Relazione del progetto di L.P.R. di fine corso a.a. 08/09.

BitCreek: Una Rete P2P per la distribuzione dei contenuti.

Autore Giulio Tranchida, n° matricola 241732.

I package

I package del programma sono tre: *bitcreek*, *bitcreek.client* e *bitcreek.server* e rispecchiano l'organizzazione logica del codice del programma. Il package *bitcreek* contiene le classi che devono mantenere una visibilità globale all'interno del progetto, il package *bitcreek.client* include tutte le classi che definiscono il client del programma, analogamente il package *bitcreek.server* definisce tutte le classi che implementano il server.

Le classi del package bitcreek

Classe *Logger*

La classe `Logger` scrive su file e stampa a monitor tutte le azioni rilevanti eseguite sia dal client che dal server. Di questa classe i metodi più usati sono: `out` ed `err`, entrambi scrivono sul file la stringa o l'oggetto passato come parametro, e precisamente il metodo `out` stampa il messaggio sullo *stdout*, mentre il metodo `err` stampa sullo *stderr*.

Inoltre, la classe implementa un buffer di dimensione variabile supportato dall'oggetto `LinkedList<StringBuilder> pending` che una volta raggiunta la sua capienza massima, definita come attributo statico all'interno della classe stessa, viene svuotato del suo contenuto effettuando la scrittura su file di tutti gli eventi. L'idea base, di questa scelta implementativa opzionale, è quella di ottimizzare il numero di accessi al disco.

La classe Logger implementa anche due costruttori. Il primo ha come parametro formale il solo nome del file che verrà usato per memorizzare su disco gli eventi del programma. Il secondo costruttore permette di definire con precisione: il nome del file, la cartella dove verrà memorizzato, la sua dimensione massima e permette, infine, di escludere l'uso del buffer imponendo la scrittura immediata di ogni evento su disco.

Se non diversamente specificato, la cartella dove il file verrà memorizzato è *log*, questa si troverà nella cartella radice del programma. Ogni evento memorizzato su file è comprensivo di data ed orario, e qualora dovesse esistere già un file con lo stesso nome, ogni nuovo evento verrebbe appeso in coda al file.

Se il file supera la dimensione massima definita verrà creato un nuovo file (*nome_del_file_2.log*), se quest'ultimo, a sua volta, raggiunge la sua dimensione massima verrà sovrascritto il file precedente (*nome_del_file_1.log*).

Nel caso che entrambi i file esistano ed entrambi siano saturi, l'oggetto andrà a sovrascrivere sempre il file più vecchio.

Se si vuole forzare lo scaricamento delle informazioni contenute nel buffer dell'oggetto il metodo `flush` serve allo scopo. L'override del metodo `finalize` obbliga l'oggetto a scrivere sul disco tutte le informazioni pendenti nel buffer, benché sia auspicabile richiamare, opportunamente, il metodo `flush` prima della chiusura del programma.

Questa classe soddisfa il requisito richiesto dalle specifiche del progetto di scrivere su file di

testo tutte le operazioni salienti del programma.

Classe *Message*

La classe Message definisce in modo univoco tutti i messaggi che il client scambierà con gli altri client e con il server. Il suo costruttore è privato. Per poter creare, dunque, una nuova istanza di questa classe bisogna usare i metodi statici definiti opportunamente a seconda del tipo del messaggio che si vuole inviare. Ogni metodo statico definisce il tipo di messaggio attraverso la proprietà `code` che è univoca e attraverso quest'ultimo è possibile identificare ogni messaggio che il client o il server riceve.

L'oggetto una volta creato è immutabile poiché ogni suo attributo interno è definito come *final.* La scelta di mantenere una visibilità pubblica per tutti gli attributi mi ha consentito di evitare di creare dei metodi *'get'* per reperire le informazioni dell'oggetto, ciò se da una parte aumenta lo spreco di memoria, l'oggetto una volta *consumato* viene gettato via, dall'altra aumenta sensibilmente le prestazioni del programma.

Classe *Peer*[1]

La classe Peer definisce in modo univoco l'oggetto *peer* come una coppia di informazioni indirizzo:porta. Gli attributi di questa classe, che sono dichiarati *final* ed hanno visibilità pubblica, sono solo due: l'oggetto `InetSocketAddress idPeer`, che rappresenta l'indirizzo:porta del client e l'intero `hash`, che mantiene l'hashcode univoco dell'oggetto `idPeer`. La classe effettua l'override dei metodi `hashCode` e `equals`, al fine di poter comparare le varie istanze dell'oggetto.

Classe *Torrent*

La classe Torrent definisce e crea oggetti che descrivono i file pubblicati sul Server. La classe divide il file, passato come argomento al costruttore, in blocchi di dimensione di 4kb, per ognuno di questi blocchi viene calcolato il checksum tramite l'algoritmo SHA1 e il risultato viene memorizzato in un array di digest.

Il numero dei blocchi è calcolato in base alla dimensione del file stesso diviso la dimensione di default di un blocco, tutti gli arrotondamenti vengono effettuati per eccesso.

