PoliShare

Architettura e Design

1 Introduzione

Questo documento descrive l'architettura e il design di PoliShare, applicazione che permette la condivisione peer-to-peer di appunti tra gli studenti iscritti al sistema.

Per poter analizzare la struttura del sistema in oggetto verranno utilizzati 3 dei 5 punti specificati nel modello 4+1[1] di descrizione di un'architettura software:

- **Logical View** – descrive i componenti principali del sistema, le operazioni che svolgono e le loro relazioni e interazioni.
- **Process View** – descrive gli aspetti dinamici del sistema mentre esegue le operazioni descritte nella logical view.
- **Use Case View** – vengono presentati alcuni casi d'uso dell'applicazione, con lo scopo di illustrare e allo stesso tempo validare il design del sistema.

Per gli altri punti del modello:

- La **Physical View** non verrà trattata in quanto esule dallo scopo di questo progetto: non viene infatti fornita alcuna reale "distribuzione" del software, rendendo quindi inutile l'analisi di questo punto.
- Nemmeno la **Development View** sarà trattata: trattandosi di un sistema di dimensioni ridotte, la developent view non porterebbe alcuna informazione aggiuntiva rispetto alla prima fase della Logical View, per questo non verrà trattata.

Oltre a questi punti verranno anche aggiunte:

- una sezione introduttiva per spiegare i problemi affrontati e gli algoritmi risolutivi utilizzati;
- una sezione per elencare le tecnologie utilizzate;
- una sezione per documentare i test realizzati.

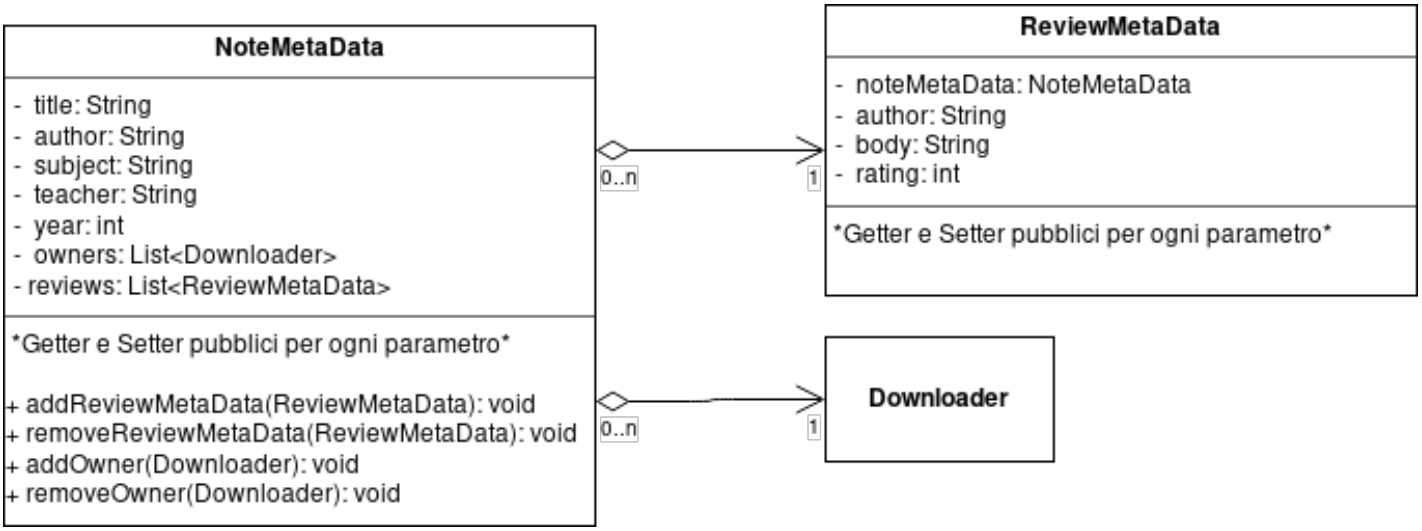
2 Algoritmi

Polishare è un'applicazione per la condivisione peer to peer di appunti (sotto forma di file PDF). A questi appunti possono essere associati dei dati quali ad esempio l'autore, il titolo e varie recensioni. Nella realizzazione del software i principali problemi affrontati sono stati 3:

- come individuare un file all'interno di una rete decentralizzata?
- come distribuire gli stessi dati a più nodi mantenendone la consistenza?
- come trasferire i file in maniera efficiente tra i nodi della rete?

Per affrontare questi problemi si è deciso di disaccoppiare il trasferimento vero e proprio dei file dalla corrispondente gestione dei dati, allo scopo di alleggerire il numero e la

dimensione dei messaggi scambiati all'interno della rete. Sono state quindi definite due classi principali, che rappresentano i dati relativi agli appunti e alle rispettive recensioni: NoteMetaData e ReviewMetaData.
La classe NoteMetaData contiene tutte le informazioni direttamente collegate ad un appunto pubblicato in rete (titolo, autore, corso, professore, anno), una lista delle recensioni associate e una lista degli effettivi possessori del file. La classe ReviewMetaData contiene invece i dati relativi ad una recensione: il riferimento all'appunto recensito, l'autore, una valutazione da 0 a 5 e il corpo testuale della recensione.



(I dettagli della classe Downloader saranno introdotti in seguito, al momento sono superflui)

La lista dei possessori (owners) contenuta nei NoteMetaData permette di poter contattare direttamente i nodi possessori di un file per effettuare il download.

Disponendo quindi di una lista di nodi da poter contattare per scaricare il file, il problema a questo punto diventa: come mettere a disposizione tale lista per tutti i nodi? E anche, come mantenere aggiornata questa lista a fronte di tutte le possibili operazioni e fallimenti dei nodi nella rete?

2.1 Distributed Hash Table

Per poter distribuire la lista aggiornata dei possessori (e più in generale tutti i dati associati ad un appunto) a tutti i nodi della rete le possibili strade sono due:

1. **Replicare i dati** e le operazioni eseguite su ogni nodo, mediante algoritmi di *state machine replication* in un rete di nodi non affidabili (ad esempio Paxos).

   PRO:
   - il meccanismo di consultazione dei dati è molto veloce (non si devono interrogare altri nodi);
   - la fault tolerance è intrinseca nell'algoritmo;

CONTRO:
- elevata ridondanza dei dati che vengono replicati su ogni nodo;
- il meccanismo di aggiornamento dei dati è molto lento;
- design complicato e difficile da implementare;

2. Trovare un sistema automatico in grado di **mappare i dati ad un preciso nodo**, realizzando una vera e propria tabella distribuita su più nodi.

   PRO:
   - basso overhead di comunicazione e di distribuzione dei dati;
   - design semplice;
   - tutte le operazioni (consultazione e aggiornamento dei dati) hanno prestazioni analoghe;

   CONTRO:
   - servono misure "speciali" per ottenere la fault tolerance;
   - la consultazione efficiente di più dati in contemporanea (broadcast) non è facilmente ottenibile.

La scelta è ricaduta sulla creazione di una tabella di hash distribuita (DHT), per le seguenti ragioni:

- in uno scenario reale l'hardware dei nodi che entrano nella rete è parecchio eterogeneo ed alcuni nodi soffrirebbero la gestione di una gran quantità di dati;
- in questo caso i benefici forniti da un algoritmo come può essere Paxos non giustificano gli sforzi necessari per ottenerne una corretta implementazione.

