Relazione Progetto WORTH

Laboratorio di Reti dei Calcolatori - A.A. 2020/2021

Sara Grecu

Indice

1 Introduzione

WORTH(WorkTogetHer) e' un progetto didattico per la gestione di lavori di gruppo in modo collaborativo da remoto. Ispirandosi ad un metodo di gestione 'agile', permette di avere una vista di insieme delle attivita' da svolgere di un progetto e ne visualizza l'evoluzione. Infatti, gli utenti potranno, oltre che creare nuovi progetti, collaborare agilmente con altri utenti iscritti al servizio, creando nuove card per ogni lavagna virtuale, riorganizzando le attivita', spostando le card da una lista ad un'altra, o aggiungendo nuovi membri al gruppo di lavoro. Inoltre, WORTH predispone un sistema di chat per favorire la comunicazione e la condivisione di idee fra i membri di ogni progetto.
L'intero sistema e' supportato da un server centrale remoto, che si occupera' di ricevere e gestire la collaborazione fra gli utenti della piattaforma.

2 Architettura generale del sistema

L'architettura generale del sistema e' basata sul paradigma Client-Server. Gli utenti del servizio possono accedere al sistema mediante un Client, che invia le richiete ad un server (in rete locale o Internet) che le elabora e restituisce le informazioni richieste dal client. Le connessioni instaurate fra client e server sono affidate al protocollo TCP/IP. Al momento del lancio di un client, questo instaura una connessione TCP con il server, presso un indirizzo IP e porta noti; inoltre, tramite il paradigma RMI, il client crea un riferimento agli oggetti remoti pubblicati dal server su un registry noto: questi saranno necessari per l'operazione di registrazione al servizio, e verranno invocati su richiesta dell'utente. In seguito alla registrazione, il client potra' quindi effettuare l'operazione di login, unica operazione che permettera' di inviare ulteriori richieste. In seguito al successo dell'operazione di login, il client potra' inviare nuove richieste al server.

I messaggi scambiati tra il client e il server sono codificati come oggetti **Request** e **Response** in formato JSON, che, rispettivamente, rappresentano una richiesta e una risposta. Le classi rappresentanti tali oggetti sono presenti nel path *WORTH/shared/worthProtocol*.

Ogni messaggio JSON inviato dal client dovra' obbligatoriamente contenere un campo *nickUtente* contenente una stringa indicante il nome utente associato al client che ha effettuato la richiesta, e un campo **request**, contenente una stringa, interpretata come RequestType, e che identifica univocamente il tipo della richiesta che il server dovra' elaborare. Ogni metodo, necessita a sua volta di altri parametri obbligatori che saranno aggiunti alla **Request**: ogni parametro necessario al metodo per essere eseguito e' identificato da un nome noto, per permettere al server di invocare i metodi necessari senza ambiguita'.

Una volta ricevuta una richiesta di un client, il server effettua la deserializzazione della **Request**, e procede all'invocazione del metodo richiesto; in seguito all'elaborazione della richiesta, viene generata una **Response**, che contiene il risultato dell'elaborazione.
La semantica della **Response** prevede:

- un booleano *done*, che indica se l'operazione e' avvenuta con successo o no;
- una stringa *explanation*, che, in caso di successo sara' uguale a 'success', altrimenti, fornisce la motivazione per cui una richiesta non e' stata portata

a termine (quindi conterra' il messaggio di una eccezione lanciata all'interno del server);

I campi di cui sopra vanno a costituire una risposta base per operazioni che non richiedono dati (ad esempio, la login). Nel caso in cui il client vada a richiedere dei dati, il server inizializzera' ulteriori campi noti (ad esempio, per la listProjects, il campo *List<Project> projects*).

La connessione TCP sulla quale viaggiano le richieste e' mantenuta aperta finche' il client non invia una richiesta di logout. Ad interrompere la connessione e' il client, subito dopo che ha inviato la richiesta. Il server, appena la riceve, cambia lo stato dell'utente ad offline.

3 Implementazione del Server

Il server si occupa di gestire le connessioni di rete verso i client e mantiene al suo interno le strutture dati necessarie al soddisfacimento delle richieste; si occupa di salvare in memoria secondaria i dati relativi allo stato della piattaforma, nonche' del ripristino delle strutture in seguito al riavvio. Al server e' inoltre delegato il compito della gestione e dell'assegnamento degli indirizzi multicast utilizzati dalle chat di progetto. E' stato sviluppato in ambiente UNIX, utilizzando Java 8. E' inoltre stata utilizzata la libreria Jackson, per la serializzazione e deserializzazione degli oggetti JSON.

