

Relazione Progetto WORTH

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

Sara Grecu

Indice

1	Introduzione	2
2	Architettura generale del sistema	2
3	Implementazione del Server	3
3.1	Funzionalita' di rete e gestione delle connessioni	3
3.2	Il metodo readMessage	4
3.3	Il metodo writeMessage	4
3.4	Strutture dati principali e gestione della Concorrenza	5
3.5	La classe UserManager	5
3.5.1	RMI per le operazioni di Registrazione e Callback	6
3.6	Le classi Handler e Worker	6
3.7	La classe Project	6
3.8	La classe User	7
3.9	La classe Card	7
3.10	Generazione e riuso di indirizzi multicast con AddressGenerator	8
3.10.1	Generazione di un indirizzo multicast	8
3.11	La persistenza sul filesystem	8
4	Implementazione del Client	9
4.1	La classe UDPClient e ProjectChat	9
4.1.1	Discovery degli indirizzi IP	10
5	Quick start guide	10
5.1	Istruzioni per la compilazione e l'esecuzione	10
5.2	Utilizzo dell'interfaccia	11
6	Problemi e possibili migliorie	11
6.0.1	Scrivere una richiesta da riga di comando	11
6.0.2	Chiusure inattese di un client	11
6.0.3	Password non cifrate	11
6.0.4	Notificare ad un utente quando viene aggiunto ad un progetto	12
6.0.5	Creazione di eccezioni ad hoc	12



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 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 richieste 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 il login, unica operazione che permettera' di inviare ulteriori richieste, se va a buon fine.

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 nella cartella WORTH/shared/worthProtocol.

Ogni richiesta dovra' obbligatoriamente avere non nulli i campi **nickUtente**, contenente una stringa indicante il nome utente associato al client che ha effettuato la richiesta, e **request**, contenente una stringa, interpretata come **RequestType**, e che identifica univocamente il tipo della richiesta che il server dovra' elaborare. In particolare, una RequestType e' una enum contenente tutti i tipi di richiesta: login, logout, addMember, etc.

Tuttavia, ogni metodo necessita a sua volta di altri parametri obbligatori che saranno aggiunti alla richiesta: 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). Invece, nel caso in cui il client li richieda, 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.

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. Quindi, 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.java`, 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:

- Un **Selector** che gli permette di effettuare il multiplexing delle richieste;
- Una **welcoming socket** con comportamento non bloccante;
- Strutture dati e API di Java **NIO**;
- Un **newCachedThreadPool** a cui, per ogni richiesta del Client, sottomette una nuova istanza di Handler che, dopo l'elaborazione, restituira' la risposta.



In seguito all'invocazione del metodo `start()`, il Server entra in un ciclo `while(!Thread.interrupted())`, dove 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 corrispondenti ai canali disponibili. Per ogni chiave restituita dall'iteratore, si individua se questa e' valida, e poi il tipo di evento pronto da essere elaborato:

- 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 sara' 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, e lo ripassa alla `attach()`; altrimenti, 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 del Selector.

3.3 Il metodo `writeMessage`

Dapprima, recupera l'attachment associato alla chiave, poi controlla che la risposta sia stata restituita dalla `call()` di Handler: se non lo e', esce e controlla all'iterazione successiva, altrimenti, recupera l'active socket relativa a quel client, serializza la risposta da inviargli 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.



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 **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 la scelta fatta 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 tra loro creando problemi di concorrenza e consistenza.

Si noti, invece, che operazioni di recupero e aggiornamento si possono sovrapporre, facendo 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. Scelta necessaria perche' per le liste all'interno dei progetti sono stati scelti semplici **ArrayList**<>, cosi' che si riducesse al minimo il rischio di rendere troppo sequenziale l'interazione con i thread.

Invece, per quanto riguarda i singoli utenti, l'unico caso in cui sia necessaria la gestione della concorrenza e' l'aggiornamento della **lista dei progetti** a cui il singolo utente appartiene. Infatti, la lista e' un **Vector**: scelta sufficiente perche' si effettuino solo modifiche semplici (una `add()`, ad esempio). Nel caso in cui un metodo effettui `add()` alla volta, e' previsto che questo sia **synchronized**.

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 **RegisteredFile**, la quale e' d'appoggio per la serializzazione/deserializzazione di **UserManager**;
- 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 vuole accedere alla **ConcurrentHashMap** degli utenti, deve prima richiedere una istanza della classe.



