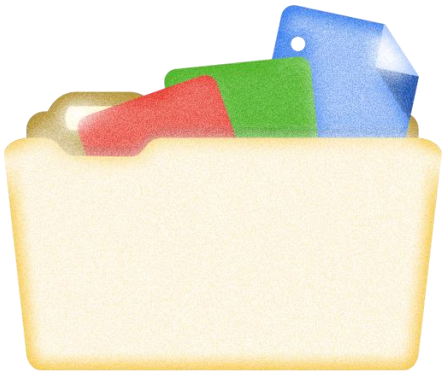


UNISHARE

Documentazione

Michele Scuttari
[Politecnico di Milano – Ingegneria informatica]

Indice

Indice **1**
1 Introduzione **2**
2 Tecnologie **2**
2.1 Client 2
2.2 Server 2
3 Installazione **3**
4 Kademlia **4**
4.1 Descrizione 4
4.2 Tabella di indirizzamento 4
4.3 Messaggi del protocollo 4
4.4 Store 5
4.5 Node lookup 5
4.6 Connessione alla rete (bootstrap) 6
4.7 Trasferimento file 6
4.8 Implementazione 7
5 Server **9**
5.1 Gestione delle credenziali 9
5.2 Registrazione 10
5.3 Login 10
6 Client **11**
6.1 Database 11
6.2 Rappresentazione dei dati 12
6.3 Managers 13

1 Introduzione

UniShare è un software che consente agli studenti universitari lo scambio di appunti delle lezioni.

Il sistema consente agli utenti generici (non autenticati) di prendere visione del catalogo degli appunti disponibili, ricercabili per nome dell'università, del dipartimento, del corso e del docente titolare dell'insegnamento.

In aggiunta alle funzionalità disponibili agli utenti generici, il sistema consente agli utenti autenticati di effettuare le seguenti azioni:

- Condividere i propri appunti (file in formato PDF con peso $\leq$ 10 MB)
- Ottenere il file relativo agli appunti trovati attraverso la ricerca
- Visualizzare le recensioni degli appunti
- Pubblicare una propria recensione relativa agli appunti scaricati

Inoltre, gli appunti sono memorizzati sulle macchine degli utenti e trasferiti da un utente all'altro in modalità peer-to-peer, lasciando quindi al server solamente l'autenticazione degli utenti. Per realizzare quanto detto è stata realizzata una implementazione di Kademlia, un protocollo di rete peer-to-peer ideato da Petar Maymounkov e David Mazières della New York University; il suo funzionamento verrà trattato dettagliatamente in seguito.

2 Tecnologie

L'intero progetto e le dipendenze sono stati gestiti attraverso **Gradle**, mentre i test sono stati realizzati usando **JUnit4**.

In seguito, si riportano i dettagli relativi agli specifici strumenti utilizzati per client e server:

2.1 Client

- La parte grafica è realizzata con **JavaFX**, con l'ulteriore ausilio delle seguenti librerie:
  - **ControlsFX**: utilizzata per l'implementazione alcune funzionalità come popup e schermate a scorrimento.
  - **FontAwesomeFX**: utilizzata per l'inserimento delle icone del font FontAwesome.
  - **PDF-Renderer** di Swing: utilizzata per l'anteprima, interna all'applicazione, dei propri file condivisi
- È stato utilizzato **SQLite3** come DBMS, in quanto ha permesso di salvare rapidamente le varie informazioni in un unico file, posizionato nella cartella dei dati specificata nel programma.
- La comunicazione con il server viene effettuata tramite chiamate **RMI**.
- Sono state utilizzate le seguenti librerie per quanto riguarda la parte core:
  - **Guava**: utilizzata in quanto già dotata di Executors mantenibili anche in seguito alla chiusura dell'applicazione (utile per i download non ancora terminati).
  - **Apache commons IO**: utilizzata per la copia dei file tramite rete senza dover gestire manualmente i buffer di trasferimento.

2.2 Server

- È stato utilizzato **MySQL** come DBMS e **Hibernate** come ORM.
- Per l'invio delle mail di registrazione è stata utilizzata la libreria **JavaMail**.

3 Installazione

Sono stati realizzati due file JAR, relativi a client e server.

