



UNIVERSITÀ DI PISA

Dipartimento di Informatica
Corso di Laurea in Informatica

WORTH

Reti di calcolatori e laboratorio

Prof.ssa Federica Paganelli
Prof.ssa Laura Emilia Maria Ricci

Luca Cirillo
545480 - Corso A

Anno Accademico 2020/2021

1. Introduzione

WORTH

WORKTogetHer, abbreviato in *WORTH*, è uno strumento per la gestione di progetti collaborativi, ispirato ad alcuni principi della metodologia *Kanban*. Il suo obiettivo è quello di aiutare le persone, generalmente i membri di un team, ad organizzarsi e coordinarsi nello sviluppo di progetti di qualsiasi tipologia, purché possano essere suddivisi in un insieme di attività da portare a termine. Il metodo *Kanban*, termine di origine giapponese che viene tradotto letteralmente come "insegna", pone il suo punto di forza nella **visualizzazione** del progresso durante lo sviluppo di un progetto: la lavagna, parete, bacheca o comunque lo strumento utilizzato per organizzare i tasks, fornisce una vista d'insieme delle attività svolte e di quelle ancora da svolgere, ed è tipicamente esposto in una posizione facilmente accessibile da tutti i membri del team. Ogni membro può prendere in carico un'attività tra quelle ancora da svolgere, lavorarci su, e dopo un'eventuale fase di revisione, contrassegnarla come completata. Questo workflow, di stampo *Agile*, permette di scomporre un compito in tanti, piccoli tasks, permettendo uno sviluppo modulare di un compito inizialmente complesso.

Parole chiave

E' utile chiarire alcune parole chiave del sistema *WORTH*, che fanno riferimento a punti cardine dello sviluppo.

1 Progetto

Un **progetto** è un qualcosa da sviluppare, che può essere suddiviso in attività più piccole. Queste vengono rappresentate da **cards** che attraversano quattro liste durante il loro ciclo di vita, in relazione alla fase in cui si trovano in quel momento. Ad ogni progetto possono far parte più **utenti** del sistema, che ne diventano **membri** e sono quindi autorizzati ad interagire con le cards al suo interno. Il sistema mette inoltre a disposizione una **chat di progetto**, che offre ai membri la possibilità di comunicare facilmente tra di loro.

2 Membro

Un **membro** di un progetto identifica un particolare **utente** del sistema *WORTH* che partecipa al suo sviluppo. Un utente che crea un progetto ne diventa automaticamente membro, e da questa posizione può **aggiungere** altri utenti come membri. Tutti possiedono gli **stessi diritti e possibilità** del creatore del progetto: aggiungere cards, spostarle, visualizzare informazioni sul loro stato, utilizzare la chat, e anche cancellare l'intero progetto.

3 Card

Una **card** rappresenta un'attività del progetto da sviluppare. E' l'equivalente di un post-it su cui è riportata proprio l'attività da portare a termine che aggiunge, rimuove o corregge aspetti del progetto nel suo insieme. Questo termine non è stato tradotto nel suo equivalente italiano di "*carta*" per non snaturare il significato iniziale che rimane comunque chiaro. Ad ogni card è associato un **nome**, che la identifica nel contesto del progetto, una **descrizione**, qualora il nome non fosse sufficiente a identificare l'attività da svolgere. Inoltre, in ogni istante di tempo del suo ciclo di vita una card può trovarsi in una ed una sola **lista** del progetto. Le liste sono quattro: in *ToDo* arrivano automaticamente le cards al momento della loro creazione, e rappresenta ovviamente lo stato iniziale di un'attività; nella lista *In Progress* si trovano le cards prese in carico dai membri del team; in *To Be Revised* vengono inserite quelle attività che, dopo essere state completate richiedono un'ulteriore revisione prima di finire in *Done*, dove risiedono le cards portate a termine.