Per ogni blocco scaricato dal Peer si calcolerà il checksum e lo si confronterà con quello memorizzato, solo se i due valori coincidono il blocco verrà ritenuto corretto. La funzione che esegue questo controllo è la `checkBlock`.

La classe mantiene anche il nome del file e la sua dimensione. Per scelta progettuale non è possibile pubblicare sul Server due file con lo stesso nome, anche se diversi come contenuto.

La gestione dei segnali

La gestione dei segnali *sigint* e *sigterm* sia da parte del client che del server permette di eseguire in fase di terminazione alcune azioni fondamentali per la corretta terminazione dei programmi. Ciò è soprattutto vero per il client che, come affronteremo in seguito, implementa una scrittura sul disco bufferizzata dei file che sta scaricando, concettualmente simile alla classe *Logger* già affrontata precedentemente.

Oltre ad attendere la corretta terminazione di ogni thread avviato e chiudere tutte la comunicazioni aperte, in fase di terminazione i due programmi si assicurano di salvare sul disco ogni informazione pendente in memoria.

Terminando il client in altro modo non è assicurata la corretta scrittura sul disco dei blocchi scaricati e degli eventi loggati. Ad ogni modo, è sempre possibile escludere il buffer implementati nella classe *Logger* e nella classe *FileCache*.

Per la gestione dei segnali sono state usate le classi `sun.misc.Signal` e `sun.misc.SignalHandler`.

1 Peer = Nodo chiave della rete funge sia da client, quando scarica un blocco, che da server, quando lo invia, e dunque contribuisce attivamente alla distribuzione del file; differisce dal seeder perché non possiede completamente il file.

Si assicura, inoltre, che i segnali *sigint* e *sigterm* sono gestiti correttamente sia su piattaforma windows che linux.

Il server

Il comando per avviare il server è `java bitcreek.server.Server` e l'unico parametro opzionale che prende è un intero che rappresenta la porta di ascolto del registro rmi che, se non diversamente specificato, verrà avviato usando la porta di default 1099. È possibile, inoltre, definire l'intervallo massimo, tra la ricezione di due keepalive e la conseguente uscita di un peer dallo swarm[2], tramite il comando `-Dkeepalive=x`, dove `x` è un intero che esprime il tempo in millesimi di secondo. Se il parametro non è stato settato verrà usato il valore di default `KEEPALIVE_DEF` definito come attributo statico nella classe `Server`.

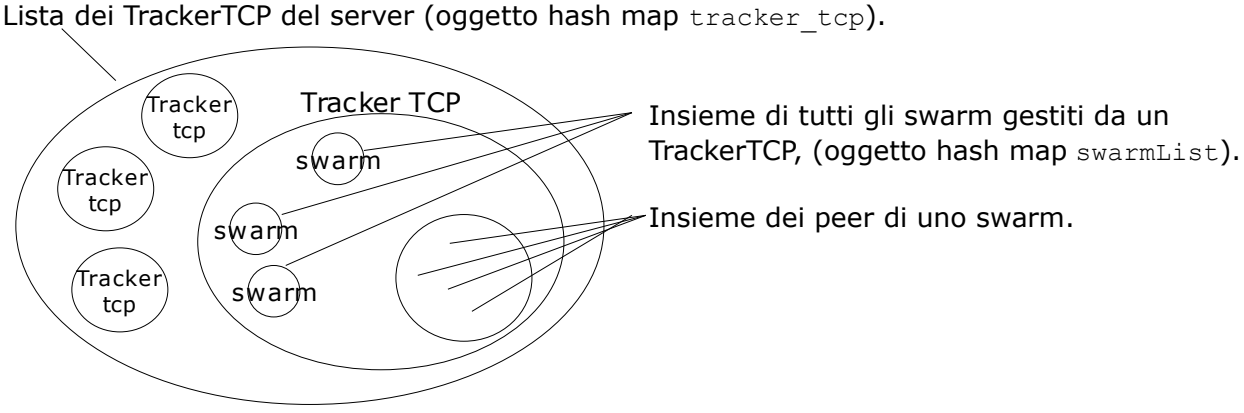
Allo stesso modo è possibile definire il numero massimo degli swarm che un tracker tcp gestirà usando il comando `-Dswarmssize=n` dove `n` è un intero. Es:

```
>java -Dkeepalive=30000 -Dswarmssize=2 bitcreek.server.Server 1100
```

Le strutture dati usate nell'implementazione del server (`ServerRMIImpl`) sono:

- `ConcurrentHashMap<String, ClientInterface> clients`, per la memorizzazione e la ricerca delle interfacce dei client per l'esecuzione delle callback da parte del server.
- `ConcurrentHashMap<String, Torrent> torrentList`, per la memorizzazione e il recupero dei descrittori di file pubblicati sul server.
- `ConcurrentHashMap<Integer, TrackerTCP> tracker_tcp`, per memorizzare e recuperare la lista dei tracker TCP e delle `SwarmList` ad essi associate del server.
  Questa hashmap viene usata per:
  - trovare un tracker disponibile ad accettare un nuovo swarm;
  - in fase di chiusura del programma, per richiamare, per ogni oggetto `TrackerTCP` memorizzato, il metodo `shutdown`, che chiude la server socket e forza la terminazione del thread del tracker.
- `Vector<TrackerUDP> tracker_udp`, per la memorizzazione degli oggetti della classe `TrackerUPD`. In fase di chiusura il server itera in questa lista e per ogni oggetto richiama la metodo `shutdown` che forza la terminazione del thread dell'oggetto e chiude la datagram socket del tracker.