- Il client è avviabile senza prerequisiti (a parte ovviamente l'installazione di Java) tramite il comando
  `java -jar client.jar`
- Il server richiede invece un'installazione di MySQL e di un database vuoto chiamato "unishare", che viene acceduto tramite l'utente "unishare" avente password "unishare". Analogamente al client, il server è avviabile attraverso il comando
  `java -jar server.jar`

4 Kademlia

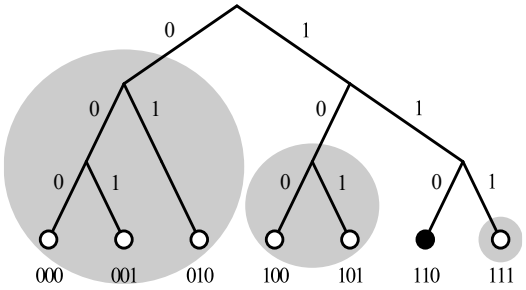
4.1 Descrizione

Kademlia si basa sulla tecnologia **Distributed Hash Table** (DHT). Su una rete già esistente viene creato un nuovo network virtuale in cui ogni nodo è identificato da un numero identificativo di 160 bit. Su tale ID viene definito un concetto di distanza utilizzando l'operazione XOR, in quanto simmetrica e in grado di far valere la disuguaglianza triangolare. Ovviamente la distanza così calcolata non ha nulla a che vedere con la distanza geografica e due nodi che risultano vicini per l'algoritmo potrebbero essere fisicamente localizzati anche in due continenti differenti.

4.2 Tabella di indirizzamento

La tabella di indirizzamento di Kademlia consiste di una lista di grandezza pari alla lunghezza in bit dell'ID de nodo. Ogni elemento della lista è a sua volta una lista chiamata **bucket**.

Ogni bucket può contenere al massimo **k** informazioni e ognuna di queste è una terna di valori (chiamata NND: "network node data) relativa a un altro nodo della rete: ID, indirizzo IP e porta di connessione. Ogni bucket corrisponde inoltre ad una specifica distanza dal nodo. I nodi che finiscono nell'ennesimo bucket devono avere i primi n-1 bit uguali a quelli del nodo stesso e differire almeno dall'ennesimo bit in poi.



Esempio di partizionamento visto dal nodo 110

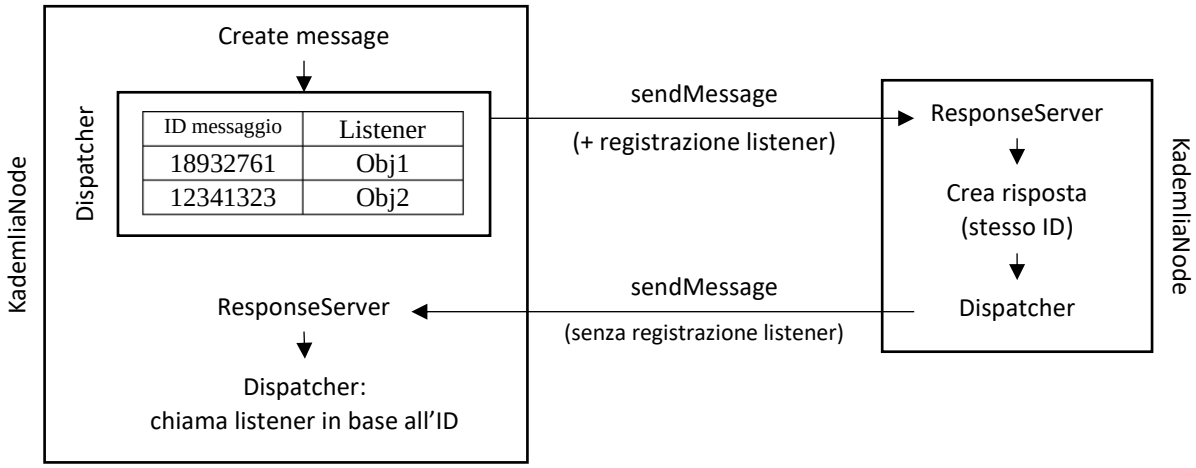
Quando si incontrano nodi nella rete Kademlia, questi sono aggiunti alle liste della tabella di indirizzamento. Quando un bucket è pieno ed è scoperto un nuovo nodo destinato a quello specifico bucket, il nodo visto meno recentemente viene pingato: se il nodo è ancora vivo, il nuovo nodo è messo in una lista di rimpiazzo. Tale lista di rimpiazzo è utilizzata solo se un nodo del bucket smette di rispondere. In altre parole, i nuovi nodi sono utilizzati solo quando i vecchi nodi smettono di rispondere; ciò favorisce i nodi connessi da maggior tempo e aumenta la stabilità della rete annullando eventuali attacchi DOS.