2. Sviluppo

Dettagli sul metodo di sviluppo

Il progetto è stato sviluppato con il supporto dell'IDE *IntelliJ IDEA*, mentre lo strumento *Maven* ne ha permesso una facile compilazione ed esecuzione; l'utilizzo del version control *Git* ha concesso il beneficio di poter sviluppare con più tranquillità e maggiore controllo sul codice, grazie alla possibilità di spostarsi in branches separati in cui portare avanti le singole features, e tornare facilmente indietro in caso di problemi. La versione di *Java* utilizzata è la numero 8. In più occasioni si è presentata la possibilità di utilizzare strumenti disponibili solo in versioni di *Java* successive, tra cui gli *enhanced switch* e le *multi-line strings* (o anche text blocks): si è preferito comunque evitare l'upgrade a nuove versioni del *JDK* e rimanere sulla versione 8 con un piccolo trade-off in termini di leggibilità del codice e altre piccole migliorie che si sarebbero potute applicare. Il codice è stato **ampiamente commentato** per garantirne una comprensione a 360°, ma comunque scritto per essere quanto più possibile "*self explanatory*", con nomi di variabili, metodi e statements chiari e significativi. Questi ultimi sono stati scritti in lingua Inglese, mentre i commenti interamente in Italiano. Per quanto riguarda le dipendenze, si è scelto di utilizzare la libreria *Jackson* per la serializzazione degli oggetti in files **JSON**. Le dipendenze sono gestite da *Maven* e, grazie anche alle istruzioni che seguono e alla presenza di uno script *Bash* per automatizzare la compilazione ed il packaging, è possibile ottenere dal codice i due eseguibili in pochi secondi. Infine, è possibile eseguire **Client** e **Server** con un ulteriore parametro (opzionale) "*debug*" per attivare la modalità omonima, in cui ogni informazione utile ai fini di analisi e test viene stampata sul terminale. Questa può essere attivata su entrambi gli eseguibili, su nessuno dei due, oppure su uno soltanto.

1 Dalla compilazione all'esecuzione

Per compilare ed eseguire correttamente il sistema *WORTH* è necessario creare **due eseguibili separati**, uno per il client ed uno per il server. Si suppone di avere già installato *JDK* versione 8 e lo strumento *Maven* sul sistema. E' presente inoltre uno script *Bash* che automatizza questo processo, ma si riportano qui i passaggi eseguiti.

```
# Supponendo di essere gia' nella directory del progetto, si creano gli eseguibili
mvn clean package
# Per semplicita' si spostano nella cartella principale
# e contestualmente si rinominano per maggiore chiarezza
mv target/Client-jar-with-dependencies.jar Client.jar
mv target/Server-jar-with-dependencies.jar Server.jar
# E' possibile finalmente eseguire le applicazioni
# Esecuzione del server (CTRL-C per uscire)
java -jar Server.jar [debug]
# Esecuzione del client (CTRL-C oppure comando <quit> per uscire)
java -jar Client.jar [debug]
```

Dettagli implementativi

WORTH è implementato seguendo il paradigma **client-server**. E' permessa tuttavia una sola istanza del server ma molteplici istanze dei clients, che rappresentano gli utenti che interagiscono con il sistema.

1 Server

ServerMain è la classe che definisce la struttura del server di WORTH. Essa mette a disposizione una comunicazione tramite protocollo **TCP** e **RMI**. La classe, oltre ad estendere **RemoteObject**, implementa (nel senso del linguaggio Java) l'interfaccia **ServerRMI**, che mette a disposizione i metodi necessari proprio al funzionamento dell'architettura RMI. In particolare:

- *register*: un utente crea un nuovo account di WORTH;
- *registerCallback*: un client si iscrive ad un servizio di ricezione notifiche tramite callbacks;
- *unregisterCallback*: il client si disiscrive dalla ricezione di notifiche da parte del server.

Questo garantisce una comunicazione unidirezionale dal client verso il server. Si raggiunge il funzionamento completo del meccanismo di RMI callbacks con **NotifyEventInterface**, che come suggerisce il nome è un'interfaccia implementata questa volta dal client, che offre una comunicazione dal server verso di esso. Sono presenti i metodi:

- *notifyUsers*: notifica tutti gli utenti online riguardo il cambiamento di stato degli utenti di WORTH;
- *notifyNewProject*: notifica l'utente aggiunto come membro di un progetto da un altro utente;
- *notifyCancelProject*: notifica tutti i membri di un progetto quando questo viene cancellato;
- *notifyEvent*: notifica il singolo utente con un messaggio arbitrario.

La comunicazione TCP viene invece offerta attraverso **ServerSocket**, in ascolto su una specifica porta nota ai clients.

