ovviamente esaustiva ma cercheremo di evidenziare i punti salienti della progettazione. Facciamo notare che ogni swarm viene gestito indipendentemente, l'unica eccezione è data dalla gestione del limite connessioni, globale per ogni peer. Come prima cosa dobbiamo evidenziare che per la realizzazione delle politiche ci siamo attenuti strettamente alle politiche reali di BitTorrent facendo riferimento, quando possibile, al testo [Michelle] ; qualora il testo non fosse risultato di aiuto abbiamo progettato autonomamente le politiche da utilizzare. In particolare la definizione dei vari messaggi scambiati tra i peer è conforme a quella del testo, per la scelta degli algoritmi ci siamo invece attenuti il più possibile alle politiche di BitTorrent introducendo eventuali modifiche per aumentare le prestazioni. Come prima cosa è utile dare visione delle variabili di istanza dell'oggetto connessione:

Listing 5: *Connessione*.

```
/**
 * Classe che virtualizza la connessione tra 2 peer
 * facenti parte di uno swarm
 * @author Bandettini Alberto
```

```
 * @author Lottarini Andrea
 * @version BitCreekPeer 1.0
 */
public class Connessione implements Serializable, Comparable<
    Connessione> {

    /* Costanti */
 ...
    /* Variabili d' istanza*/
    /** Socket in download */
    private Socket down;
    /** Socket in upload */
    private Socket up;
    /** Porta di ascolto del partner */
    private int portaVicino;
    /** IP del partner */
    private InetAddress ipVicino;
    /** Flusso in ingresso sulla socket in download */
    private ObjectInputStream inDown;
    /** Flusso in uscita sulla socket in download */
    private ObjectOutputStream outDown;
    /** Flusso in ingresso sulla socket in upload */
    private ObjectInputStream inUp;
    /** Flusso in uscita sulla socket in upload */
    private ObjectOutputStream outUp;
    /** Stato della connessione in download */
    private boolean statoDown;
    /** Stato della connessione in upload */
    private boolean statoUp;
    /** Interesse della connessione in download */
    private boolean interesseDown;
    /** Interesse della connessione in upload */
    private boolean interesseUp;
    /** Bitfield del partner */
    private boolean[] bitfield;
    /** Flag che indica se Ã¨ possibile fare upload su questa
        connessione */
    private boolean uploadable;
    /** Numero chunk scaricati su questa connessione */
    private int downloaded;
    /** Flag che indica di terminare la connessione */
    private boolean termina;
```

Si nota che l'oggetto connessione wrappa 2 socket, una dedicata al download e una all'upload. Così facendo abbiamo numerevoli vantaggi. In primo luogo un'astrazio-

ne sulle socket vere e proprie che facilita notevolmente la comprensibilità del codice operante sulla connessione. In secondo luogo i Thread Downloader e Uploader che operano sulle connessioni risultano completamente autonomi, l'utilizzo di due socket separate permette ai due thread di operare in maniera totalmente indipendente e con un protocollo a scambio di messaggi semplice e intuitivo. E' risultato a questo punto molto naturale estendere il comportamento del Thread Downloader con le politiche INIT,RAREST e ENDGAME direttamente ricavate dal protocollo BitTorrent. Diamo una rapido riassunto di queste politiche:

- INIT: al momento dell'avvio di uno swarm il Chunk da scaricare va scelto in maniera totalmente casuale in maniera da massimizzare la probabilità di ottenere un pezzo utile al più presto, una volta raggiunto questo obbiettivo si passa in RAREST.

- RAREST: la scelta dei pezzi avviene in base alla rarità, ad ogni passo il pezzo prescelto è quello più raro, si noti che come politica aggiuntiva abbiamo introdotto una scelta casuale tra pezzi di pari rarità, in un ambiente di test con numero di peer ristretto come quello che avevamo a disposizione (o comunque su swarm poco popolati) l'aumento di prestazioni e distribuzione complessiva del file tra i peer è risultata assai maggiore.