Il server, per prima cosa, legge da memoria secondaria i progetti memorizzati (quindi anche le relative card, l'indirizzo IP e la lista dei membri di ciascuno di essi) e gli utenti registrati a WORTH. In seguito, inizializza le strutture dati in memoria principale, in base ai dati appena letti.

Terminata l'inizializzazione delle strutture dati, il server prova a creare un'istanza della classe **SocketServices**, il cui costruttore accetta come argomenti tre parametri: la struttura dati appena inizializzata, l'hostname e la porta per effettuare il bind della socket. In caso di fallimento viene lanciata un'eccezione di tipo *IOException*, altrimenti, la funzione main invoca il metodo *start()* sull'oggetto appena creato. A questo punto, il nuovo oggetto SocketServices avvia il server RMI e il selector, cosi' da poter accettare nuove richieste alle porte specificate (TCP:8080, RMI:8081).

3.1 Funzionalita' di rete e gestione delle connessioni

Il server TCP, implementato nella classe **SocketServices**, e' stato implementato con comportamento non bloccante, utilizzando la libreria NIO. Questo effettua il multiplexing delle richieste basandosi su tre componenti fondamentali:

- un Selector per iterare sulle connessioni attive ed effettuare il multiplexing di queste;
- una Welcoming socket per interfacciarsi con i nuovi client;
- un Threadpool che per ogni richiesta dei client assegna un thread. Questo e' un *newCachedThreadPool*, per cui, quando possibile, ricicla i thread, non dovendone sempre creare uno nuovo per ogni richiesta.

In seguito all'invocazione del metodo start(), il server entra in un ciclo while(!Thread.interrupted()); ad ogni iterazione e' verificata la presenza o meno di canali pronti per effettuare qualche operazione. Se il selettore non e' vuoto (ovvero, restituisce un valore diverso da zero invocando su di esso il metodo select()), si crea un oggetto di tipo Iterator<SelectionKey> per iterare sul set di chiavi corrispondendi ai canali disponibili. Per ogni chiave restituita dall'iteratore, si individua, dapprima se questa e' valida, e poi il tipo di evento pronto da essere elaborato:

- se la chiave corrisponde ad un evento di tipo OP_ACCEPT, vi e' una nuova connessione in arrivo e si invoca il metodo registerClient(SelectionKey key), che si occupa di creare una active socket per la comunicazione client-server, di allocare un ByteBuffer per la memorizzazione dei dati (e farne la conseguente attach()) e di registrare la SelectionKey come pronta in lettura;
- se la chiave corrisponde ad un evento di tipo OP_READ si procede con l'handling della richiesta chiamando il metodo readMessage(SelectionKey key;
- se la chiave corrisponde ad un evento di tipo OP_WRITE si procede con l'invio della risposta al client utilizzando il metodo writeMessage(SelectionKey key.

3.2 Il metodo readMessage

Innanzitutto recupera l'attachment associato alla chiave corrente, il quale e' un ByteBuffer di lunghezza pari a quella di un intero, perche' si aspetta di ricevere la lunghezza del messaggio che il client gli inviera'. Poi, recupera la active socket necessaria, ed effettua la read. Si possono verificare tre casi: il client si e' disconnesso, per cui legge -1; legge la size della richiesta del client; legge la richiesta del client.

Nel caso in cui il server legga la size della richiesta, rialloca la dimensione del ByteBuffer passato come attachment; altrimenti, dopo una sleep di 200 millisecondi[1], invia la richiesta da svolgere al threadpool e registra la chiave come pronta in scrittura. Se non ha finito di leggere la richiesta, continuera' all'iterazione successiva.

3.3 Il metodo writeMessage

Dapprima recupera l'attachment associato alla chiave, poi controlla che la risposta sia definita: se non lo e', esce e controlla all'iterazione successiva, altrimenti, recuperata l'active socket relativa a quel client, serializza la risposta da inviare al client e gliela inoltra. Se non finisce di scrivere a questa iterazione, continuera' in quella seguente.

Allora, crea un nuovo ByteBuffer di dimensione 4 (perche' si aspetta di ricevere la dimensione del prossimo messaggio) di cui fare l'attach(), e imposta la chiave come pronta in lettura.