1.1 Persistenza

Ad ogni avvio, il server controlla che sul disco sia presente una specifica struttura di directories e files necessari al corretto funzionamento. Si possono verificare due situazioni: l'intera struttura non viene trovata ed il server entra in una fase di **bootstrap**, in cui crea tutti ciò di cui ha bisogno, trattandosi di un primo avvio. Altrimenti, tutto il necessario esiste già sul disco ed il server lo usa per **ripristinare** una sessione precedente, caricando in memoria tutti i dati presenti. Eventuali informazioni corrotte vengono se possibile recuperate, altrimenti cancellate dal disco ed inizializzate nuovamente. Nel dettaglio, questa struttura è così composta:

- *data/*: contiene tutte le informazioni generate dal server e dagli utenti
 - *Users.json*: dati per identificare gli utenti del sistema
 - *Multicast.json*: informazioni sull'assegnazione degli indirizzi IP multicast
 - *Projects/*: contiene tutti i dati relativi ai progetti creati dagli utenti

1.1.1 Users.json

Il file degli utenti è un file JSON che contiene tutte le informazioni necessarie al server per offrire servizi di accesso, quali **registrazione** e **login**. Al momento della registrazione, ogni utente che vuole creare un account WORTH è tenuto a fornire un nome utente ed una password di accesso. Il nome utente non deve essere già stato scelto, in quanto identifica univocamente gli utenti del sistema. Al login queste informazioni vengono utilizzate per l'autenticazione, i cui dettagli sono riportati nella sezione dedicata. Dopo alcuni eventi di registrazione, il file è così costituito:

```
[ {
  "username" : "luca",
  "password" : "3ed5ee22d96ad38af21f057b064a8eae4838f819440c594cbf9e60781dbbc7fe",
  "salt" : "03ce7dac948ed7ab74fe6f3c44477bf3"
}, {
  "username" : "sofia",
  "password" : "c31fe5131f5b4848ef100ab91a79739ccefab64437b13c213c1d146d91e711c",
  "salt" : "4dabe8f807d9471f9583007b2bda5bc4"
} ]
```

1.1.2 Multicast.json

Il file di configurazione degli **indirizzi IP multicast** permette di tenere traccia dell'ultimo indirizzo assegnato, nonché degli indirizzi **rilasciati** a seguito della cancellazione dei progetti a cui erano stati precedentemente assegnati; per i dettagli si rimanda alla sezione dedicata. Dopo alcuni eventi di creazione e cancellazione di progetti, il file è così composto:

```
{
  "lastIP" : "224.0.0.4",
  "releasedIP" : [ "224.0.0.2", "224.0.0.3" ]
}
```

1.1.3 Projects

I progetti sono salvati nella cartella *Projects/* e ad ognuno è dedicata una **sottocartella** che porta il nome del progetto. Dentro quest'ultima si troverà un file JSON con lo stesso nome del progetto, contenente alcune informazioni quali **nome**, **lista di membri** e **indirizzo multicast**. Inoltre, ogni **card** aggiunta al progetto persiste sul disco come un file JSON a sé stante, che prende il nome della card. Al suo interno si trovano le informazioni relative alla card stessa, quali nome, descrizione, storia dei movimenti e lista in cui si trova attualmente. A titolo di esempio, il file del progetto *esame-reti.json* è così costituito:

```
{
  "name" : "esame-reti",
  "members" : [ "luca" ],
  "multicastIP" : "224.0.0.1",
  "multicastPort" : 45099
}
```

Mentre il file *relazione.json*, che rappresenta la card nominata relazione, risulta così:

```
{
  "name" : "relazione",
  "description" : "Scrivere la relazione del progetto di Reti",
  "history" : "TODO|INPROGRESS|",
  "section" : "INPROGRESS"
}
```