La classe `ServerList`, implementa una altra struttura dati: `ConcurrentHashMap<String, ServerSwarm> swarmList`, che rappresenta l'insieme di tutti gli swarm gestiti da una singola coppia di tracker tcp e udp. A sua volta la classe ServerSwarm, implementa una hashmap `ConcurrentHashMap<Integer, Peer> swarm`, che rappresenta l'insieme di tutti i peer di quello swarm.



2 Swarm = letteralmente sciame, in questo ambito indica la totalità dei peer che condividono lo stesso file.

L'interfaccia del server

Il server implementa l'interfaccia `ServerRMIInterface`, che mette a disposizione dei client quattro metodi:

- `publishDescriptor`: permette la pubblicazione di un nuovo descrittore di file;
- `getDescriptor`: consente il recupero di un descrittore già esistente sul server;
- `registerForCallback`: permette ad un client, che ha pubblicato un descrittore di file "*f*", di registrarsi per la ricezione delle notifiche delle identità di ogni altro peer che effettua la ricerca per quel determinato descrittore "*f*";
- `unregisterForCallback`: permette la cancellazione da parte di un peer che si era precedentemente registrato per l'invio delle callback.

Metodo `publishDescriptor`

Il metodo *publisDescriptor* riceve come parametri formali l'identificativo del peer che sta pubblicando il nuovo descrittore e un' istanza dell'oggetto torrent rappresentante il descrittore dello stesso file.

Come richiesto dalle specifiche del programma i tracker tcp non possono gestire più di n swarm alla volta, dove n è un valore che, come già discusso precedentemente, può essere settato a tempo di compilazione. Quando tutte le `SwarmList` dei tracker attivi sono sature, il programma ne creerà una nuova, che cercherà due porte libere sulla macchina per istanziare una *server socket tcp ssl* e una *datagram socket* univoche, le quali verranno rispettivamente usate dal tracker tcp e dal tracker udp associate al nuovo oggetto `SwarmList`.

I due tracker implementano due threads i quali vengono lanciati per rimanere in ascolto e soddisfare le connessioni in ingresso da parte dei client che si collegheranno per richiedere lo swarm di un particolare descrittore di file (tracker tcp) o effettueranno il keepalive per continuare ad affermare la loro presenza nello swarm stesso (tracker udp).

Il metodo `publishDescriptor`:

- si accerta inizialmente che non esistano altri descrittori di file con lo stesso nome;
- aggiunge il descrittore di file all'oggetto `ConcurrentHashMap<String, Torrent> torrentList`, il quale mantiene la lista di tutti i torrent pubblicati sul server.
- ricerca un tracker tcp con un lista di swarms ancora non satura scorrendo l'oggetto `Vector<TrackerTCP> tracker_tcp`, che mantiene traccia di tutti i tracker tcp creati;
- nel caso che la ricerca abbia dato esito negativo viene creata una nuova `SwarmList`, e lanciati i threads dei tracker tcp e udp associati a quest'ultima, altrimenti viene semplicemente usato l'oggetto `SwarmList` trovato;
- crea un nuovo swarm (`ServerSwarm`) per il descrittore di file che si sta pubblicando e lo aggiunge all'oggetto `SwarmList` recuperato (o creato) precedentemente;
- setta le porte del tracker tcp e udp nel descrittore di file passato al metodo stesso e
- termina, in caso di successo, restituendo al mittente il descrittore di file con le porte dei tracker settate e stampa a monitor un messaggio di successo, altrimenti, restituisce `null` e stampa a monitor un messaggio di errore.

Metodo `registerForCallback`

Il server usa un hashmap `ConcurrentHashMap<String, ClientInterface> clients`, che memorizza tutte le interfacce dei client che hanno pubblicato un descrittore di file sul server, che si sono registrati affinché, in seguito, gli venga notificato l'identità di ogni altro peer che effettua una ricerca per quel descrittore.

Quando un client pubblica, con successo, un descrittore di file invocherà il metodo `registerForCallback`, che aggiunge all'hashmap `clients`, un nuovo elemento definito come una coppia di informazioni <*nome_file*, *interfaccia_del_client*>. Si ricorda che per scelta progettuale non è possibile pubblicare sul server due file con lo stesso nome, tanto meno è permessa la

registrazione per più di un peer per lo stesso descrittore di file, per la ricezione dell'evento di notifica.

Metodo `unregisterForCallback`

La rimozione di un peer dall'hashmap `clients` avviene tramite l'invocazione del metodo `unregisterForCallback`, presente nell'interfaccia del server, che rimuove il client registrato per quel descrittore di file. Il metodo stampa a monitor un messaggio di successo se quel client si era effettivamente registrato per le notifiche su quel determinato file, altrimenti, stampa un messaggio di errore.