[1] La sleep e' stata inserita di modo tale da distanziare le submit al threadpool.

3.4 Strutture dati principali e gestione della Concorrenza

Il server, durante l'esecuzione, mantiene i dati in memoria principale attraverso mappe chiave-valore del tipo **HashMap<>** e **ConcurrentHashMap<>**. La scelta di oggetti di tipo Map e' stata ritenuta piu' efficiente dal momento che la maggior parte degli accessi alle risorse sono effettuati per nome: si pensi all'operazione di recupero delle informazioni di un utente, o alle informazioni relative alla lista dei membri di un singolo progetto. Dunque, il server possiede due strutture dati principali per le memorizzazione dei contenuti:

- Una ConcurrentHashMap<String, Project> per la memorizzazione dei progetti;
- Una ConcurrentHashMap<String, User> per la memorizzazione degli utenti.

Vediamo le scelte fatte piu' in dettaglio:

La ConcurrentHashMap<>, sia per gli utenti che per i progetti, e' stata adottata per la natura multi-threaded del server, cosi' che operazioni di aggiornamento non si sovrappongano creando problemi di concorrenza e consistenza. Si noti, tuttavia, che tipicamente operazioni di recupero e aggiornamento si possono sovrapporre, facendo, quindi, restituire (ad operazioni come una get()) dati non consistenti, ma relativi al piu' recente e completo aggiornamento. Per quanto riguarda l'accesso e l'aggiornamento dei singoli progetti, tutti i metodi presenti sono synchronized, cosi' che non si verifichino problemi di concorrenza (visto che la combinazione di operazioni atomiche non e' atomica). Infatti, per le liste all'interno dei progetti sono stati scelti semplici ArrayList<>, cosi' che si riducesse al minimo il rischio di serializzarne troppo l'interazione.

3.5 La classe UserManager

Come suggerisce il nome, la classe si occupa della gestione dei vari utenti, e dell'interazione tra loro e il server. Questa include una **ConcurrentHashMap<>** contenente gli utenti registrati a WORTH, l'istanza del file su disco, un'istanza della classe d'appoggio per la serializzazione/deserializzazione della classe (RegisteredFile, e la List<NotifyUsersInterface> che contiene i riferimenti ai client registrati al servizio di notifica RMI Callback.

La classe e' un **Singleton**, per cui esiste una ed una sola sua istanza all'interno del server. Quindi, ogni volta che un thread voglia accedere alla ConcurrentHashMap degli utenti, deve prima richiedere una istanza della classe. Tutti i metodi presenti nello UserManager sono **synchronized** per evitare problemi di concorrenza, dal momento che, come gia' scritto, la combinazione di operazioni atomiche non e' atomica.

3.5.1 RMI per le operazioni di Registrazione e Callback

UserManager include, oltre le operazioni di get, login e logout, l'operazione di **registrazione** tramite RMI, e quelle che permettono ad un utente di registrarsi al servizio di notifica tramite RMICallback. Quest'ultimo, avvisa agli utenti registrati al servizio, quando un utente si registra e/o cambia stato. Le interfacce delle operazioni implementate di cui sopra sono nella cartella WORTH/shared/rmi.

3.6 Le classi Handler e Worker

La classe *Handler* e' un Future che prende in ingresso una richiesta di tipo **Request**, ne effettua il parsing e chiama un metodo di Worker per poterla eseguire. Dopo che la richiesta e' stata svolta, la stessa classe Handler prepara la risposta di tipo *Response* per il client e la restituisce. Tale risposta verra' quindi restituita al thread Main nella funzione **writeMessage**. Quindi, *Handler* tramite la funzione parser() effettua il parsing della richiesta, e tramite response() costruisce una risposta base che verra' poi specializzata all'interno della *call()*. La risposta base prevede un booleano ed una stringa:

1. <true, "success"> se l'operazione e' andata a buon fine;
2. <false, explanation>, se l'operazione e' fallita.

Explanation e' una stringa contenente il messaggio restituito dalle eccezioni lanciate dai metodi chiamati da *Worker*.

La classe *Worker*, invece, e' una classe ausiliaria a *Handler*, che esegue controlli di consistenza e poi chiama i metodi necessari allo svolgimento della richiesta, e implementati in Project.