1.2 Connessione TCP

La connessione TCP viene iniziata dai clients, ed è gestita lato server tramite il meccanismo di **Channel Multiplexing** offerto da NIO. La scelta di implementare il server tramite multiplexing dei canali invece che optare per una struttura multithreaded ha come lato positivo una **migliore gestione della concorrenza**, nonché un livello di **leggibilità del codice** decisamente più alto unito infine ad una più snella struttura a runtime, in quanto soltanto un thread si occupa di gestire il traffico in entrata. Dopo aver inizializzato il necessario per fornire la connessione TCP ed il meccanismo di RMI, il server entra in un ciclo e qui rimane in attesa di nuove **keys** in entrata. L'operazione di selezioni delle keys è bloccante, di conseguenza il server rimane in idle finché un evento esterno non modifica questo stato, richiedendo una qualche operazione. Una key corrisponde ad un token, valido fino ad esplicita cancellazione, in cui sono contenuti tutti i riferimenti necessari al server per processare la richiesta, tra cui il **ready set**, che identifica la tipologia di eventi (connessione in ingresso, lettura o scrittura di un messaggio) e l'**attachment**, di fatto un allegato o ancora meglio uno spazio di memorizzazione associato al client. Nel momento in cui arriva una nuova key, si controlla il suo ready set per capire a quale evento il server deve reagire. Gli stati considerati sono:

- *OP_ACCEPT*: richiesta di instaurare una connessione con il server;
- *OP_READ*: sono presenti dati da leggere;
- *OP_WRITE*: sono presenti dati da scrivere.

1.2.1 OP_ACCEPT

Se la chiave trasporta un evento di "**accettazione**", il server instaura una connessione TCP con il client sul `ServerSocket` e lo registra sul selettore precedentemente creato, accettando di fatto un evento di connessione.

1.2.2 OP_READ

Se la chiave trasporta un evento di "**lettura**", il client sta cercando di comunicare con il server inviando un messaggio. Quest'ultimo procede quindi a leggere quanto il client ha inviato, ponendo particolare attenzione a leggere **tutto** il contenuto del messaggio. Viene infatti invocato un metodo, *readRequest*, che grazie ad un *ByteBuffer* e ad un *ciclo do/while*, legge "finché c'è qualcosa da leggere", non rischiando di troncare un messaggio solo perché più lungo del previsto. Il messaggio in ingresso rappresenta una richiesta da parte del client, costituito dal nome utente, che viene sempre inviato in ogni richiesta, seguito da un comando e dagli eventuali parametri necessari. La richiesta viene divisa per distinguere tutte le sue componenti, e proprio in base al comando il server sceglie quale metodo invocare e come passare gli argomenti. Nel codice, questo aspetto è reso più leggibile grazie alla presenza, in ogni ramo case dello

switch applicato sul comando, di commenti che indicano cosa rappresentano le singole componenti della richiesta. Si riporta un esempio, comprensivo dei costrutti di gestione degli errori, discussi subito dopo:

```
case "addCard":
    try{
        // cmd[1] = username, nome utente del proprio account
        // cmd[2] = projectName, nome del progetto
        // cmd[3] = cardName, nome della card
        // cmd[4] = cardDescription, descrizione testuale della card
        addCard(cmd[1],cmd[2],cmd[3],cmd[4]);
        key.attach("ok:Card "+cmd[3]+" created");
    } catch (ForbiddenException fe) {
        key.attach("ko:403:You're not member of this project");
    } catch (ProjectNotFoundException pnfe) {
        key.attach("ko:404:Can't found "+cmd[2]+", are you sure that exists? ... ");
    } catch (CardAlreadyExists cae) {
        key.attach("ko:409:A card with the same name already exists");
    }
    break;
```

Le risposte del server seguono uno stile "standard" e comune a tutti gli endpoints. Il messaggio di risposta ad un'esecuzione andata a buon fine prevede un **"ok"** come primi due caratteri, seguiti dal carattere di due punti (:), utilizzati come separatore, ed infine un **messaggio di successo** che il client visualizzerà a schermo. In caso di errore, i vari metodi che gestiscono i comandi lanciano delle **eccezioni** create ad-hoc, quali ad esempio *AuthenticationFailException* e *UserNotFoundException*. Queste vengono gestite dal server, inviando in risposta al client un messaggio di errore relativo alla situazione, seguendo sempre uno stile comune: i primi due caratteri sono **"ko"**, seguiti dal **separatore**, dal **codice di errore** dell'operazione, da un ulteriore separatore ed infine da un **messaggio di errore** che il client visualizzerà a schermo. I codici di errore sono proprio quelli utilizzati dal protocollo HTTP, a cui è stata data, in alcuni casi, un'interpretazione che potrebbe discostarsi dall'utilizzo che si era pensato originariamente: ad esempio, *404* viene utilizzato per eventi di *contenuto non trovato*, mentre *406*, indicato come *"Not Acceptable"*, viene utilizzato per notificare movimenti non consentiti di una card da una lista ad un'altra. Per dare maggiore coerenza ed aumentare la leggibilità del codice, le risposte ai vari eventi di errore sono stati ordinati rispetto al codice di errore utilizzato, in maniera crescente.