Per realizzare la DHT si è scelto l'algoritmo **Chord**, che fornisce una tabella distribuita basandosi su un **consistent hash ring**. I dati vengono identificati univocamente all'interno della tabella in base al titolo degli appunti a cui si riferiscono (scelto dall'utente al momento della pubblicazione) e grazie ad un piccolo sistema di replicazione si riesce ad ottenere la fault tolerance: nel caso in cui un nodo fosse improvvisamente scollegato dalla rete, i dati gestiti in quel momento verrebbero trasferiti su altri nodi.

La trattazione dettagliata di tale algoritmo esula dagli obiettivi di questo documento. Per una formalizzazione corretta si rimanda alla bibliografia[2][3], così come per l'algoritmo che consente il broadcast in una rete chord[4]. Per i dettagli implementativi si rimanda invece ai JavaDoc del progetto.

2.2 Trasferimento dei file

Una volta individuati i possessori di un file, serve un algoritmo in grado di trasferire il suddetto file tra uno e più nodi. Il trasferimento dei file deve essere:

- **Corretto:** il file non deve arrivare danneggiato al destinatario;
- **Robusto:** deve poter gestire fallimenti arbitrari durante il trasferimento;
- **Efficiente:** nei limiti della connesione utilizzata.

L’algoritmo utilizzato per il trasferimento dei file (per i dettagli di processo si veda la Process View) è il seguente:

download(title, List<> owners):

1. Seleziono un nodo attivo tra gli owners: il master. Nel caso in cui non fosse individuato alcun nodo attivo, termino.
2. Richiedo al master la dimensione in byte del file da scaricare e il valore dell’hash MD5 del file. Nel caso in cui le richieste fallissero, ritorno al punto 1.
3. Calcolo il numero di segmenti in cui dividere il file e gli indici dei byte di inizio e di fine di ogni segmento.
4. Richiedo in parallelo un segmento al primo nodo degli owners, poi uno al secondo e così via fino a che non termino i segmenti. Nel caso in cui la lista degli owners fosse terminata, riparto dall’inizio. Nel caso in cui un segmento non arrivi entro il tempo di timeout, viene richiesto all’owner successivo nella lista. Nel caso non vi fossero più owner attivi, termino.
5. Una volta ricevuti tutti i segmenti, calcolo l’MD5 del file ricevuto e lo confronto con quello ottenuto inizialmente dal master. In caso di match, posso proseguire.
6. Salvo il file
7. Mi aggiungo alla lista degli owners del suddetto file.

Tale algoritmo garantisce la correttezza del trasferimento tramite il controllo dell’MD5 del file, la robustezza in quanto i fallimenti nel download di un segmento non invalidano il download di tutto il file (l’unico caso sarebbe quello in cui non vi fosse più alcun owner attivo, caso nel quale il download risulterebbe comunque impossibile) e l’efficienza in quanto piccoli segmenti vengono scaricati in parallelo da nodi differenti.

Alcune note sull’algoritmo di download:

- il progetto è basato sull’assunzione di una rete “piatta”: non vengono prese in considerazione velocità e banda disponibile in una connessione. Prendendo in considerazione questi dati, si potrebbe migliorare l’algoritmo di download interrogando gli owner dotati di una connessione migliore.
- nel caso in cui un solo nodo attivo possedesse il file richiesto, l’algoritmo “degenererebbe” in un normale download punto a punto.

2.3 Conclusioni

Chord fornisce un ottimo sistema per l’individuazione di un file all’interno della rete mentre l’algoritmo del punto 2.2 può essere utilizzato per scaricare il file in maniera effciente.

Oltre a funzionalità di “routing”, l’algoritmo Chord fornisce anche un servizio consistente di mantenimento dei dati, mettendoli a disposizione di tutti i nodi all’interno della rete.

3 Tecnologie

PoliShare è un'applicazione desktop realizzata in Java.

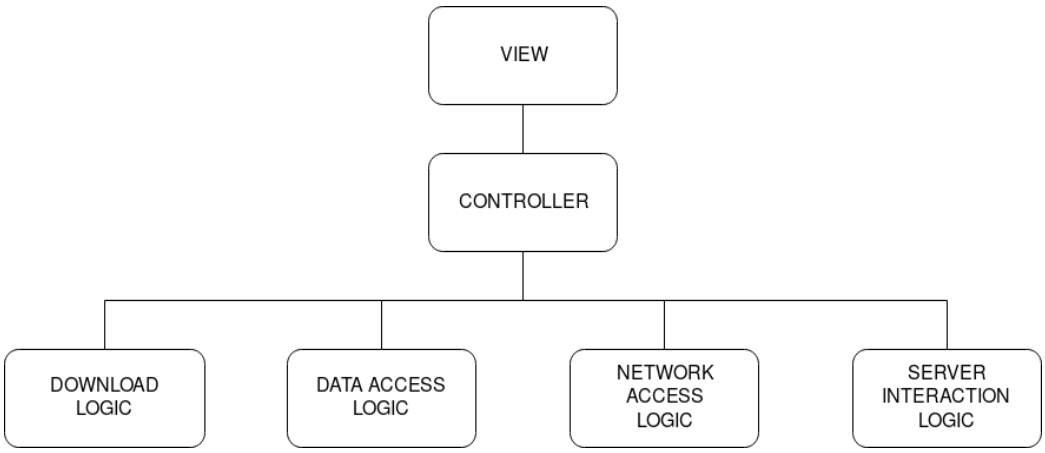
- L'interfaccia grafica è stata realizzata mediante JavaFX combinato con librerie esterne come DataFX (libreria che semplifica lo sviluppo di applicazioni in JavaFX) e JFoenix (libreria di componenti grafici che seguono i principi del Material Design)
- Il DBMS utilizzato è SQLite3.
- Viene sfruttato RMI per la chiamata remota di metodi.
- I test sono stati realizzati utilizzando JUnit 5.
- Le dipendenze sono state gestite utilizzando Maven.

4 Logical View

In questa sezione i moduli del sistema sono inizialmente presentati in termini di componenti di alto livello per poi essere analizzati nel dettaglio nei punti principali.
Nonostante l'applicazione abbia principalmente un'architettura peer-to-peer il sistema possiede anche un piccolo server per svolgere funzioni di autenticazione e interazione tra utenti. Di seguito verranno quindi presentati sia i moduli presenti sui client che quelli presenti sul server.

4.1 High-level Design

4.1.1 Client



Per la realizzazione del client è stato utilizzato il pattern MVC (Model-View-Controller).