- ENDGAME: al raggiungimento di una soglia specificata ogni singolo Thread effettua una richiesta unica di tutti i pezzi mancanti all'altro peer, l'overhead introdotto da questa tecnica è giustificato dall'aumento delle probabilità di completamento di un file con il conseguente rilascio di tutte le risorse occupate dai Thread Downloader (di cui beneficiano anche gli altri peer dato che si aggiunge automaticamente un nuovo seeder).

L'utilizzo degli oggetti connessione ha inoltre facilitato lo sviluppo del Thread UploadManager, responsabile delle politiche di CHOKE e UNCHOKE; nel realizzarle abbiamo introdotto gli stessi meccanismi utilizzati da BitTorrent con 4 peer UNCHOKE e un peer scelto a caso (OPTIMISTIC UNCHOKE). Nel realizzare le varie politiche abbiamo adottato un approccio coeso realizzando sia la classe Connessione che la classe PIO come implementazione di Comparable, questo ha permesso l'utilizzo dei meccanismi forniti dal package Collections per l'ordinamento:

- nel caso della politica Rarest utilizzato per ordinare i pezzi da scaricare in base alla rarità.

- nel caso dell'UploadManager per ordinare le connessioni in base al numero di pezzi scaricati.

Si noti a questo punto la bonta della scelta di un oggetto che virtualizzi la connessione: I thread Downloader e Uploader utilizzano metodi e variabili della classe connessione distinti senza introdurre overhead tra di loro dovuto ad eventuali sincronizzazioni.

Il thread UplodManager opera interamente sull'oggetto connessione e in basse alle informazioni salvate dal thread Downloader comunica al Thread Uploader il CHOKE o l'UNCHOKE sul pezzo, il tutto però in maniera molto naturale e comprensibile a livello di codice. Se si osserva le operazioni svolte dal Thread UploadManager si nota che esso effettua esclusivamente l'ordinamento della lista Connessioni e le segnalazioni sui Monitor associati.

4.3.4 Meccanismi di sincronizzazione

La scelta dei corretti meccanismi di sincronizzazione è stata ampiamente ponderata. Per la sincronizzazione tra Thread si è fatto uso di metodi synchronized. Come gia specificato precedentemente l'UploadManager fa uso di monitor per gestire gli upload, in questa maniera abbiamo il duplice vantaggio di una strutturazione del codice elegante e la riduzione dell'overhead dovuta ai Thread Uploader in quanto esclusivamente i thread che posso uploadare sono attivi. Infine bisogna notare che sono presenti numerosi thread all'interno di BitCreek che operano a intervalli regolari di tempo, per ogni caso è stato scelto un meccanismo adeguato allo scopo valutando l'overhead introdotto e i benefici, in particolare.

- I thread relativi alla GUI, avendo minore priorità rispetto ai thread della logica, utilizzano SwingTimer per andare in esecuzione a intervalli regolari. I motivi di questa scelta sono essenzialmente due: in questa maniera abbiamo una gestione coerente esattamente come se fossere eventi, possiamo utilizzare meccanismi quali InvokeLater per evitare che l'esecuzione di questi thread rallenti l'applicazione. [Sun]

- I thread quali: sono gestiti come TimerTask in quanto abbiamo ritenuto che un loro eventuale delay nell'esecuzione non costituisse un problema e fosse preferibile un minore overhead.

- I thread a maggiore priorità infine sono gestiti tramite sleep esplicite in quanto la loro esecuzione a intervalli regolari è fondamentale per il corretto funzionamento dell'applicazione o sono necessarri meccanismi di sincronizzazione esplicita come gli interrupt.