Metodo `getDescriptor`

Una volta invocato, il metodo `getDescriptor`:

1. controlla l'effettiva esistenza del descrittore nel server;
2. controlla che la lista degli swarms del tracker tcp associato al descrittore non sia vuota;
3. rimuove i peer vecchi dallo swarm del descrittore;
4. controlla che lo swarm associato al descrittore richiesto non sia vuoto.
   Nel caso di swarm vuoto rimuove:
   - lo swarm dalla lista degli swarm del tracker;
   - il descrittore del file dalla lista dei descrittori pubblicati sul server;
   - la callback associata al descrittore, se ancora presente, dalla lista delle callback del server.
5. controlla l'esistenza di un client, nella struttura dati `clients`, a cui deve essere notificata l'identità del peer che ha richiesto il descrittore di file. Se ne esiste uno, viene invocato il metodo `notifyMe`, presente nell'interfaccia del client, il quale prende come parametri formali il nome del file, su cui si sta effettuando la notifica, e il l'id univoco del peer che ha scaricato tale descrittore;

In qualsiasi caso di errore, il metodo termina stampando un messaggio di errore specifico a monitor e ritornando `null` al chiamante, altrimenti restituisce al peer il descrittore di file richiesto e stampa a monitor un messaggio di successo.

Il client

Il comando per avviare il client è `java bitcreek.client.Client`, specificando obbligatoriamente, come parametri da riga di comando: il file di testo che indica le azioni che dovrà eseguire; l'indirizzo del server e, opzionalmente, la porta del registro rmi sui cui il server è in ascolto, nel caso non sia quella di default 1099.

È possibile, inoltre, definire l'intervallo di invio dei messaggi di keepalive usando il comando `-Dkeepalive=n`, dove n è un intero che esprime il tempo in millesimi di secondo, se il parametro non è stato settato, verrà usato il valore di default `KEEPALIVE_DEF` definito staticamente nella classe `Client`.

È, inoltre, possibile disabilitare l'uso del buffer per la scrittura ritardata dei blocchi su disco attraverso il comando `-Ddirectwrites=true` (l'argomento viene approfondito nel paragrafo che tratta della classe `FileCache`). Es:

```
>java -Dkeepalive=10000 -Ddirectwrites=true bitcreek.client.Client op1.txt
        192.168.1.2 1100
```

La sintassi del file di testo che indica al client le operazioni da eseguire è semplice:

- *pubblica filename:* per pubblicare un descrittore file sul server; il parametro *filename* può contenere spazi e identificare tanto il nome del file quanto il percorso dove questo è memorizzato. Es: *pubblica ../../mio file.mp3*.
- *recupera filename:* per recuperare un descrittore di file pubblicato sul server. Il parametro

filename rappresenta il solo nome del file privo del percorso. Es: *recupera mio file.mp3*.

- *rimuovi filename:* per rimuovere un file che il client ha pubblicato e uscire dallo swarm dello stesso. Se il client non ha pubblicato il file il comando effettua solamente l'uscita del peer dallo swarm di quel file. Anche in questo caso bisogna specificare il solo nome del file. Es:*rimuovi mio file.mp3*.
- *aspetta n:* dove *n* è un long che indica il tempo di attesa per l'esecuzione della prossima istruzione espresso in millesimi di secondo. Es: *aspetta 180000* ritarda la prossima istruzione di tre minuti.

È possibile aggiungere dei commenti al file basta che il primo carattere di ogni riga di commento sia "#". Si tiene a precisare che il client, una volta eseguito il parsing di tutto il file ed eseguite tutte le istruzioni in esso contenute termina, è opportuno, quindi, ritardarne la terminazione con un comando di attesa.

Le strutture dati usati dall'implementazione del client sono:

- `Hashtable<String, ClientSwarm> swarmsList`, statica e con visibilità protetta che conterrà tutti gli oggetti di tipo `ClientSwarm`, ognuno dei quali rappresenta uno swarm associato ad un file che il client sta scaricando e distribuendo.
- `Hashtable<String, Torrent> torrentList`, dichiarata con visibilità protetta che conterrà tutti i descrittori dei file che il client sta scaricando e distribuendo.
- `List<String> callback = Collections.synchronizedList(new Vector<String>())` che conterrà tutte le callback che il client ha registrato sul server per i file pubblicati.

Identificazione dei Peer

Ogni client viene identificato univocamente, come già accennato nel paragrafo trattante la classe peer, da un indirizzo e un numero di porta.

L'individuazione dell'indirizzo del client viene affidato al metodo statico `getComputerAddress`, implementato nella classe `ClientImpl`, che esegue una enumerazione di tutte le schede di rete del computer e per ogni scheda esegue l'enumerazione di tutti gli indirizzi ad essa associati. Da questa lista vengono rimossi tutti gli indirizzi di rete ipv6 e gli indirizzi di *loopback;* per ogni indirizzo si verifica se esso è effettivamente raggiungibile. Il risultato di questa operazione viene salvato in un array di `InetAddress`. Il primo indirizzo dell'array verrà usato per identificare il client. Diversamente, se la lista dovesse risultare vuota, viene lanciata un'eccezione.