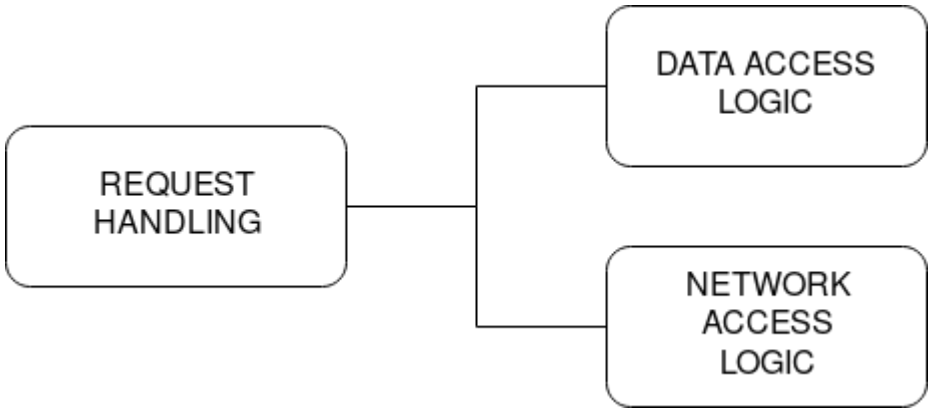
- **View** – gestisce le interfacce grafiche presentando le informazioni all'utente finale.
- **Controller –** risponde agli input dell'utente interagendo con i componenti del Model, per poi passare alla View i risultati di tali interazioni.

Le componenti View e Controller non saranno approfondite nelle sezioni successive. Per i dettagli implementativi si rimanda ai JavaDoc del progetto. Il Model è segmentato in 4 componenti principali:

- **Server Interaction Logic –** gestisce le interazioni con il server (es. autenticazione e chat).
- **Network Access Logic** – gestisce l'accesso e l'interazione con la rete p2p a cui il peer è collegato.
- **Data Access Logic** – gestisce l'accesso al database dei propri file.
- **Download Logic** – gestisce il trasferimento di file tra i vari peer.

Nelle sezioni successive verranno approfonditi i componenti nei quali è suddiviso il Model.

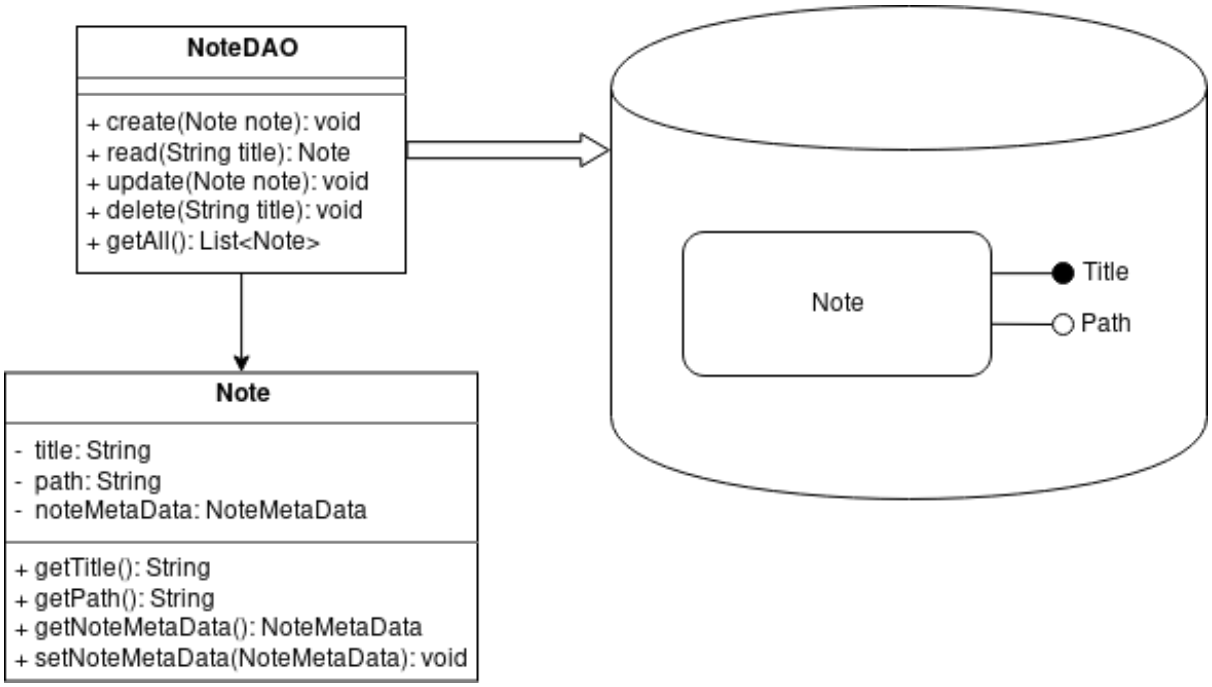
4.1.2 Server



- **Request Handling –** gestisce le richieste di interazione provenienti dai client collegati.
- **Data Access Logic** – come nel client, gestisce l'accesso ad una propria base di dati per la gestione delle autenticazioni.
- **Network Access Logic –** il server è a tutti gli effetti anche un membro della rete p2p e quindi possiede tutta la logica necessaria all'accesso a tale rete. Si tratta di un nodo "passivo": non effettua mai interrogazioni, tuttavia funge da entry point per gli altri nodi (garantendo sempre la presenza di un nodo online).

4.2 Low-Level Design

4.2.1 Data Access Logic



(Diagramma UML dell'accesso al database sul client, più relativo diagramma ER)

Sia sul client che sul server è presente una base di dati. Sul client serve a tenere traccia degli appunti scaricati e della loro posizione nel file system locale, sul server serve invece a tenere traccia delle credenziali degli utenti registrati nel sistema.

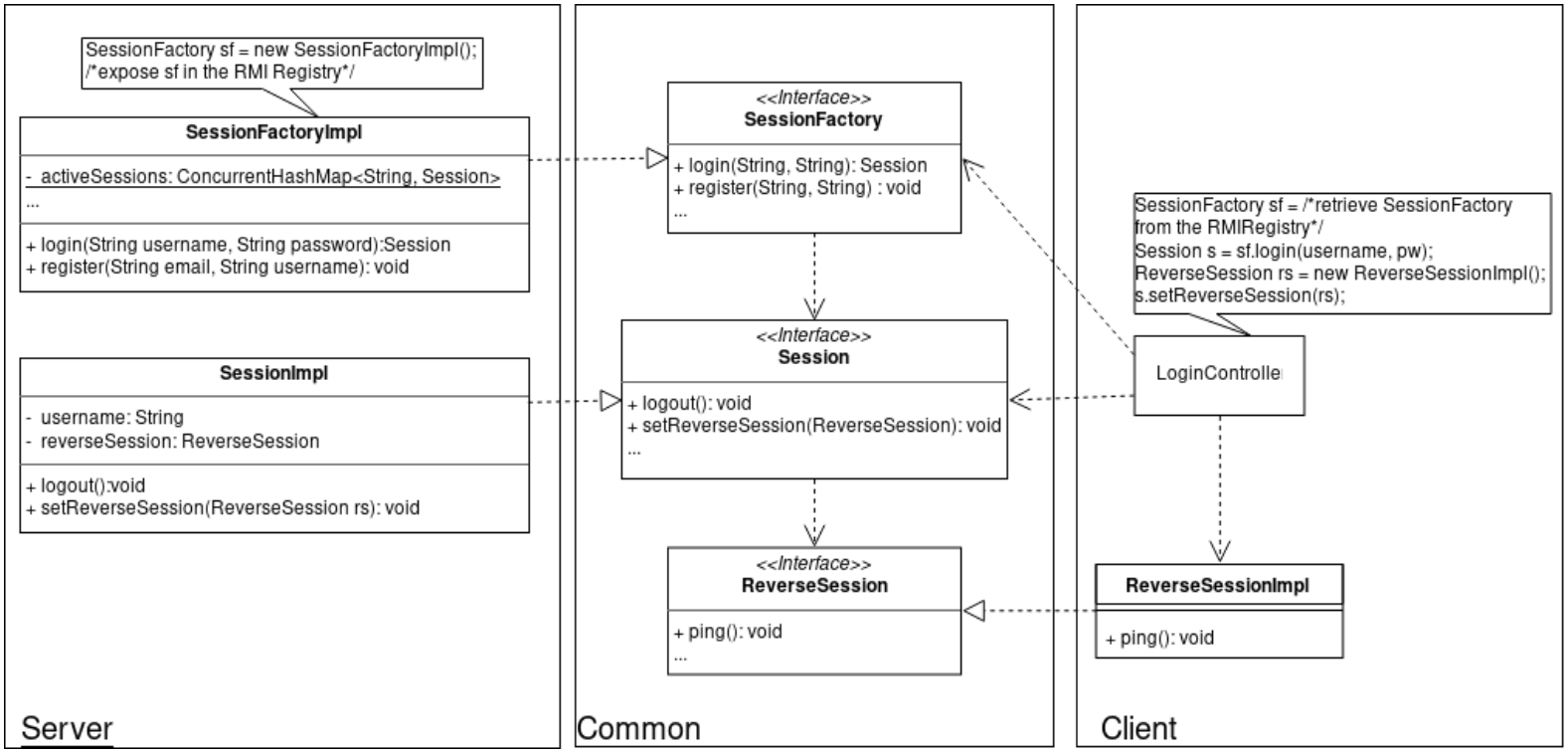
In entrambi i casi l'accesso ai dati viene effettuato sfruttando il pattern Data Access Object[5]: un'oggetto che mappa le chiamate dell'applicazione sul database astraendone l'interazione diretta. Grazie a questo livello di astrazione la logica di accesso ai dati risulta disaccoppiata dal database utilizzato.