4.3 Messaggi del protocollo

Kademlia prevede quattro tipi di messaggi:

- **PING**: utilizzato per verificare che un nodo sia vivo.
  ```
  void ping(NND node);
  ```
- **STORE**: utilizzato per archiviare una coppia (chiave, valore) in un nodo.
  ```
  void storeFile(F file);
  ```
- **FIND_NODE**: utilizzato per cercare nodi; il nodo che riceve il messaggio fornisce al mittente i k nodi più vicini all'ID ricercato.
  ```
  void lookup(NodeId targetId, LookupListener endProcessListener);
  ```
- **FIND_VALUE**: come FIND_NODE, ma se il ricevente dispone della chiave richiesta, al posto dei k nodi fornisce il valore corrispondente alla chiave. In questa versione semplificata di Kademlia questo messaggio è implementato con il nome FIND_DATA e comporta un funzionamento leggermente diverso: il messaggio contiene un filtro di ricerca e il nodo che riceve il messaggio risponde con un elenco dei file conosciuti che rispettano i criteri indicati.
  ```
  void searchData(FD filter, SearchListener<F, FD> listener);
  ```

Ogni messaggio include un ID random, dello stesso tipo di quello dei nodi, generato dal mittente. Tale valore è utilizzato dal Dispather per assicurare che la risposta sia relativa alla domanda effettuata.



Inoltre, ad ogni messaggio inviato è associato un timeout fisso di 10 secondi, dopo il quale il messaggio viene considerato perso e si provvede a pingare il nodo per verificare se è ancora online. In caso di mancata risposta, il nodo viene rimosso dalla tabella di routing.

4.4 Store

Ogni dato non è un file in senso stretto bensì un insieme di dati in grado di rappresentarlo all'interno della rete. Tale insieme di dati è costituito da:

- ID: dello stesso tipo dell'ID dei nodi; è ottenuto generando l'impronta SHA1 del contenuto del file vero e proprio.
- Proprietario: ID, indirizzo IP e porta del nodo che è in possesso del file vero e proprio.
- Metadati: campi personalizzabili utilizzati per riassumere le principali informazioni del file e utilizzabili durante la ricerca.

Quando un nodo desidera memorizzare nella sua memoria un file da egli stesso posseduto, provvede alla sua ulteriore pubblicazione presso i k nodi più vicini all'ID del file, in modo da aumentare la raggiungibilità del dato all'interno della rete. Tale pubblicazione è ripetuta ogni ora fintanto che il nodo è online.

Quando si desidera invece tenere traccia dell'esistenza di file altrui (procedura effettuata alla ricezione di un messaggio STORE), i dati del file vengono semplicemente salvati all'interno della memoria e viene creato un timer che provvede a scartare l'informazione dopo un'ora, in modo da eliminare automaticamente dalla rete i file non più realmente ottenibili.

4.5 Node lookup

```
void lookup(NodeId targetId);
```

La ricerca dei nodi procede asincronamente e la quantità di ricerche simultanee è denotato da α (tipicamente pari a tre). Un nodo inizia una ricerca inviando una richiesta FIND_NODE ai nodi contenuti nel bucket più vicino all'ID da trovare. Quando i nodi riceventi ricevono il messaggio, ricercano nei propri bucket e ritornano i k nodi più vicini al nodo desiderato. Sulla base delle risposte ricevute il richiedente aggiorna la propria lista delle risposte e reitera la richiesta verso gli α nodi più vicini che non ha ancora contattato. L'algoritmo termina quando il richiedente ha contattato e ottenuto risposta dai k nodi più vicini all'ID cercato.

4.6 Connessione alla rete (bootstrap)

```
void bootstrap(NND bootstrapNode);
```

Un nodo può collegarsi alla rete di Kademlia solo se ne conosce già un altro che ne fa parte. Tale nodo è il server, che rappresenta di fatto il primo nodo che crea la rete e che permette agli altri di unirsi. Il server rappresenta quindi un single point of failure del sistema, ma tale limite sarebbe comunque superabile se si avessero ulteriori macchine con IP fisso da usare come nodi multipli di bootstrap.