Si è reso necessario implementare questa funzione per risolvere un problema verificatosi con Ubuntu. Infatti, dalla versione 8.04 la chiamata `InetAddress.getLocalHost`, a causa di un errata configurazione del file /etc/host, restituiva l'indirizzo di *loopback*, assolutamente inutile ai fini dell'identificazione del client.

L'individuazione della porte del client viene affidata alla funzione statica `getsocket`, implementata nella classe `ClienImpl`, che restituisce un `ServerSocket`, testando le porte disponibili sulla macchina a partire dalla 2025. La socket verrà usata dal client per accettare tutte le comunicazioni in ingresso per la richiesta dei blocchi da parte degli altri peer.

L'unione di queste due informazioni, indirizzo della macchina e numero di porta di ascolto del client, vengono salvate nella variabile `InetSocketAddress myId` che verrà usata come identificativo del client in tutte le sue comunicazioni verso altri peer o verso il server.

Pubblicazione di un torrent

Eseguito il parsing del comando *pubblica filename*, il client esegue il metodo `pubblica` presente nella classe `ClientImpl`, che prende, come parametro formale, una stringa che identifica il percorso e il nome del file da pubblicare. Creato il torrent, il metodo si accerta che non ne esista già uno con lo stesso nome nell'hashtable `torrentlist;` se questa condizione non si è verificata si esegue il metodo `publishDescriptor` dell'interfaccia del server.

Il descrittore aggiornato dei valori delle porte dei tracker tcp e udp, restituito dal metodo

`publishDescriptor` del server viene aggiunto all'hashtable `torrenList`. Il metodo crea un oggetto, `FileCache`, che verrà usato in seguito per effettuare tutti gli accessi in lettura al file sul disco.

Crea un nuovo swarm, lo associa all'oggetto `filecache` e al descrittore di file pubblicato sul server e lo aggiunge alla lista di swarm del client (`swarmsList`).

Al fine di aver notificata l' identità di tutti i peer che richiedono il descrittore al server, il metodo `pubblica` esegue il metodo `registerForCallback` dell'interfaccia del server e aggiunge nella lista delle callback (`callback`) un nuovo elemento col nome del file pubblicato.

Richiesta di un torrent pubblicato sul server

Quando il client esegue il parsing del comando *recupera filename*, esegue il metodo *retrive* che:

- controlla che il descrittore di file non sia già presente nel client;
- contatta il server rmi, indicando il nome del descrittore che sta richiedendo e la sua identità;
- restituisce al client il descrittore già presente nel server e crea un nuovo oggetto `FileCache`;
- crea un nuovo oggetto `ClientSwarm` e contatta il tracker tcp associato al descrittore per richiedere i peer dello swarm, usando il metodo `getSwarmFromServer` dell'oggetto `ClientSwarm`;
- memorizza nella sua `torrentList` il descrittore di file restituito dal server, ed il nuovo swarm nell'hashtable `swarmsList`;
- controlla se il client possiede il file nella sua interezza, in caso negativo lancia il thread dell'oggetto `ClientSwarm`, che si preoccuperà di scaricare il file.

Rimozione del client da uno swarm

Il parsing del comando *rimuovi filename*, provoca l'esecuzione del metodo corrispondente `removeDescriptor`, che esegue nell'ordine le seguenti azioni:

- controlla l'esistenza del descrittore nel client, se già esiste lo rimuove dalla lista dei descrittori, altrimenti ritorna `false` e stampa un messaggio di errore a monitor;
- controlla l'esistenza di un oggetto `ClientSwarm`, registrato col il nome del descrittore stesso, se già esiste lo rimuove dall'hashtable `swarmsList`;
- setta a `true` la variabile booleana `shutdown` dell'oggetto `ClientSwarm`. La variabile forza la terminazione di tutti i thread che gestiscono le comunicazioni in uscita verso i peer dello swarm e la chiusura di tutte le comunicazioni sia in ingresso che in uscita attive per quello swarm.

Classe *ClientSwarm*

La classe ClientSwarm mantiene tutte le informazioni riguardanti lo swarm di un dato descrittore di file.

Strutture dati

In particolare, la struttura dati `ConcurrentHashMap<Integer, PeerTalk> swarm`, viene usata dall'oggetto per memorizzare la lista dei peer ricevuta dal server. Allo `swarm` possono aggiungersi nuovi peer in qualsiasi momento, ad esempio, se un nuovo peer, di cui prima non se ne conosceva l'esistenza, contatta il client, quest'ultimo viene aggiunto automaticamente all'hashmap `swarm`. Questa scelta implementativa è stata adottata per aumentare le possibilità di ogni peer di completare il file che sta scaricando.

La struttura dati `ConcurrentHashMap<Integer, PriorityList> blockList`, viene usata per memorizzare oggetti della classe privata `PriorityList`; la chiave che viene associata ad ogni oggetto della'hashmap è un intero univoco rappresentante l'indice del blocco da scaricare.

Il `Vector<PriorityList> prioritylist`, è usato anche esso per memorizzare oggetti della classe `PriorityList`, ordinati per rarità.

È bene spendere due righe sulla classe privata `PriorityList`.