4.2.2 Server Interaction Logic e Request Handling

Queste componenti gestiscono tutto il processo di registrazione e autenticazione degli utenti interagendo attraverso la rete (tramite RMI).

Un utente può registrarsi inviando la propria mail e l'username selezionato al server il quale, dopo aver controllato l'unicità dei dati forniti, genererà una password casuale, contatterà il componente di logica d'accesso ai dati per registrare il nuovo utente ed invierà all'indirizzo fornito un'email contenente le credenziali registrate. Successivamente l'utente potrà autenticarsi inserendo nel sistema le credenziali d'accesso ricevute via email.

Il server gestisce gli utenti loggati tramite il Remote Session Pattern[6], adattato a supportare una connessione bidirezionale:



(UML class diagram del Reverse Remote Session Pattern. Il blocco "Common" contiene classi presenti sia sul client che sul server)

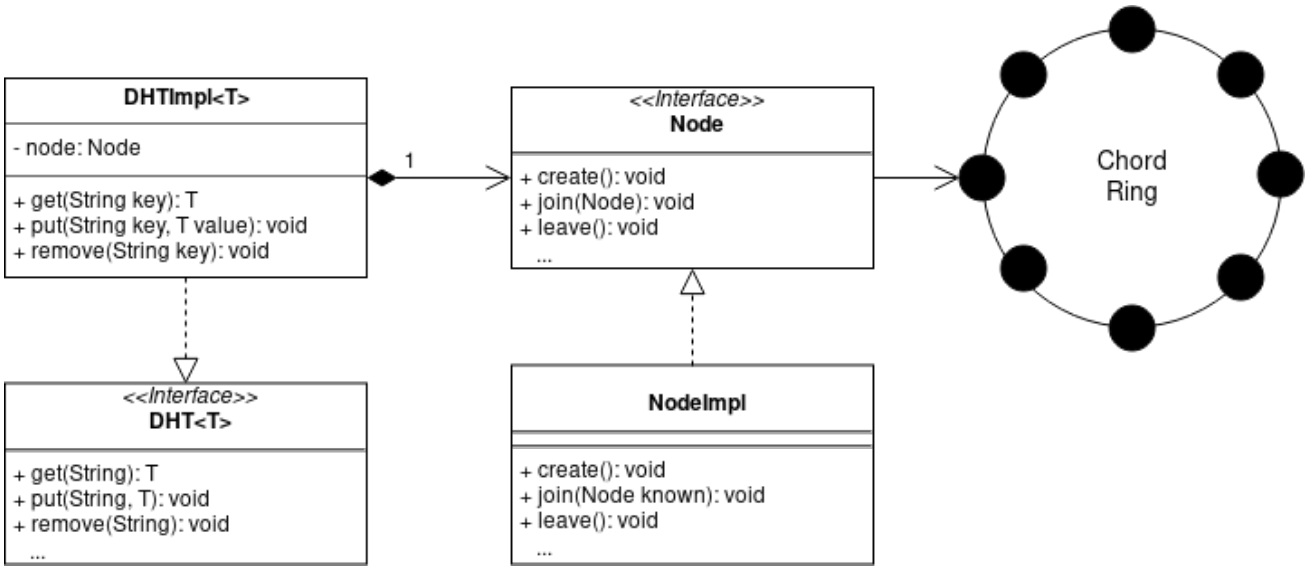
all'avvio del server viene esposta nel suo registro RMI l'implementazione dell'interfaccia remota SessionFactory. L'utente interessato a loggarsi può recuperare la SessionFactory dal RMI Registry sul server e chiamare il metodo remoto *login(String username, String password)*. Nel caso in cui le credenziali fossero valide, il metodo ritorna un'implementazione dell'interfaccia Session, tramite la quale il client potrà invocare metodi sul server. Per attivare una connessione bidirezionale (che consenta al server di invocare metodi sul client) il client, una volta ottenuta la Session, può chiamare *session.setReverseSession(ReverseSession)* e settare la propria ReverseSession nella sessione sul server.

La SessionFactory esposta sul server tiene traccia delle sessioni attive salvandole in una tabella interna(activeSessions): in questo modo potrà ricontattare i client quando necessario e allo stesso tempo tenere traccia degli utenti loggati.

Per uscire dal sistema il client può invocare il metodo session.logout() per cancellare la sessione dalle sessioni attive sul server. Nel caso in cui un client perdesse la connessione, il server può individuare l'errore tramite il metodo reverseSession.ping() e rimuovere di conseguenza la sessione scaduta.