1.2.3 OP_WRITE

Infine, se la chiave trasporta un evento di **"scrittura"** significa che il server ha qualcosa da dire al client. Il messaggio presente nell'allegato della chiave viene inserito in un buffer e successivamente **inviato** proprio al client. Qui viene gestito il caso di disconnessione *"hard"* del client, che non ha seguito le procedure messe a disposizione dell'utente ma invece ha terminato il processo corrispondente (o potrebbe essersi chiuso inaspettatamente in seguito ad una qualche situazione di errore irrecuperabile): la connessione con il client viene quindi chiusa così come il canale ad esso associato. Inoltre, se tramite questo client un utente era collegato al sistema WORTH, il server provvede ad aggiornare il suo stato e a notificare gli utenti online di questo cambiamento.

1.3 Strutture dati

Il server mantiene tre tipologie di dati durante la sua esecuzione:

- *users*: utenti registrati al sistema WORTH;
- *projects*: progetti creati dagli utenti;
- *clients*: oggetti `NotifyEventInterface` per utilizzati comunicare con i clients grazie al meccanismo di RMI callbacks.

La lista degli utenti è stata implementata proprio come una lista, in particolare un **`ArrayList`**. Nonostante sia di fatto una struttura dati molto delicata e potenzialmente soggetta ad eventi di concorrenza, si è rivelata essere una sfida interessante garantirne il corretto funzionamento senza appoggiarsi ad una struttura che implementa da sé meccanismi *thread-safe*. Il metodo *register*, che è l'unico metodo del sistema che può generare eventi di concorrenza sulla lista degli utenti, è stato reso **`synchronized`**, in quanto al suo interno sono presenti operazioni di lettura e scrittura sulla lista che sono sì atomiche ma che agiscono in sequenza, non garantendo da sole la coerenza dei dati. Per maggiore controllo, la stessa sorte è toccata al metodo *getUser*, che è stato reso anch'esso `synchronized` in quanto va ad iterare sulla lista degli utenti, ed una chiamata a questo metodo viene fatta proprio in *register*. Infine, l'essere una lista ha reso molto più semplice la serializzazione in formato JSON, mantenuta sul disco.

Le liste dei progetti e dei clients sono state implementate rispettivamente con una **`LinkedHashMap`** e con una **`ConcurrentHashMap`**. La scelta di usare dei dizionari deriva dalla possibilità di associare al nome utente i relativi progetti e l'interfaccia di notifica eventi, in quanto sia i nomi degli utenti che quelli dei progetti sono univoci nel contesto globale del sistema. Per quanto riguarda i clients, è stata utilizzata una struttura dati *Concurrent*, in quanto usata dai metodi critici *registerCallback* e *unregisterCallback*.

1.4 Endpoints


Per ogni **comando** messo a disposizione dei clients è stato creato un metodo apposito, con i suoi parametri e le eccezioni che può lanciare in caso di errore. In questi metodi vengono fatti tutti i **controlli** necessari sugli argomenti, sull'esistenza dei dati prima di accedervi oppure sulla loro assenza prima di inserirli nelle strutture dati del server. Per i dettagli su ognuno di essi si rimanda al codice.

1.5 Utils

Sono infine presenti alcuni metodi **ausiliari**, utilizzati internamente dal server, tra qui troviamo il salvataggio su disco delle informazioni multicast, la formattazione dell'ora corrente per poter inviare messaggi nelle chat dei progetti, e notificare eventi ad essi relativi (p.e. spostamento di cards).