3.7 La classe Project

La classe Project.java rappresenta un progetto del sistema WORTH e si occupa di memorizzare ed effettuare operazioni sui dati relativi al progetto (come la gestione di persistenza sul disco, spostamento di card e controlli sui vincoli imposti dalla specifica). All'interno della classe troviamo svariati attributi:

- Un *nameProject*, nome del progetto;
- Le varie liste *List<Card> to_Do, inProgress, toBeRevised, done*, che contengono eventuali oggetti Card;
- La *List<String> members*, che contiene i nomi dei membri del progetto;
- Un *File project*, e cioe' il riferimento su disco del progetto corrente;
- Uno *ProjectUtils info*, istanza della classe di supporto per la serializzazione /deserializzazione JSON, e che conterra' informazioni sul progetto (lista dei nomi dei membri del progetto e indirizzo IP associato ad esso);
- Un *InetAddress addressUdo*, indirizzo IP associato al progetto'
- Un *AddressGenerator addressGenerator*, istanza della classe che si occupa del riciclo e della creazione di indirizzi IP per associarli ai progetti;
- Un *serialVersionUID* per la serializzazione/deserializzazione del Project;

La classe, come User e Card, contiene due costruttori: uno vuoto necessario alla serializzazione/deserializzazione per Jackson, e uno che inizializza effettivamente gli attributi della classe. Infine, e' possibile trovare metodi richiesti, come addCard o moveCard, nella specifica. Si noti che non e' presente un ProjectManager di modo tale che l'utilizzo dei Project sia piu' parallelizzabile.

3.8 La classe User

Questa classe rappresenta un utente della piattaforma. Un'istanza di una classe User e' creata dal server RMI al momento dell'iscrizione. Ogni oggetto della classe presenta cinque attributi:

1. *String name*, nome dello user;
2. *String password*, password dello user;
3. *LinkedList<Project* list_prj, lista dei progetti a cui appartiene l'utente;
4. *boolean online*, che indica se l'utente e' online oppure no;
5. Un *serialVersionUID* per serializzazione/deserializzazione della classe;

I metodi presenti sono semplici get() e set(), oltre che la override della *toString()*, utile per l'operazione listUsers, dato che vengono restituiti semplicemente il nome e lo stato dell'utente.

3.9 La classe Card

Questa classe rappresenta una card all'interno di una lista di un Project. Un'istanza di una classe Card e' creata al momento dell'operazione *addCard*. Ogni oggetto della classe presenta sei attributi:

1. *String nameCard*, nome della card;
2. *String description*, descrizione della card;
3. *List<String> history*, history della card;
4. *String currentList*, lista corrente in cui si trova la card;
5. *ObjectMapper mapper e serialVersionUID*, necessari alla serializzazione/-deserializzazione della classe;

3.10 Generazione e riuso di indirizzi multicast con AddressGenerator

La generazione e il riuso degli indirizzi multicast e' implementato in AddressGenerator. Infatti, la classe contiene una lista che include tutti gli indirizzi liberi riutilizzabili da un nuovo progetto. Quest'ultima viene serializzata (e rispettivamente deserializzata) su disco, cosi' da poter riutilizzare indirizzi multicast anche dopo un eventuale riavvio del server.

Anche AddressGenerator, come UserManager, e' un *Singleton*, cosi' che tutti i progetti utilizzino la stessa lista di indirizzi liberi. Si noti che un indirizzo diventa "libero" nel momento in cui un progetto viene cancellato e il suo indirizzo IP viene aggiunto a tale lista.

3.10.1 Generazione di un indirizzo multicast

La funzione che si occupa di generare un indirizzo multicast (se non ce ne sono di liberi) e' *newAddress*, costituita da:

```
int index = (projectName.hashCode()) / 256;
return InetAddress.getByName("239." +
    Math.abs(((index / 256) / 256)) + "." +
    Math.abs(((index / 256) % 256)) + "." +
    Math.abs(index % 256));
```

Come si puo' notare, l'indirizzo IP e' generato a partire dall' hashcode del nome del progetto.
Ma, attenzione: all'inizio e' necessario dividere index per 256 perche' una stringa troppo lunga o con tante lettere uguali puo' far lievitare esponenzialmente la sua grandezza.