4.2.3 Network Access Logic

E' il componente che fornisce ai controller l'accesso alla tabella distribuita. Può essere suddiviso in 2 parti principali:



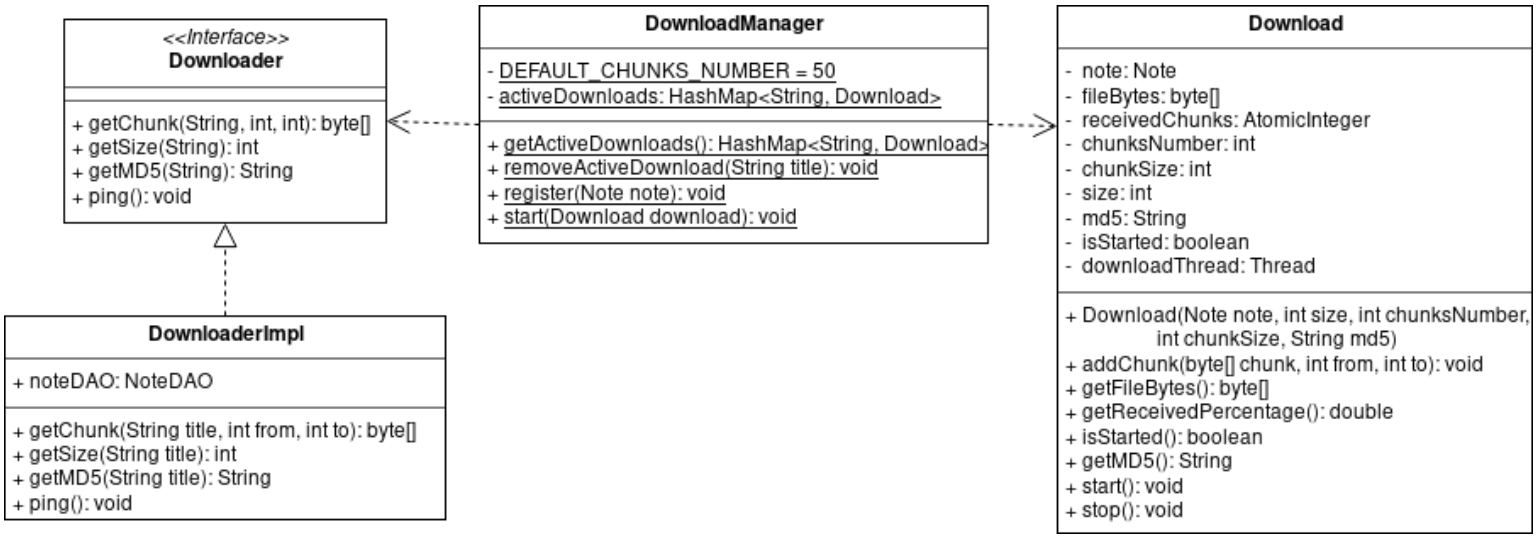
- **Node** – (costituita dall'interfaccia Node e dalla sua implementazione) è la parte dove viene effettivamente implementato l'algoritmo chord. E' l'unica parte del sistema in grado di interagire direttamente con gli altri nodi della rete p2p.
- **DHT –** (costituita anch'essa da interfaccia DHT<T> e rispettiva implementazione) incapsula l'algoritmo p2p e fornisce all'utente un'interfaccia simile a quella di una normale struttura dati per interagire con la tabella distribuita.

In questo caso è stato usato il pattern Strategy[7] per consentire alla classe DHTImpl di operare tramite il Nodo.

4.2.4 Download Logic

E' la componente che gestisce il trasferimento dei file da un peer all'altro.

Ogni peer espone nel proprio RMI Registry un Downloader, ovvero un oggetto in grado di leggere informazioni relative ad un appunto disponibile per la condivisione. La classe Download rappresenta un il trasferimento di un file inizializzato con le informazioni necessarie e con i metodi start() e stop() in grado di avviare il thread che effettivamente scaricherà il file (si veda la Process View per una spiegazione dettagliata del processo di download).



Il DownloadManager è una classe statica che fornisce un meccanismo di gestione per i Download attivi e la procedura stessa di download. L'utilizzo di questa classe non era necessario in quanto le stesse attività potevano essere svolte direttamente dal componente dei Controller. Tuttavia, trattandosi di operazioni corpose, sono state spostate in una classe separata per "alleggerire" i controller. La staticità della classe è necessaria per unificare la gestione dei download attivi e renderla thread safe.

5 Process View

In questa sezione vengono descritti gli aspetti dinamici dell'applicazione, con focus principale sui thread di esecuzione.

5.1 Server

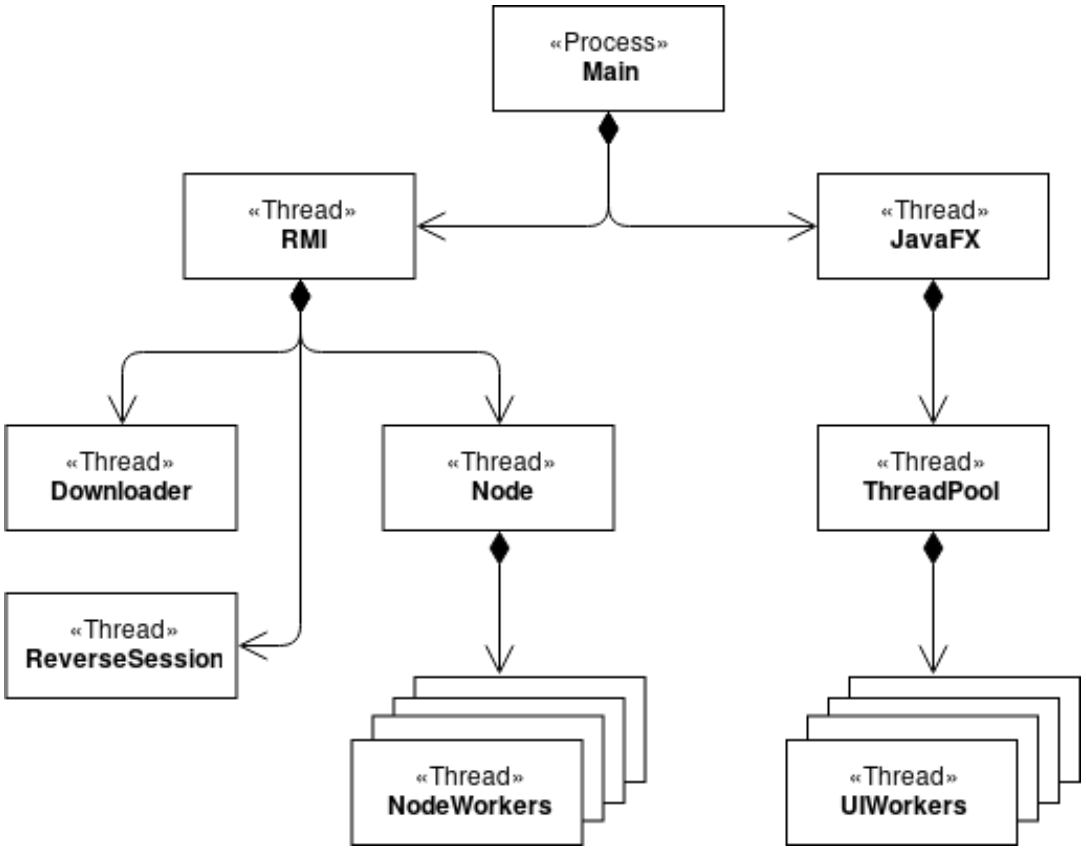
Sul server l'unico thread principale è il Request Handler di RMI. La concorrenza nelle gestione delle richieste viene affidata completamente ad RMI: vi è soltanto la garanzia che non tutte le richieste verranno gestite da un unico thread.

5.2 Client

Sul client la gestione dei thread viene effetuata in 2 punti:

- da RMI, che gestisce le richieste provenienti dalla rete p2p e dalla ReverseSession del server;
- dal thread principale di JavaFX, che crea nuovi thread per svolgere i task più complicati che altrimenti bloccherebbero l'interfaccia grafica.