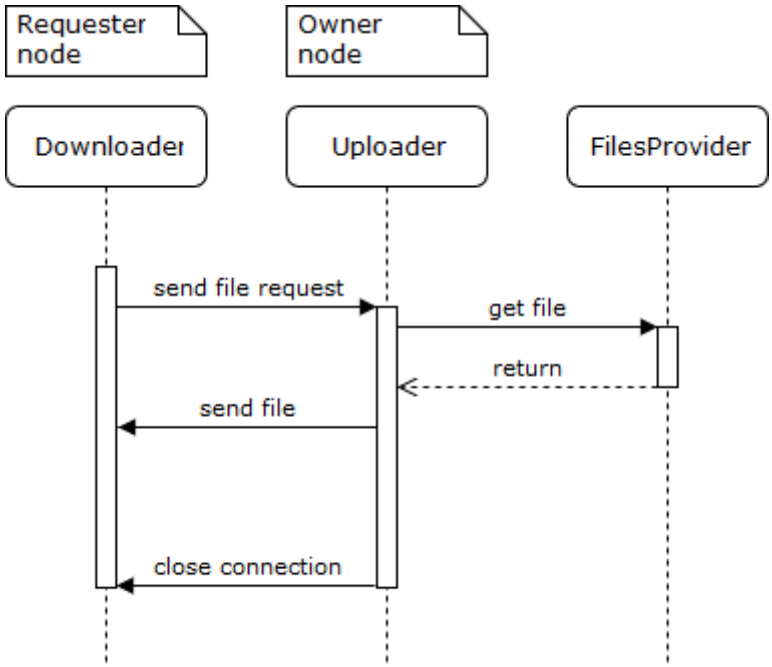
4.7 Trasferimento file

Ogni nodo della rete provvede a creare un server TCP che ascolta sulla porta avente lo stesso numero di quella utilizzata per i messaggi, scambiati invece tramite UDP. Questa configurazione è stata scelta al fine di garantire un trasferimento affidabile dei file (in quanto caratteristica intrinseca di TCP), requisito invece non necessario per la semplice comunicazione tra i nodi.

Il trasferimento dei file è gestito attraverso le due classi Uploader e Downloader, rispettivamente utilizzate dal nodo proprietario e dal nodo richiedente il file. Ogni Uploader e Downloader vengono eseguiti in un thread dedicato e in grado di proseguire anche in seguito a un'eventuale chiusura dell'applicazione.

Ogni Uploader utilizza un FilesProvider, assegnato direttamente al nodo, in grado di fornire al thread di upload il file vero e proprio da trasferire a fronte di una iniziale conoscenza dei soli metadati del file. In questo modo è possibile mantenere una separazione tra la logica della rete e quella di accesso ai file.

Di seguito viene riportato il diagramma di iterazione tra i vari attori del processo di trasferimento di un file:



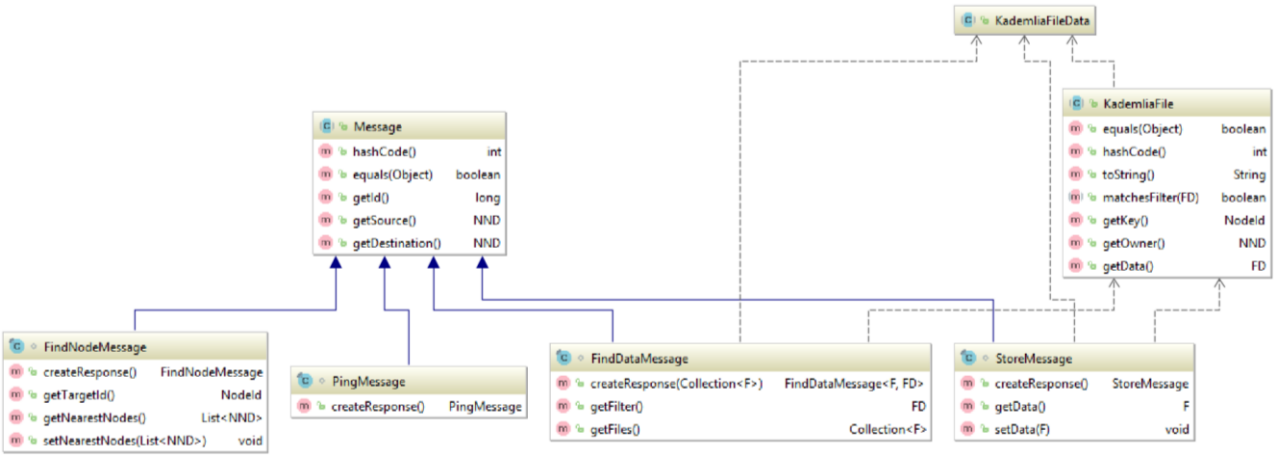
4.8 Implementazione

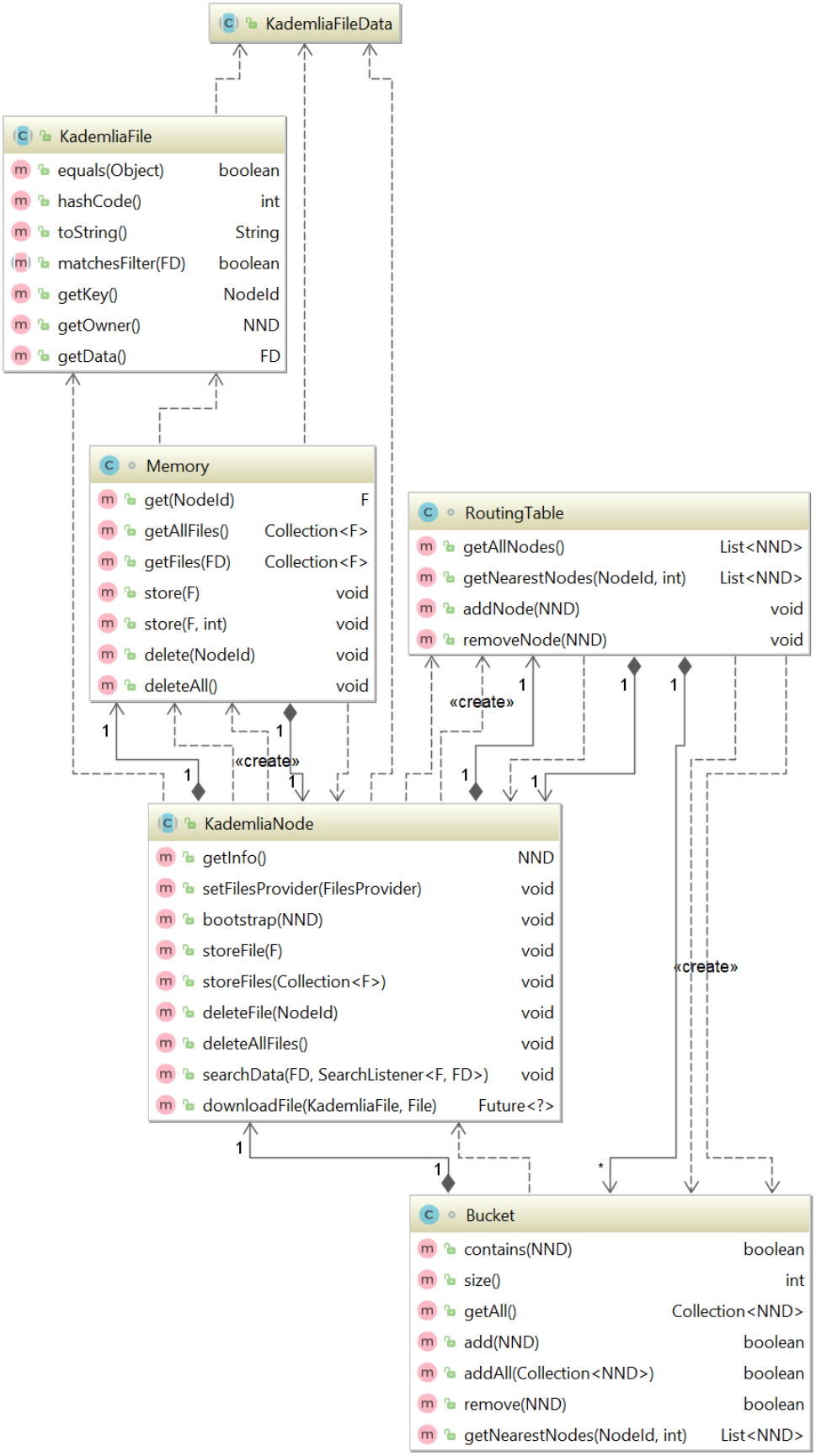
L'implementazione realizzata prevede l'estensione di tre classi di "base", le quali, come verrà spiegato nella parte relativa al client, verranno poi rese specifiche per il contesto dell'applicazione. Lo scopo di tale struttura è rendere indipendente la logica di funzionamento della rete da eventuali personalizzazioni volte allo specifico contesto applicativo.

- **KademliaNode**: rappresenta il nodo della rete e fornisce la logica di base per la gestione della rete, a partire dal semplice ping di un nodo fino alla procedura di node lookup e di ricerca di un file. Una volta estesa, è possibile ricevere messaggi personalizzati attraverso il metodo *onMessageReceived*.
- **KademliaFile (F)**: rappresenta un file della rete e contiene al suo interno le informazioni relative a:
    - ID del file (dello stesso formato dell'ID del nodo).
    - Proprietario: ID, indirizzo IP e porta.
    - Metadati configurabili a seconda delle informazioni da condividere in rete.
- **KademliaFileData (FD)**: metadati di un file

A queste si aggiungono altre classi, utilizzate esclusivamente nella parte "core" della rete, e quindi non visibili al client. Alcuni esempi sono la tabella di routing e la memoria di un nodo, ma anche i messaggi di Ping o di node lookup.