3.11 La persistenza sul filesystem

La persistenza sul filesystem e' garantita dall'utilizzo della libreria Jackson e dalle classi di appoggio presenti nella cartella *persistence*. Queste sono: CardFile, RegisteredFile, UtilsFile, IPFile; dove rispettivamente: CardFile e' necessaria alla serializzazione/deserializzazione di una card, RegisteredFile, degli utenti registrati al servizio; ProjectUtils, dei nomi dei membri del progetto e dell'indirizzo IP ad esso associato; IPFile, degli indirizzi IP liberi sino a quel momento.

Genericamente, si procede in tale modo:

1. Mentre viene creato un oggetto della classe, viene anche creato un mapper e viene istanziato un oggetto della classe di appoggio;
2. Se non esiste gia' un file su disco riguardante quelle informazioni, viene creato; altrimenti, si legge da esso e viene memorizzato in memoria principale;
3. Ogni volta che viene aggiornata la struttura in memoria principale viene aggiornato anche il file su disco;

4 Implementazione del Client

Il Client implementa le funzionalita' con cui gli utenti di WORTH interagiscono con il server. Questo, essendo stato sviluppato come tool a riga di comando, prevede che l'utente scriva, dopo ">" il comando e i parametri richiesti (login nickname password, ad esempio). Se necessario, per conoscere la sintassi dei comandi da usare e' sufficiente scrivere "help" e verranno stampati a video tutti i comandi e i relativi parametri da inserire.

La prima classe ad essere eseguita e' **MainClient**, che esegue il parsing delle richieste e le instrada alle classi specifiche. Quindi, ora, differenziando i comandi che un utente puo' utilizzare, andiamo ad approfondire le classi che costituiscono il Client:

1. Nel caso in cui il comando preveda una connessione TCP con il server (addMember o addCard, ad esempio), le classi protagoniste sono: **WORTHClient** e **TCPClient**. Per cui, dopo aver superato i vari controlli, dal

MainClient viene chiamato il metodo opportuno di WORTHClient per inviare la richiesta. WORTHClient si appoggia a TCPClient, dove sono implementate la send() e la receive() su connessioni TCP;

2. Nel caso in cui il comando preveda interazione RMI con il server (con una register, ad esempio), dopo aver superato i controlli, dal MainClient viene chiamato il metodo opportuno di RMIClient;

3. Infine, nel caso in cui si preveda una interazione connection-less con il server (con un comando readChat, ad esempio), vengono utilizzate le classi **WORTHClient** e **UDPClient**, di cui parleremo nel seguente paragrafo.

4.1 La classe UDPClient e ProjectChat

UDPClient si occupa dell' interazione connection-less con il server, tramite la porta 8081. La classe consta di:

- Una HashMap<String, ProjectChat> chat, che mappa al nome di un progetto un oggetto di tipo ProjectChat, la quale e' una classe composta da un **indirizzo IP** e un **vettore** di messaggi associati a quell'indirizzo;

- La socket multicast utilizzata per la comunicazione connection-less.

Essendo UDPClient un thread, rimane in ascolto sulla socket, bloccandosi sulla receive() se non ci sono messaggi. Quando, invece, ne arrivano, effettua il parsing del messaggio (che contiene nome del progetto `\n` messaggio effettivo), e lo aggiunge al buffer dell'oggetto ProjectChat. Quando poi un utente vorra' leggere i messaggi della chat di un progetto, gli verra' restituito quel buffer, il cui contenuto verra' stampato a video. Nel caso in cui il client voglia inviare un messaggio su una chat, conoscendo l'indirizzo IP, invia il datagramma sulla socket multicast.

Il Client legge i messaggi associati ad un progetto tramite metodi implementati in **WORTHClient**.

4.1.1 Discovery degli indirizzi IP

Perche'il Client conosca gli indirizzi IP a cui collegarsi, per comunicare tramite chat, sono stati fissati dei momenti e dei comandi che inviano al Client gli indirizzi IP dei progetti di cui e' membro.

Questi momenti sono:

- Subito dopo il login (se va a buon fine);

- Subito dopo la listProjects (se va a buon fine);

- Subito dopo la creazione di un progetto (se va a buon fine);

Per cui, se un utente viene aggiunto ad un progetto, e' necessario che esegua la listProjects se vuole ricevere i messaggi dalla chat. Si veda il paragrafo *Problemi e possibili migliorie* per leggere di una soluzione migliore e come sarebbe stata implementata.