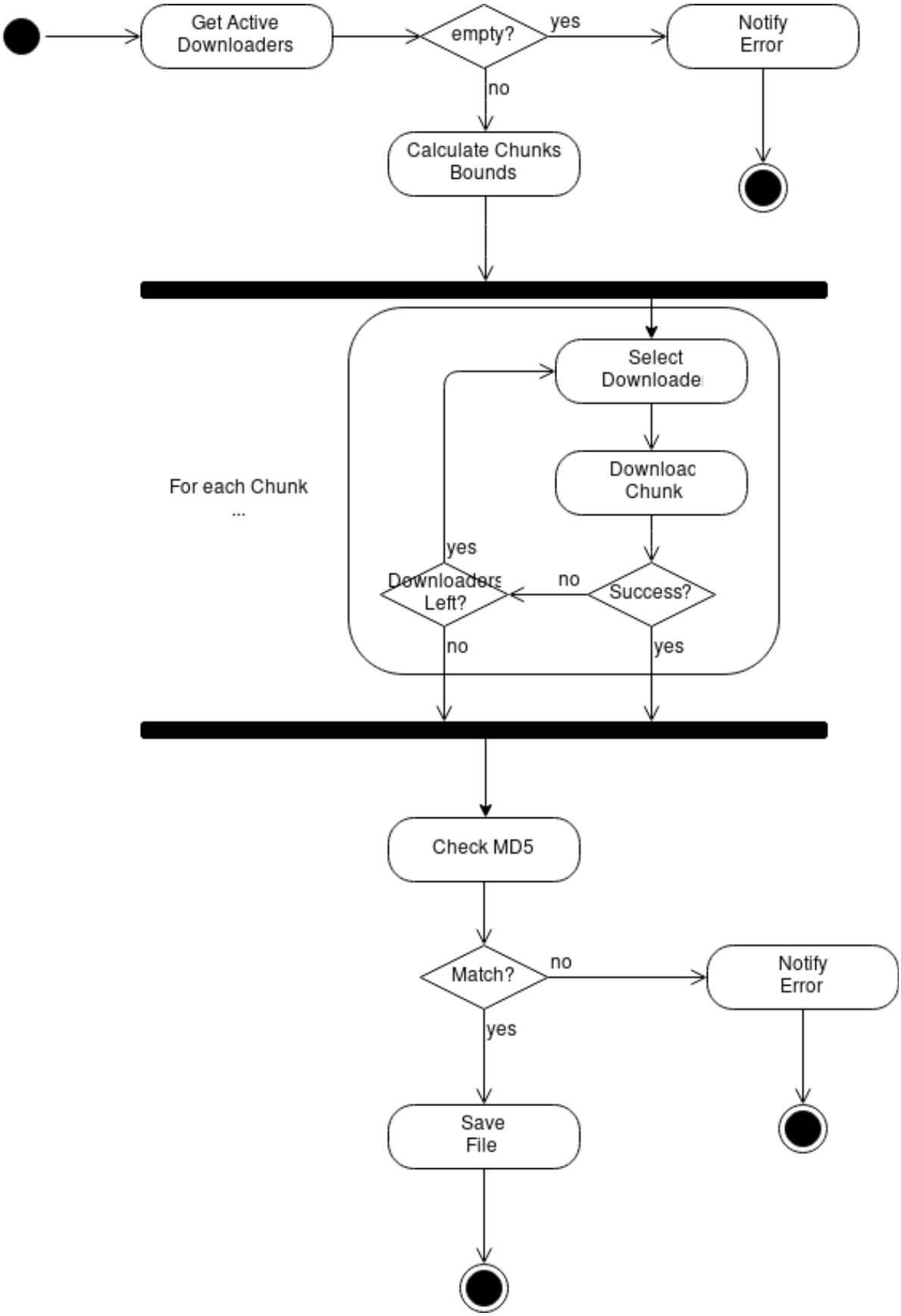
Come per il server, RMI gestisce internamente la concorrenza lasciando soltanto l'assunzione che le richieste non verranno gestite sempre dallo stesso thread. Il thread di JavaFX si appoggia invece ad una ThreadPool per l'esecuzione dei task più complicati che altrimenti bloccherebbero l'interfaccia.



(Schema di Process View del client)