Di seguito sono riportati i diagrammi UML della parte relativa ai messaggi e della parte core sopra descritta:




KademliaFileData
KademliaFile
equals(Object) boolean
hashCode() int
toString() String
matchesFilter(FD) boolean
getKey() NodeId
getOwner() NND
getData() FD
Memory
get(NodeId) F
getAllFiles() Collection<F>
getFiles(FD) Collection<F>
store(F) void
store(F, int) void
delete(NodeId) void
deleteAll() void
RoutingTable
getAllNodes() List<NND>
getNearestNodes(NodeId, int) List<NND>
addNode(NND) void
removeNode(NND) void
«create»
«create»
«create»
KademliaNode
getInfo() NND
setFilesProvider(FilesProvider) void
bootstrap(NND) void
storeFile(F) void
storeFiles(Collection<F>) void
deleteFile(NodeId) void
deleteAllFiles() void
searchData(FD, SearchListener<F, FD>) void
downloadFile(KademliaFile, File) Future<?>
Bucket
contains(NND) boolean
size() int
getAll() Collection<NND>
add(NND) boolean
addAll(Collection<NND>) boolean
remove(NND) boolean
getNearestNodes(NodeId, int) List<NND>
1
*

5 Server

Il server ha il solo scopo di autenticare gli utenti e fornire un punto di accesso per la rete Kademlia. Quest'ultima funzionalità è già stata descritta nel capitolo relativo alla fase di bootstrap di un nodo. Viene quindi descritto in seguito come vengono gestite le credenziali degli utenti e il funzionamento dei processi di registrazione e login.

5.1 Gestione delle credenziali

L'elenco degli utenti è contenuto in un'unica tabella del database. I campi presenti in tale tabella, e di conseguenza nella classe Utente, sono i seguenti:

- **ID**: numero identificativo dell'utente.
- **Email**: email fornita in fase di registrazione
- **Password**: password crittografata usando l'algoritmo SHA512.
- **Salt**: chiave ("sale") utilizzata durante il processo di crittografia della chiave. Questo valore è generato casualmente e assegnato in modo permanente durante la fase di creazione di un utente.
- **First name**: nome fornito in fase di registrazione.
- **Last name**: cognome fornito in fase di registrazione.

L'utilizzo di un "sale" durante la fase di hashing della password è volto ad incrementare la sicurezza dei dati, rendendo difficili gli attacchi di tipo rainbow table a cui una tabella contenente solo hashing "puri" sarebbe vulnerabile.

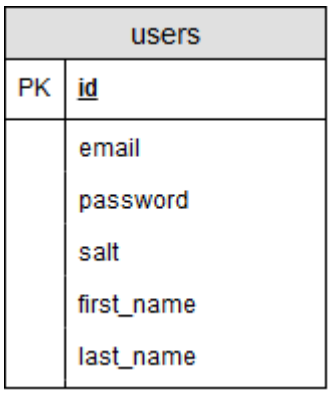
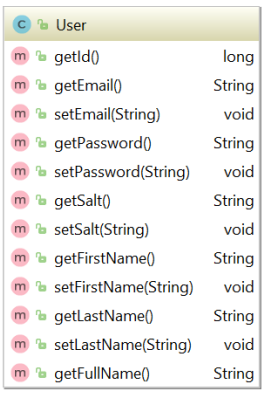


Tabella del database



Classe Java

5.2 Registrazione

```
User signup(User user);
```

1. Il client crea un oggetto User, da inviare al server, contenente i dati desiderati per il nuovo utente. La password viene lasciata in chiaro in quanto sarà compito del server generare il sale e ottenere quindi un hashing "salato" della password desiderata.
2. Una volta ricevuti i dati del nuovo utente, il server controlla la validità dei campi forniti.
3. Viene verificata la non esistenza di un utente avente già l'email specificata.
4. Viene generato il sale da applicare alla password e si calcola quindi l'hashing di quest'ultima.
5. L'utente viene salvato nel database.
6. Viene eseguito in modo asincrono l'invio dell'email di avvenuta registrazione.
7. Viene restituito al client un nuovo oggetto di tipo User, contenente i dati dell'utente ma privo di password e sale.

5.3 Login

```
User login(User user);
```

1. Il client crea un oggetto User, da inviare al server, contenente email e password in chiaro.
2. Una volta ricevuti i dati di login, il server controlla l'esistenza di un utente avente l'email specificata.
3. Viene calcolato l'hash salato della password specificata e viene confrontato con il valore della password crittografata presente nel database.
4. Se i due valori sono uguali, viene restituito al client un nuovo oggetto di tipo User, contenente i dati dell'utente ma privo di password e sale.