Il metodo utilizzato per memorizzare gli indirizzi IP di ciascun progetto e' *setIPAddresses(List<Project> projects*, che prende come parametro la lista di

progetti restituita nei momenti di cui sopra, e cosi' da ogni Project preleva l'indirizzo IP associato, e crea un oggetto di tipo ProjectChat, che verra' aggiunto all' HashMap<String, ProjectChat> chat, in UDPClient. Tale metodo e' implementato nella classe WORTHClient.

5 Quick start guide

Il software e' stato sviluppato usando l'ambiente di sviluppo JetBrains IntelliJ IDEA 2021.2.2. La versione Java di riferimento e' la versione 8. E' stata usata la libreria esterna Jackson, e Maven per gestire le dipendenze.

5.1 Istruzioni per la compilazione e l'esecuzione

E' stato utilizzato il plugin *exec-maven-plugin*, aggiunto all'interno di pom.xml. Questo permette di poter sia eseguire che compilare il codice con:

```
mvn exec:java@server -Dexec.args="localhost"
```

per il Server, e

```
mvn exec:java@client -Dexec.args="localhost"
```

per il Client. Ciascun comando prevede di prendere in ingresso un argomento, e cioe' l'hostname del server. Qui sopra e' stato inserito *localhost*.

5.2 Utilizzo dell'interfaccia

Dopo che il Client viene messo in esecuzione, verra' stampato a video il carattere >, dopo cui sara' possibile scrivere un comando da eseguire. Nel caso in cui non si sappia a prescindere quali comandi usare, sara' possibile scrivere il comando help e verranno visualizzati tutti i possibili comandi con la relativa sintassi. Allora, sara' possibile effettuare il comando **register**, e cosi' anche il **login** (oppure solo il login, nel caso di registrazione gia' avvenuta). Durante l'esecuzione l'utente potra' usare i comandi riportati da *help*. In caso di errori, verra' stampato un breve messaggio relativo alla natura del problema.

Invece, in caso di successo, sia per operazioni che prevedono la ricezione dei dati, sia per quelle che non la prevedono, sara' stampato a video `< success:` *operation*. Nel caso si ricevano dei dati, questi saranno visualizzabili dall'utente in maniera intuitiva.

6 Problemi e possibili migliorie

6.0.1 Scrivere una richiesta da riga di comando

Per poter scrivere una richiesta da riga di comando si attende il ">", ma se precedentemente e' stato stampato il messaggio di una eccezione ci troveremmo nel caso in cui a video troveremmo:

```
> Messaggio dell'eccezione
— Spazio in cui inserire la richiesta —
```

Il che e' controintuitivo, ma e' da attribuirsi al buffering dello Standard Output e Standard Error.

6.0.2 Chiusure inattese di un client

Nel caso in cui un Client si chiuda inaspettatamente, il Server attuale non memorizza l'informazione, per cui, sia in memoria principale che secondaria risultera' che quell'utente e' online. Una possibile **soluzione** a questo problema e' memorizzare i client nella *ConcurrentHashMap<SelectionKey, User>* mappandoli tramite la chiave che gli e' stata assegnata dal selector. Cosi', nel momento in cui il server riceve un -1 dal client (informazione che ci dice che l'utente si e' disconnesso), da **SocketServices** sarebbe possibile cambiare lo stato dell'utente.

6.0.3 Password non cifrate

Una possibile **soluzione** a questo problema e' cifrare/decifrare le password importando, ad esempio, la dipendenza esterna SecretKeySpec combinandola con l'uso dell'algoritmo AES.

6.0.4 Notificare ad un utente quando viene aggiunto ad un progetto

Una **soluzione** piu' funzionale sarebbe utilizzare RMICallback per notificare quando un utente viene aggiunto ad un progetto, e quindi, per fargli anche avere l'indirizzo IP del suddetto progetto a cui collegarsi per ricevere i messaggi della chat. Una soluzione sarebbe potuta essere la creazione di un ProjectManager analogo a UserManager, nonostante questo avrebbe inficiato sulle performance in termini di parallelizzazione, perche', come UserManager, sarebbe dovuto essere un *Singleton*.

6.0.5 Creazione di eccezioni ad hoc

Invece di lanciare sempre una nuova eccezione di tipo Exception, si potrebbero creare eccezioni ad hoc, visto che molte di essere ricorrono spesso (come ad esempio, se si vuole fare la login ma quell' utente e' gia online).