4.3.5 Features Aggiuntive: Salva, Terminazione

Queste due features sono state aggiunte

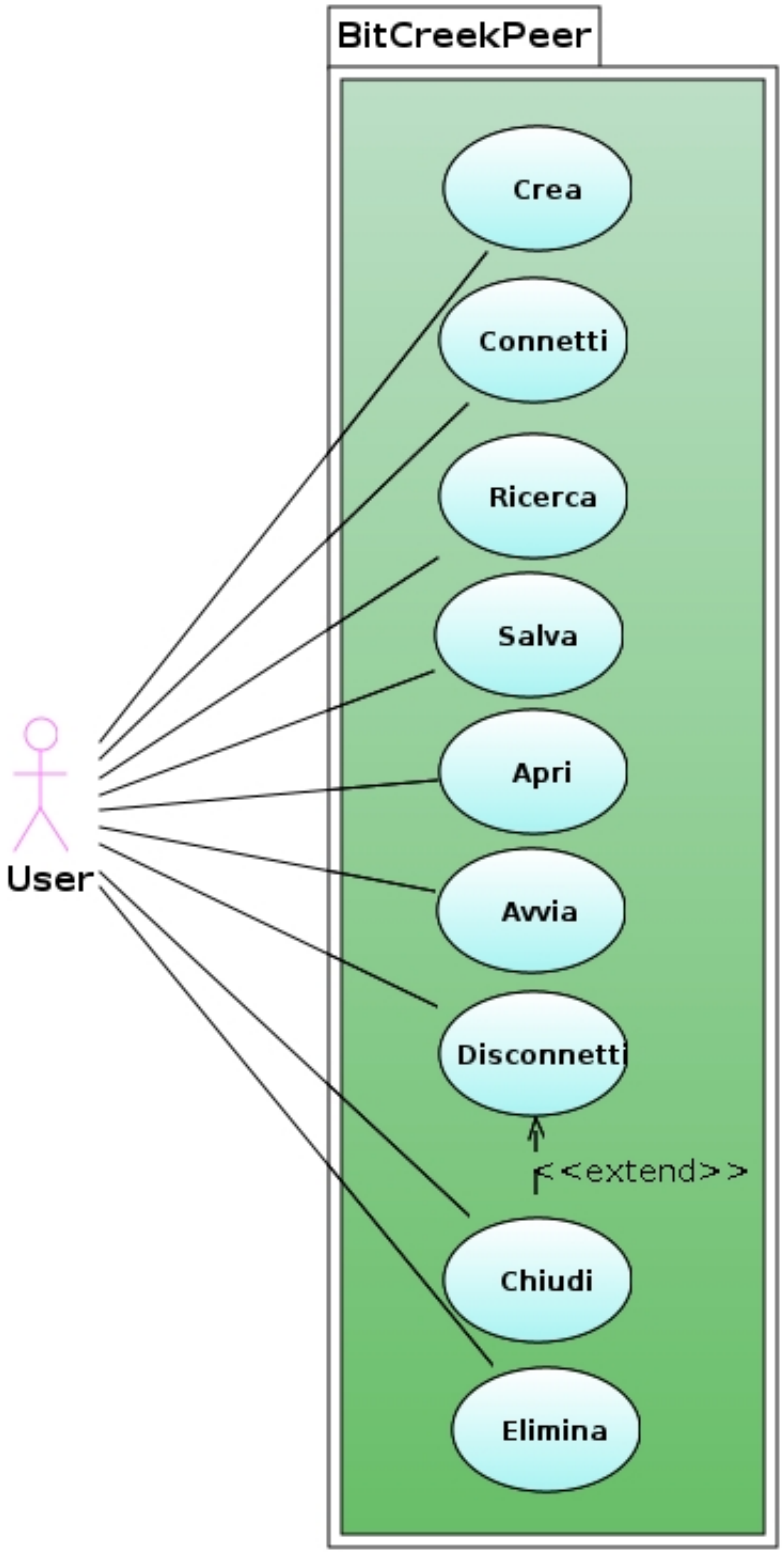

Figura 4: Casi d'uso.

5 Interazione con la GUI

Al momento dell'avvio dell'applicazione viene ripristinato lo stato precedente all'ultima terminazione. Tutti gli swarm a cui il peer stava partecipando sono visibili nelle apposite tabelle.

Riferimenti bibliografici

Teo Wei Ling Michelle. *A BitTorrent Implementation and Simulation.* PhD thesis.

Sun. Concurrency in swing. In *Java Tutorials.*