Classe privata PriorityList

La classe privata `PriorityList` ha solo tre campi informativi, due interi uno che rappresenta l'indice del blocco da scaricare (`indexblock`), l'altro che rappresenta il numero dei peer che lo possiedono (`count`). L'ultima variabile è l' `AtomicBoolean download` che, se settata a `true`, indica che il blocco è stato richiesto ad un peer.

Di questa classe, ciò che è importante, è che essa implementa l'interfaccia *Comparable* che, attraverso il metodo `compareTo`, permette di dare un ordine crescente ai suoi elementi, quando questi sono stati inseriti in una lista.

Ricapitolando, ogni oggetto `PriorityList` rappresenta un blocco da scaricare, mentre la lista che restituisce i blocchi ordinati per rarità è l'oggetto `Vector<PriorityList> prioritylist`, della classe `ClientSwarm`. Per scelta implementativa gli oggetti presenti nel `Vector<PriorityList> prioritylist` vengono eliminati solo quando il blocco di file ad essi associato è stato effettivamente scaricato.

WarmUp

In fase di inizializzazione, l'oggetto controlla se possiede il file completamente, in caso negativo contatta il tracker tcp, al fine di ricevere la lista dei peer che appartengono allo swarm.

Ricevuta correttamente la lista, viene creato per ogni oggetto `Peer` della lista, un oggetto della classe `PeerTalk`, che viene memorizzato nella struttura dati `swarm`.

Un oggetto della classe `PeerTalk`, gestisce la comunicazione verso un peer remoto e mantiene le informazioni sul bitset dei blocchi che questo possiede. L'oggetto `PeerTalk` contatta il peer remoto a cui è associato e gli richiede il bitset, una volta che questo ha reso l'informazione la comunicazione viene chiusa. Ogni bitset rappresenta la lista dei blocchi che un peer possiede, mentre il numero dei blocchi posseduti è dato dalla sua cardinalità.

Effettuata questa operazione con tutti i peer, il programma ha una visione d'insieme dei client che compongono lo swarm e dei blocchi che questi possiedono. Ciò permette di effettuare delle scelte su quali peer preferire in base al numero dei blocchi posseduti e in base alla rarità degli stessi.

Vengono creati in questa fase tanti oggetti `PriorityList` quanti sono i blocchi mancanti del file. Ognuno di questi viene memorizzato sia nella struttura dati `blockList`, che nella struttura dati `prioritylist`. La prima permette di accedere agli oggetti che identificano la priorità di un blocco tramite il loro indice, la seconda permette di accedere agli stessi oggetti ordinati per rarità.

Ogni qual volta si riceve un bitset da parte di un peer questo viene confrontato con la lista dei blocchi posseduti. Per ogni blocco, che il peer remoto possiede e il client no, si incrementa di uno la variabile `count` dell'oggetto `prioritylist` corrispondente.

Eseguita questa procedura iniziale di *warmUp*, i peer dello swarm vengono ordinati per numero di blocchi posseduti e vengono contattati dando maggiore priorità a quelli che possiedono più blocchi.

Gestione delle connessioni

Il numero di connessioni remote che si possono effettuare per ogni swarm è definito dalla variabile statica `MAX_CONN_IN` definita nella classe `Client`. Quindi, se il numero dei peer dello swarm è inferiore al numero massimo di connessioni in uscita, verranno stabilite tante connessioni quanti sono i peer presenti nello swarm. Diversamente verranno stabilite tante connessioni fino raggiungere il valore stabilito con `MAX_CONN_IN`.

Il conteggio delle connessioni attive, che il client ha stabilito per questo swarm, viene mantenuto dalla variabile `AtomicInteger connection`.

Se viene raggiunto il numero massimo di connessioni possibili, l'esecuzione del thread dell'oggetto `ClientSwarm` si sospende rimanendo in attesa sull'oggetto `connection`.

Le connessioni verso i client dello swarm vengono eseguite lanciando i corrispettivi threads degli

oggetti `peertalk` che autonomamente richiedono i blocchi ai peer remoti.

Metodo getRarestBlock

Il thread dell'oggetto `peertalk` richiama il metodo `getRarestBlock` della classe `ClientSwarm`, e gli passa come parametro attuale il bitset del peer remoto. Il metodo confronta il bitset con la lista dei blocchi da scaricare e identifica il blocco più raro posseduto dal peer remoto (Si ricorda che un blocco è rappresentato da un oggetto della classe `PriorityList`). Se l'oggetto `PriorityList` esiste, il suo parametro `AtomicBoolean download` viene settato a `true`, al fine di impedire la richiesta multipla dello stesso blocco a più peer.

Infatti, il metodo `getRarestBlock` scarta tutti gli oggetti `PriorityList` aventi questo parametro settato a `true`. Viene, infine, ritornato al chiamante l'oggetto `PriorityList` identificato, altrimenti, se il peer remoto ha solo blocchi che sono già in possesso del client, il metodo ritorna `null`.

Nel caso in cui al chiamante viene restituito un oggetto `PriorityList`, questo ne controlla l'indice e richiede il blocco al peer remoto. Ricevuto il blocco, ne viene controllata la correttezza calcolandone il digest che viene confrontato con quello del descrittore del file. Il metodo `put`, della classe `FileCache`, esegue questa operazione e ritorna `true` in caso positivo.