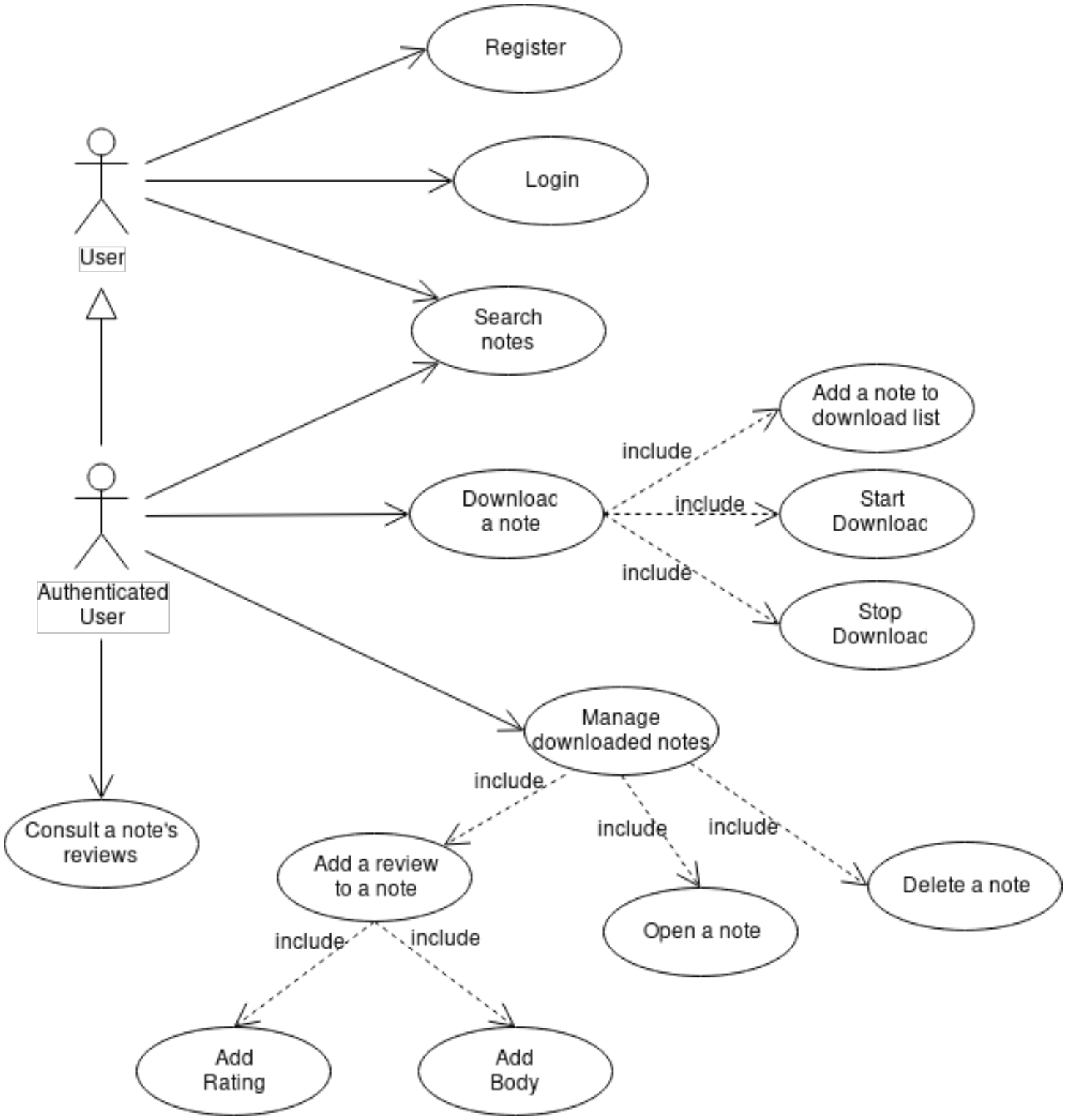
5.3 Download Thread

Particolare attenzione richiede il thread di esecuzione del download. Questo infatti genera un thread figlio per ogni segmento da scaricare, ne aspetta la terminazione e procede di conseguenza a salvare il file. Nella pagina seguente viene riportato l'activity diagram corrispondente.


Get Active Downloaders
empty?
yes
Notify Error
no
Calculate Chunks Bounds
For each Chunk
...
Select Downloade
Downloac Chunk
yes
Downloaders Left?
no
Success?
no
yes
Check MD5
Match?
no
Notify Error
yes
Save File

6 Use Case View

In questa sezione viene mostrato il diagramma del casi d’uso dell’applicazione.



7 Test

In questa sezione verranno documentati i test dell'applicazione. Sono stati testati soltanto i componenti rilevanti per un corretto funzionamento dell'applicazione e in alcuni casi è stato necessario realizzare Mock di componenti esterni.

7.1 Server

Titolo	RandomString nextString
Funzionalità	Generazione di una stringa random.
Obiettivo	Verificare la corretta generazione di una stringa casuale, data inizialmente la lunghezza.
Dati	Lunghezza = 10
Azioni	1. istanzio RandomString con la lunghezza data 2. genero una nuova stringa
Risultato atteso	Viene generata una stringa non nulla della lunghezza specificata inizialmente

Titolo	Mailer testSendCredentials
Funzionalità	Invio della mail di conferma della registrazione, contenente le credenziali di accesso.
Obiettivo	Verificare il corretto invio della mail
Dati	Mock del server SMTP tramite GreenMail username = "username" password = "password" email = "prova@mail.com"
Azioni	1. invio la mail 2. recupero la mail ricevuta dal server GreenMail
Risultato atteso	La mail ricevuta avrà destinatario, oggetto e corpo corrispondenti a quelli specificati nella classe Mailer.

Titolo	UserDAO create
Funzionalità	Salvataggio nel DB di un nuovo utente.
Obiettivo	Verificare il corretto salvataggio di un nuovo utente.
Dati	username = "username" password = "password" email = "email"

Azioni	1. creo l’utente 2. leggo le credenziali dal database
Risultato atteso	Il risultato della lettura coinciderà con i dati forniti inizialmente

Titolo	UserDAO `createFailsIfUsernameIsAlreadyTaken`
Funzionalità	Salvataggio nel DB di un nuovo utente.
Obiettivo	Verificare l’impossibilità di salvare nel database un nuovo utente con un username già “occupato”.
Dati	User1: (“username”, “password”, “email”) User2: (“username”, “password2”, “email2”)
Azioni	1. creo l’utente 1 2. creo l’utente 2
Risultato atteso	Viene sollevata una AddFailedException alla creazione dell’utente 2

Titolo	UserDAO `createFailsIfMailIsAlreadyTaken`
Funzionalità	Salvataggio nel DB di un nuovo utente.
Obiettivo	Verificare l’impossibilità di salvare nel database un nuovo utente con una mail già “occupata”.
Dati	User1: (“username”, “password”, “email”) User2: (“username2”, “password2”, “email”)
Azioni	1. creo l’utente 1 2. creo l’utente 2
Risultato atteso	Viene sollevata una AddFailedException alla creazione dell’utente 2

Titolo	UserDAO `delete`
Funzionalità	Eliminazione di un utente dal DB.
Obiettivo	Verificare la corretta eliminazione di un utente dal database.
Dati	User1: (“username”, “password”, “email”)
Azioni	1. creo l’utente “username” 2. elimino l’utente “username” 3. leggo l’utente “username”
Risultato atteso	L’ultima lettura ritorna null.

Titolo	DHT `putAndGet`
Funzionalità	Lettura e aggiunta di dati nella tabella distribuita
Obiettivo	Verificare la corretta aggiunta e lettura di dati nella tabella distribuita.
Dati	key = “toGet” value = “get” notExistingKey = “dummy”
Azioni	1. con il nodo1 effettuo una put di (key, value) 2. con ognuno dei 3 nodi effettuo una get(key) 3. con ognuno dei 3 nodi effettuo una get(notExistingKey)
Risultato atteso	Le get del punto 2 ritornano tutte value mentre le get del punto 3 ritornano tutte null.

Titolo	DHT `putReplaceOldValue`
Funzionalità	Lettura e aggiunta di dati nella tabella distribuita
Obiettivo	Verificare che la put di una chiave già mappata sostituisce il vecchio valore con il nuovo.
Dati	key = “chiave” value = “valore” value1 = “valore1”
Azioni	(azioni svolte con uno qualsiasi dei nodi) 1. put di (key, value) 2. get(key) 3. put di (key, value1) 4. get(key)
Risultato atteso	La get del punto 2 ritorna value, quella del punto 3 ritorna value1

Titolo	DHT `putAndRemove`
Funzionalità	Eliminazione di dati dalla tabella distribuita.
Obiettivo	Verificare il corretto funzionamento della remove.
Dati	key = “toRemove” value = “remove”
Azioni	(per ogni nodo) 1. put di (key, value) 2. get(key)
Risultato atteso	La get del punto 2 ritorna sempre null per ogno nodo

Titolo	DHT `query`
Funzionalità	Query di dati nella tabella
Obiettivo	Verificare il corretto funzionamento di una query.
Dati	key1 = “chiave1” value1 = “abc” key2 = “chiave2” value2 = “bcd” predicate = //il valore inizia per ‘a’
Azioni	(azioni svolte con uno qualsiasi dei nodi) 1. put di (key1, value1) 2. put di (key2, value2) 3. query(predicate)
Risultato atteso	La query ritorna solo value1.