6 Client

6.1 Database

Come precedentemente accennato, il client fa uso di SQLite per salvare i propri dati. Tali informazioni sono:

- L'elenco dei file condivisi.
- Le recensioni ricevute relative ai file condivisi.
- L'elenco dei file scaricati.

Si riportano in seguito alcune giustificazioni sulla struttura del database, visibile attraverso il diagramma ER riportato a fianco:

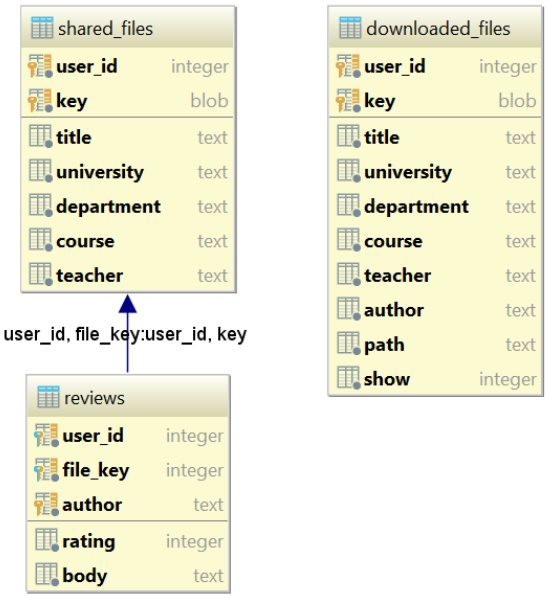


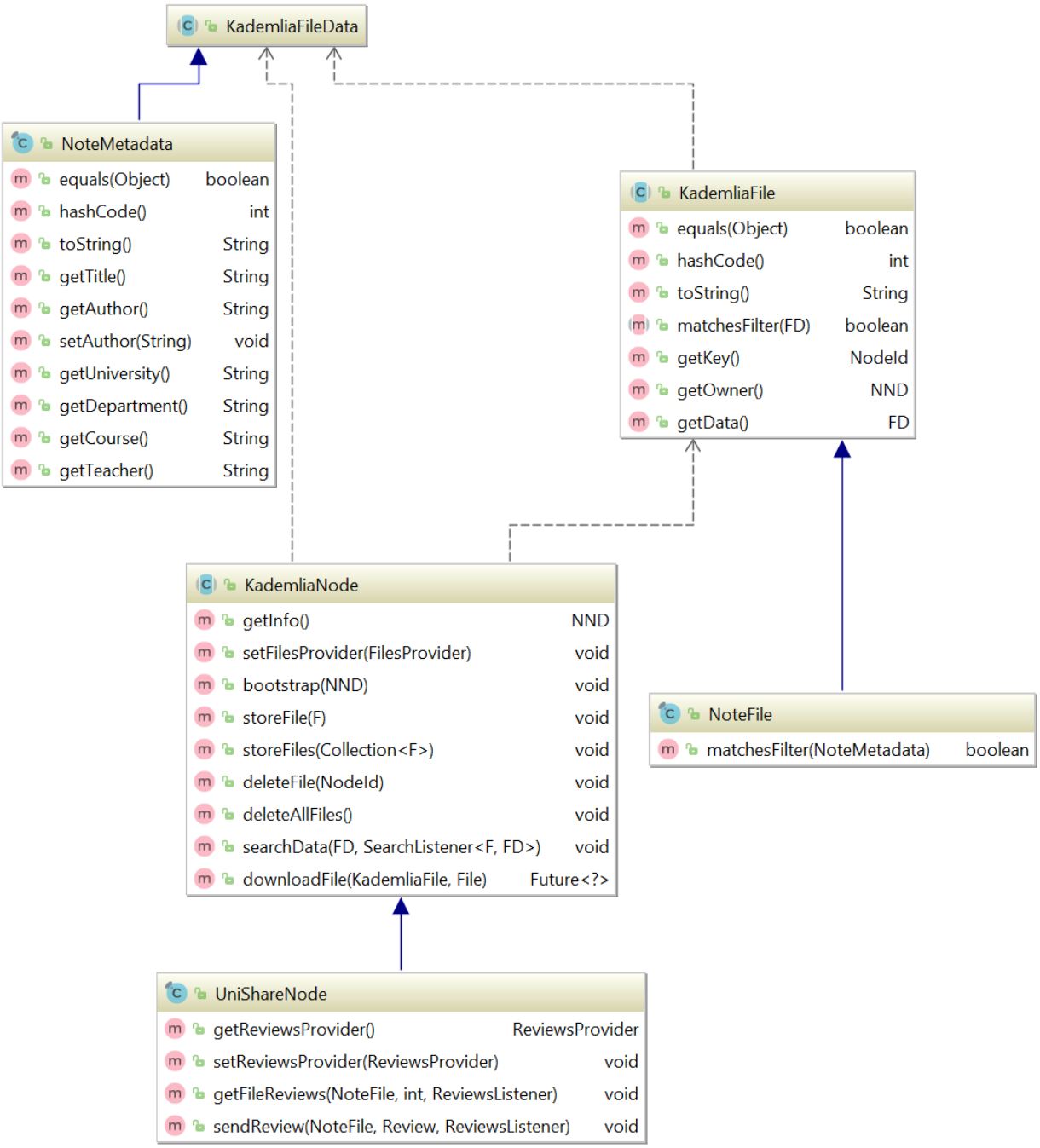
Diagramma ER

- L'utilizzo del campo **user_id** è finalizzato alla separazione delle informazioni tra i vari utenti che utilizzano l'applicazione. Due utenti diversi possono condividere lo stesso file e avere metadati diversi.
- Nell'elenco dei file condivisi non è presente alcun campo relativo al vero percorso del file, in quanto questo viene determinato automaticamente secondo il seguente formato: *Percorso**userId_key.pdf*, dove "Percorso" è il percorso della directory dei dati dell'applicazione (specificabile nelle impostazioni).
- Viene mantenuto un elenco dei file scaricati in quanto necessario per abilitare o meno la possibilità di recensire un file trovato tramite la ricerca. Se questo è presente nella tabella, è infatti disponibile un ulteriore pannello attraverso cui scrivere e inviare al proprietario la propria recensione.
- Nella tabella dei file scaricati, il campo **author** viene popolato con nome e cognome del proprietario del file. Non viene utilizzato il suo ID del nodo in quanto diverso ad ogni connessione alla rete.
- Nella tabella dei file scaricati, il campo **show** è utilizzato per decidere se mostrare o meno il file nella lista dei download (un file potrebbe essere eliminato dalla lista dei download visibili ma deve comunque essere recensibile).
- Nella tabella dei file scaricati, il campo **path** indica il percorso del file scaricato. Questo non è infatti determinabile in modo fisso in quanto l'utente può decidere ogni volta dove e con che nome salvare gli appunti.

6.2 Rappresentazione dei dati