Una volta che il blocco è stato scaricato l'oggetto p`riority`l`ist` viene rimosso, tramite la chiamata al metodo `removeBlock` dell'oggetto `ClientSwarm`, dal vettore `prioritylist` e dalla struttura dati `blockList`.

Se per un qualsiasi motivo la richiesta del blocco fallisce, il parametro `AtomicBoolean download` dell'oggetto p`riority`l`ist` viene settato a `false`, permettendo così la richiesta del blocco ad un altro peer dello swarm..

Nel caso in cui il metodo `getRarestBlock` della classe `ClientSwarm`, restituisca `null`, l'oggetto `peertalk` richiederà il bitset dei blocchi aggiornato al suo peer remoto. Questa scelta implementativa permette di aumentare le possibilità del client di completare il file, poiché se il peer remoto nel frattempo ha scaricato dei nuovi blocchi quest'ultimi vengono segnalati con il nuovo bitset ricevuto.

Tuttavia, se il peer remoto ritorna consecutivamente un bitset non aggiornato, la comunicazione termina e il peer viene rimosso dallo `swarm`.

Ogni qual volta una comunicazione con un peer remoto termina, o per iniziativa del client o per iniziativa del peer remoto, il thread dell'oggetto `peertalk` decrementa il numero di connessioni attive ed esegue una notifica sull'oggetto `connection`. Se, precedentemente, si era raggiunto il numero di comunicazioni massime possibili per questo swarm, questo evento risveglierà il thread dell'oggetto `ClientSwarm`.

Questo thread una volta risvegliatosi, controllerà l'esistenza di un peer, iterando nello `swarm`, con cui non si è ancora stabilita una connessione. Se il peer esiste lo contatterà lanciando il thread dell'oggetto `peertalk` corrispondente.

Nel caso in cui tutti i peer dello swarm sono stati contattati senza che il limite delle connessioni in uscita sia stato raggiunto, al fine di evitare che il thread dell'oggetto `ClientSwarm` iteri costantemente sugli elementi della struttura dati `swarm`, questo si sospende autonomamente per risvegliarsi ad intervalli regolari. Ad ogni risveglio il thread controlla che ci siano dei nuovi peer con cui è possibile stabilire una connessione e li contatta opportunamente.

Ogni qual volta una comunicazione con un peer viene interrotta, oppure un blocco ricevuto è errato, o non è più possibile ricevere blocchi utili da un peer, questo viene rimosso dallo swarm del client. È facile notare che, con queste politiche lo swarm si può svuotare in fretta e il client non sarebbe più in grado di completare il file. Per evitare una situazione del genere è stata prevista la possibilità di richiedere al server nuovamente lo swarm del file.

Se lo swarm esiste questo verrà restituito e la procedura ricomincerà nuovamente dall'inizio come descritto. Diversamente se il server restituisce uno swarm vuoto, dove per vuoto si intende che l'unico peer dello swarm è il client stesso, il client richiederà ad intervalli regolari lo swarm al server, nel tentativo di completare il file.

Classe *Listener*

A grandi linee si può dire che la classe `Listener` si preoccupa di accettare le comunicazioni ingresso da parte dei peer remoti. La sua classe privata, `MessageIn`, è quella che effettivamente espleta il compito di ricevere i messaggi, interpretarli ed rispondere coerentemente con le richieste ricevute dai peer.

Viene instanziato dal client un nuovo oggetto della classe `Listener` e messo in esecuzione il suo thread, che rimane in ascolto sulla server socket passata come parametro attuale al momento della creazione dell'oggetto.

Ad ogni nuova connessione con un peer remoto viene creato un nuovo oggetto della classe `MessageIn` e messo in esecuzione il suo thread. Nelle prime fasi della comunicazione viene individuato lo swarm a cui il peer remoto appartiene. Se il client non sta scaricando quel file viene rifiutata la connessione.

Se il client, invece, appartiene allo swarm richiesto, si controlla che il peer locale non abbia raggiunto il numero massimo di connessioni in ingresso per lo swarm individuato precedentemente.

Inoltre, per scelta implementativa, non è possibile che un peer remoto apra due comunicazioni con un client per lo stesso swarm. Nei due casi precedenti la comunicazione con il peer remoto viene rifiutata.

Se la connessione con il peer remoto viene accettata l'oggetto `MessageIn` viene inserito nella struttura dati `ConcurrentHashMap<Integer, MessageIn> skPeers` dell'oggetto `ClientSwarm` associato al file richiesto dal peer remoto. Questa struttura dati `skPeers` mantiene in memoria la lista di tutte le comunicazioni in ingresso accettate per quello swarm. Ogni qual volta che la comunicazione tra i due peer, si perde o termina, il thread dell'oggetto `MessageIn` rimuove autonomamente l'oggetto a cui appartiene dalla struttura dati `skPeers`.