Titolo	DHT `exec`
Funzionalità	Eseguire operazioni arbitrarie su un elemento della tabella distribuita
Obiettivo	Verificare il corretto funzionamento dell’esecuzione di operazioni arbitrarie
Dati	key = “exec” value = new StringBuffer(“min”) operation = //toUpper
Azioni	(azioni svolte con uno qualsiasi dei nodi) 1. put di (key, value) 2. exec(operation) 3. get(key)
Risultato atteso	La get ritorna “MIN”

Titolo	DHT `testFaultTolerance`
Funzionalità	Tolleranza ai fallimenti arbitrari
Obiettivo	Verificare la tolleranza ai fallimenti arbitrari.
Dati	50 coppie chiave valore (“i”, i) dove 0<= i < 50 predicato = //get All
Azioni	(azioni 3-4-5-6 ripetute più volte, in ordine sparso e con diversi nodi) 1. put di tutte le coppie (“i”, i) 2. getAll 3. kill di un nodo 4. getAll 5. join di un nodo

	6. getAll
Risultato atteso	Tutte le getAll (2, 4, 6) ritornano sempre tutte le 50 coppie (“i”, i)

7.2 Client

Titolo	FixedSizeQueue `testBasicQueue`
Funzionalità	Basilare funzionamento della coda
Obiettivo	Verificare il corretto funzionamento della coda.
Dati	Una FixedSizeQueue di interi vuota e lunga 20
Azioni	1. aggiungo valori da 1 a 20 2. leggo tutti i valori dalla coda
Risultato atteso	Al punto 2 ottengo tutti i valori da 1 a 20 nello stesso ordine di inserimento (FIFO).

Titolo	FixedSizeQueue `testExceedingSize`
Funzionalità	Viene sforata la dimensione massima della coda
Obiettivo	Verificare che, allo sforare della dimensione massima della coda, gli elementi più “vecchi” vengono eliminati.
Dati	Una FixedSizeQueue di interi vuota e lunga 10
Azioni	1. aggiungo valori da 1 a 20 2. leggo tutti i valori dalla coda
Risultato atteso	Al punto 2 ottengo tutti i valori da 11 a 20 nello stesso ordine di inserimento (FIFO).

Titolo	NoteDAO create
Funzionalità	Salvataggio nel DB di un nuovo appunto.
Obiettivo	Verificare il corretto salvataggio di un nuovo appunto.
Dati	title = “title” path = “path”
Azioni	1. creo l’appunto 2. leggo l’appunto “title” dal database
Risultato atteso	Il risultato della lettura coinciderà con i dati forniti inizialmente

Titolo	NoteDAO `createFailsIfTitleIsAlreadyPresent`
Funzionalità	Salvataggio nel DB di un nuovo appunto.
Obiettivo	Verificare l'impossibilità di salvare nel database un nuovo appunto con un titolo già "occupato".
Dati	note1: ("title", "path") note2: ("title", "path2")
Azioni	1. creo note1 2. creo note2
Risultato atteso	Viene sollevata una AddFailedException alla creazione dell'appunto 2.

Titolo	NoteDAO getAll
Funzionalità	Lettura dal DB di tutti gli appunti.
Obiettivo	Verificare la corretta lettura di tutti gli appunti.
Dati	Note(titolo: "getAll" + i, path: "path") con 0 <= i < 20
Azioni	1. creo tutti gli appunti 2. leggo tutti gli appunti
Risultato atteso	Il risultato della lettura coinciderà con i dati forniti inizialmente

Titolo	NoteDAO getAllReturnsEmptyCollection
Funzionalità	Lettura dal DB di tutti gli appunti.
Obiettivo	Se non sono presenti appunti il tentativo di lettura ritorna una lista vuota.
Dati	
Azioni	1. leggo tutti gli appunti
Risultato atteso	Il risultato della lettura sarà una lista vuota.

Titolo	NoteDAO readReturnsNullIfNothingIsFound
Funzionalità	Lettura dal DB di tutti un appunto.
Obiettivo	Se l'appunto non è presente il tentativo di lettura ritorna null.
Dati	
Azioni	1. leggo l'appunto "dummy"
Risultato atteso	Il risultato della lettura sarà null.

Titolo	NoteDAO update
Funzionalità	Aggiornamento dei dati relativi ad un appunto.
Obiettivo	Verificare il corretto funzionamento dell'update
Dati	note("titolo", "path1") noteU("titolo", "path2")
Azioni	1. creo note 2. update con noteU 3. leggo la nota "titolo"
Risultato atteso	La nota letta al punto 3 avrà path uguale a "path2"

Titolo	NoteDAO updateFailsIfNoElementIsFound
Funzionalità	Aggiornamento dei dati relativi ad un appunto.
Obiettivo	Se l'appunto non è presente il tentativo di aggiornamento fallisce.
Dati	note("updatable", "path")
Azioni	1. update note
Risultato atteso	Viene sollevata una UpdateFailedException

Titolo	NoteDAO delete
Funzionalità	Cancellazione di un appunto.
Obiettivo	Verificare il corretto funzionamento della cancellazione.
Dati	note("deletable", "path")
Azioni	1. crea note 2. delete note 3. leggi l'appunto "deletable"
Risultato atteso	La lettura al punto 3 ritorna null.

Titolo	DownloadManager register
Funzionalità	Registrazione di un nuovo download.
Obiettivo	Verificare la registrazione di un nuovo download.
Dati	noteMetaData("titolo", "autore", "corso", "prof", 2018) note(noteMetaData, "titolo", "path") downloader = //Mock di un downloader remoto
Azioni	1. aggiunti downloader agli owner di note 2. registra note

	3. leggi download attivi
Risultato atteso	La lettura al punto 2 ritorna un solo download corrispondente con la nota appena registrata.

Titolo	DownloadManager registerFailsIfNoDownloaderIsFound
Funzionalità	Registrazione di un nuovo download.
Obiettivo	Se una nota non ha possessori attivi, la registrazione fallisce.
Dati	noteMetaData(“titolo”, “autore”, “corso”, “prof”, 2018) note(noteMetaData, “titolo”, “path”)
Azioni	1. registra note
Risultato atteso	Viene sollevata un’eccezione.

Titolo	DownloadManager downloadFile
Funzionalità	Scaricamento di un file da più peer.
Obiettivo	Verificare il download di un file remoto da più peer.
Dati	noteMetaData(“titolo”, “autore”, “corso”, “prof”, 2018) note(noteMetaData, “titolo”, “titolo.pdf”) downloader1, downloader2, downloader3 = //Mock Downloader
Azioni	1. registra downloader1, 2, 3 come owner della nota 2. registra la nota tra i download attivi 3. scarica il file
Risultato atteso	Il file scaricato ha lo stesso MD5 fornito dai downloader e ciascuno dei downloader stessi ha inviato almeno un chunk. La somma totale dei chunk inviati dai downloader sarà ugule alla dimensione in chunk del file.

8 Bibliografia

1. Kruchten, P. *“The 4+1 View Model of architecture”* // IEEE Software (1995) vol. 12, iss. 6 pp. 42 -50 .
2. *Stoica, I.; Morris, R.; Karger, D.; Kaashoek, M. F.; Balakrishnan, H. (2001). "Chord: A scalable peer-to-peer lookup service for internet applications"*
3. Pamela Zave, “*How to Make Chord Correct*”. AT&T Laboratories (2015)
4. Sameh El-Ansary, Luc Onana Alima, Per Brand, Seif Haridi, *“Efficient Broadcast in Structured P2P Networks”*.
5. *"Core J2EE Patterns - Data Access Objects"*. Sun Microsystems Inc. 2007-08-02.
6. K. McNiff and E. Pitt, *java.rmi: “The Remote Method Invocation Guide”*, *Addison-Wesley Professional (2001).*
7. *Gamma, E., Helm, R., Johnson, R. e Vlissides, J., “Design Patterns: Elements of Reusable Object-Oriented Software”, Addison Wesley, 1995, ISBN 0-201-63361-2*