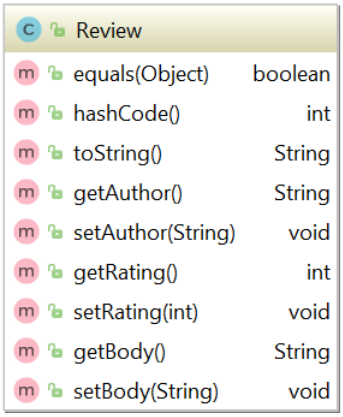
Come anticipato nel capitolo relativo all'implementazione di Kademlia, la personalizzazione del nodo e dei metadati dei file è lasciata al client:

- La classe KademliaNode viene estesa a **UniShareNode**, che implementa anche la logica delle recensioni.
- La classe KademliaFile viene estesa a **NoteFile**.
- La classe KademliaFileData viene estesa a **NoteMetadata** e vengono quindi specificati i metadati degli appunti (titolo, autore, università, dipartimento, corso e docente).



L'implementazione lato client prevede l'utilizzo della classe Review, volta a rappresentare una recensione, e di un nuovo messaggio (ReviewMessage) finalizzato all'invio di una recensione o alla ricezione di quelle esistenti. Tale messaggio estende ovviamente la classe Message, mostrata nel capitolo relativo all'implementazione di Kademlia, e la logica di ricezione e risposta è implementata nel metodo onMessageReceived di UniShareNode.

È inoltre previsto l'utilizzo di listeners (come ReviewsListener) in quanto la procedura di comunicazione è asincrona e non si ha pertanto conoscenza di quando, e se, arriveranno dei messaggi di risposta dagli altri nodi della rete.



6.3 Managers

Sono state create cinque classi volte a fornire metodi di supporto all'intera applicazione:

- **ConnectionManager**: permette di comunicare con il server ai fini di login / registrazione (mantenendo traccia del corrente stato di login) e fornisce accesso al nodo attraverso cui il client si presenta sulla rete.
- **DatabaseManager**: permette di effettuare operazioni sul database e provvede alla sua creazione qualora non esistesse.
- **DownloadsManager**: tiene traccia dei download in corso o precedentemente effettuati.
- **FilesManager**: si tratta dell'implementazione del FilesProvider e permette pertanto di ottenere il file reale data la sua rappresentazione nella rete di Kademlia.
- **ReviewsManager**: permette di ottenere le recensioni di un file o di salvarne una.