Questa struttura, oltre a conteggiare il numero di comunicazioni remote attive, viene usata, sia in fase di terminazione del client, sia in caso di uscita dello stesso dallo swarm, per richiamare su tutti gli oggetti `MessageIn` memorizzati il metodo `shutdown`, che chiude la socket di comunicazione e forza la terminazione del thread.

Classe *FileCache*

La classe *FileCache* si occupa di eseguire la scrittura, la lettura dei blocchi del file che si sta scaricando e/o inviando e implementa un buffer per i blocchi, al fine di ottimizzare l'accesso al disco e il loro reperimento. Il buffer ha una dimensione di default di mille blocchi, ed è usato solo per memorizzare temporaneamente un blocco prima della sua scrittura sul disco.

Quando si crea un nuova istanza della classe *FileCache,* essendo il suo costruttore privato, bisogna usare i metodi statici *`createFileCacheFromRemoteTorrent`* e *`createFileCacheFromLocalTorrent`* che differiscono nel settare a `false` nel primo e `true` nel secondo caso, il secondo parametro formale del costruttore della classe.

Questo boolean cambia il comportamento della classe in base ad un'importante esigenza: se il descrittore di file è stato pubblicato da questo peer il percorso di dove si trova il file nel disco è definito dal client stesso che l'ha pubblicato, altrimenti se il peer ha scaricato il descrittore di file dal server vorrà scaricare il file in un'opportuna cartella dove verranno posti tutti i file in download.

La cartella di default, dove si salvano i file che si stanno scaricando, è appunto "*download*".

Quando una nuova istanza di questa classe viene creata da un peer che non possiede il file o ne possiede solo delle parti, l'oggetto controlla l'esistenza del file nel disco, se questo è presente controlla blocco per blocco la correttezza del file, i blocchi la cui correttezza è verificata sono settati come posseduti nel *bitset* dell'oggetto, evitando di doverli scaricare nuovamente.

Quando un blocco deve essere reperito per essere inviato ad un peer remoto, ne viene dapprima controllato l'effettivo possesso, successivamente viene cercato nel buffer, se la scrittura differita è abilitata (`direct_writes == false`), e restituito alla funzione chiamante. Se la ricerca fallisce questo viene letto direttamente dal disco e restituito alla funzione che lo ha richiesto per essere inviato al peer.

Il buffer è attivo solo durante la fase di scaricamento dei blocchi, una volta che il peer diventa un seeder[3] il buffer viene svuotato di tutto il suo contenuto che sarà scritto definitivamente sul disco.

Aumentando la dimensione del buffer si avranno migliori prestazioni, in quanto ci saranno minori scritture sul disco, di contro, però, ci sarà un maggiore spreco di memoria.

L'oggetto mantiene memoria di tutti i blocchi che il peer possiede per quel file, attraverso l'istanza `bs_blocks` della classe `BitSet`. Ogni qualvolta un nuovo blocco viene scaricato si calcola il checksum e lo si confronta con quello del descrittore di file richiamando il metodo `checkblock` dell'oggetto torrent associato, se il confronto è positivo il corrispondente bit dell'oggetto `bs_blocks` sarà settato a 1 e il blocco verrà salvato.

Il metodo `flush` forza il salvataggio nel disco delle informazioni contenute nel buffer, se questo è attivo, mentre il metodo `close` oltre a salvare le informazioni sul disco chiude il file.

L'override del metodo `finalize` obbliga l'oggetto a scrivere sul disco tutte le informazioni pendenti nel buffer, tuttavia è preferibile richiamare il metodo `close` prima della chiusura del programma.

Ricordo che se il programma viene ucciso con un *sigkill* nessuna informazione contenuta nel buffer verrà salvata, ad ogni modo, è sempre possibile escludere l'uso di qualsiasi buffer da parte del programma settando a tempo di esecuzione `-Ddirectwrites=true`, altrimenti settando a true l'attributo statico `DIRECT_WRITES` della classe *Client*.

Classe *KeepAlive*

La classe *KeepAlive* si preoccupa di inviare, ad intervalli regolari di tempo, un messaggio di *keepalive*, tramite l'uso di una *socket udp*, al Server per notificare la propria presenza nello swarm a cui il client si è registrato.

Si invia un messaggio di *keepalive* per ogni swarm a cui il peer appartiene; il mancato invio di uno di questi messaggi per un periodo di tempo, superiore al tempo massimo definito nel Server, provoca l'uscita del client da quel determinato swarm.

Questo è l'unico metodo che il client ha per potersi cancellare da uno swarm. L'intervallo di tempo è definito nella classe `Client` del programma attraverso l'attributo statico `keepAliveTime` che può essere settato a tempo di esecuzione attraverso il comando `-Dkeepalive=n`, dove `n` è un intero che esprime l'intervallo di tempo in millisecondi tra un keepalive ed un altro.

Altrimenti, se non diversamente specificato, verrà usato il tempo di default definito dall'attributo statico `KEEPALIVE_DEF` presente nella classe `Client` del programma.

In ogni messaggio, mandato al Server, il peer indica: l'indirizzo e la porta tcp su cui è in ascolto e che lo identificano in modo univoco, nonché il nome del file, che identifica univocamente lo swarm.

3 Seeder = Sono i client che possiedono il file completo.