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** a WORTH tramite RMI, e quelle che permettono ad un utente di registrarsi al **servizio di notifica** tramite RMICallback, che avvisa gli utenti registrati al servizio quando un utente si registra o cambia stato.

Operazioni di registrazione a WORTH e al servizio di notifica Le operazioni di registrazione a WORTH e al servizio di notifica sono sviluppate all'interno di UserManager, che implementa l'interfaccia **Remote Interface** (che si può trovare al path `./WORTH/shared/rmi`).

Servizio di notifiche Invece, le operazioni necessarie a notificare ai client la registrazione su WORTH di altri utenti e il loro cambiamento di stato, sono sviluppate nella classe `RMIClient`, che implementa l'interfaccia `NotifyUsersInterface` (che si può trovare al path `./WORTH/shared/rmi`).

3.6 Le classi Handler e Worker

La classe **Handler.java** e' un Callable<Response> 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, che verra' quindi restituita informato Future<Response>, al thread Main nella funzione **writeMessage**. Andando piu' nello specifico, tramite la funzione **parser()**, Handler 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.java**, invece, e' una classe ausiliaria di Handler, che esegue controlli di consistenza e chiama i metodi necessari allo svolgimento della richiesta, implementati in Project e in UserManager.

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 diversi attributi:

- Una stringa contenente il nome del progetto;



-
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 alla 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`, ed e' 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));
```

Si noti che e' necessario dividere index per 256 perche' una stringa troppo lunga o con tante lettere uguali puo' far lievitare esponenzialmente l'ordine di grandezza dell'hash. Nell' improbabile caso di collisioni, dove si ha un projectName simile ad un progetto gia' creato, verra' stampato un messaggio d'errore in cui si chiede di creare un progetto con un nome diverso, aggiungendogli un carattere a scelta dell'utente.

3.11 La persistenza sul filesystem

La persistenza sul filesystem e' garantita dall'utilizzo della libreria Jackson e dalle classi di appoggio presenti nel path `./WORTH/persistence`.

Queste sono:

- CardFile, necessaria alla serializzazione/deserializzazione di una card;
- RegisteredFile, necessaria alla serializzazione/deserializzazione degli utenti registrati al servizio;
- ProjectUtils, necessaria alla serializzazione/deserializzazione dei nomi dei membri del progetto;
- IPFile, necessaria alla serializzazione/deserializzazione 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 già 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 funzionalità 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 carattere ">" il comando e i parametri richiesti (login nickname password, ad esempio). Se necessario, per conoscere la sintassi dei comandi da usare è sufficiente scrivere "help" e verranno stampati a video tutti i comandi e i relativi parametri da inserire.

La prima classe ad essere eseguita è **MainClient**, che esegue il parsing delle richieste e le instrada alle classi specifiche. Quindi, ora, differenziando i comandi che un utente può 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 è 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.



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

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.

- 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);

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`.

Il software e' stato sviluppato in ambiente UNIX, 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.

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:

per il Server, e



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

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

5.2 Utilizzo dell'interfaccia

Dopo che il Client viene messo in esecuzione, verrà stampato a video il carattere `>`, dopo cui sarà possibile scrivere un comando da eseguire. Nel caso in cui non si sappia a prescindere quali comandi usare, sarà possibile scrivere il comando `help` e verranno visualizzati tutti i possibili comandi con la relativa sintassi. Allora, sarà possibile effettuare il comando `register`, e così anche il `login` (oppure solo il `login`, nel caso di registrazione già avvenuta). Durante l'esecuzione l'utente potrà usare i comandi riportati da `help`. In caso di errori, verrà 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, sarà 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 carattere `>`, ma se precedentemente è stato stampato il messaggio di una eccezione ci troveremo nel caso in cui a video troveremo:

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

Il che è controintuitivo, ma è 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 risulterà che quell'utente è online. Una possibile **soluzione** a questo problema è memorizzare i client nella `ConcurrentHashMap<SelectionKey, User>` mappandoli tramite la chiave che gli è stata assegnata dal selector. Così, nel momento in cui il server riceve un `-1` dal client (informazione che ci dice che l'utente si è disconnesso), da `SocketServices` sarebbe possibile cambiare lo stato dell'utente.

6.0.3 Password non cifrate

Una possibile **soluzione** a questo problema è 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** più 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, perché, 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).