2 Client

ClientMain è la classe che definisce la struttura dei clients che utilizzano i servizi del sistema WORTH. Estende anch'essa **RemoteObject** ed implementa **NotifyEventInterface**, per permettere la ricezione di notifiche da parte del server. All'avvio il client prova ad instaurare una connessione TCP con il server: se questa fallisce, perché ad esempio il server non è raggiungibile, il processo informa l'utente dell'accaduto e termina immediatamente. Se il tentativo di connessione va a buon fine viene instaurata anche una connessione RMI. Il client informa quindi l'utente che per accedere a tutti i servizi offerti da WORTH è necessario effettuare il **login**, senza il quale sono permesse soltanto le operazioni di registrazione e, appunto, login. La visualizzazione dell'input viene resa graficamente attraverso la tipica impostazione di una qualsiasi **console** di sistema. L'utente può quindi decidere di creare un nuovo account sul sistema WORTH e, se l'operazione va a buon fine, il login viene effettuato automaticamente e l'utente può finalmente utilizzare tutte le fantastiche funzionalità di WORTH!



```

  W O R T H

Connection attempt, wait...

You need to login to WORTH in order to use it. Here some commands:
  register <username> <password> | Create a new WORTH account;
  login   <username> <password> | Login to WORTH using your credentials;
  help                                         | Show this help;
  quit                                         | Close WORTH.

Guest@WORTH > register luca p4ssw0rd
Signup was successful!
I try to automatically login to WORTH, wait..
Now you're logged as luca!
luca@WORTH >
```

Figure 2.1: Schermata di avvio del client, registrazione e login automatico di un utente

2.1 Strutture dati

Il client mantiene nella memoria locale alcuni dati, prevalentemente utilizzati per la gestione delle chat e dei relativi messaggi:

- *usersStatus*: utenti del sistema e relativo stato;
- *chatListeners*: riferimenti ai threads in ascolto sulle chat dei progetti;
- *projectsMulticast*: tasks contenenti il necessario per gestire connessioni multicast;
- *chats*: code per ogni chat in cui vengono salvati i messaggi.

La lista **usersStatus** è implementata come una **LinkedList** di stringhe, sincronizzata grazie al metodo *synchronizedList*. Questo garantisce un'implicita gestione della concorrenza in operazioni di inserimento e cancellazione, quindi durante l'aggiornamento da parte del server, ma non in fase di accesso, dove la lista deve essere esplicitamente posta all'interno di un blocco *synchronized* (si veda l'implementazione di *listUsers* e *listOnlineUsers*). L'elemento tipico è appunto una stringa nel formato "*ONLINE:nomeUtente*" e consente di sapere quali utenti si trovano attualmente collegati in WORTH, e quali invece risultano offline. La visualizzazione di queste informazioni, richiamate dai comandi *listUsers* e *listOnlineUsers*, che agiscono in locale senza mai contattare il server, è resa visivamente più efficace grazie all'utilizzo di due simboli ASCII con cui indicare i due possibili stati degli utenti. Si riporta un esempio:

```
luca@WORTH > listUsers
● luca
○ mario
○ sofia
○ paolo
● giulia
luca@WORTH > listOnlineUsers
● luca
● giulia
luca@WORTH > |
```

Figure 2.2: Output dei comandi *listUsers* e *listOnlineUsers*

L'aggiornamento dello stato degli utenti non viene mai richiesto esplicitamente dal client ma è il server che si occupa di aggiornare questa lista tramite **callback**, solamente nel caso in cui si verifica un evento che ne muta lo stato (un utente effettua il login e quindi il suo stato passa da offline ad online, un nuovo utente si registra...). Le rimanenti strutture dati sono utilizzate per la gestione delle chat dei progetti e dei messaggi che su queste vengono scambiati. Sono tutte dei dizionari, implementate come **HashMap** oppure come **LinkedHashMap**, e si appoggiano ad una **ThreadPool** per recuperare i messaggi inviati. Per una trattazione più specifica si rimanda alla sezione dedicata alle chat di progetto.

2.2 Comandi

Il client distingue il comando inserito dall'utente allo stesso modo del server. In base a questo, chiama il metodo corrispondente passandogli i parametri necessari. Questi vengono controllati e in caso di inconsistenze viene visualizzato un messaggio di errore. Se l'utente non inserisce il numero di parametri richiesti per l'esecuzione del comando, il metodo ovviamente non viene invocato e viene invece visualizzato un messaggio di *Usage* per quel comando specifico e, se disponibili, anche dei consigli per usare al meglio quel comando. Ai fini di una maggiore leggibilità del codice, anche nel client i parametri passati in input dall'utente sono stati commentati e descritti prima di invocare il relativo metodo, le eccezioni sono sempre poste nello stesso ordine tra i vari comandi, e questi sono a loro volta posti nello stesso ordine presente sul server. Anche sul client sono stati implementati gli stessi meccanismi di lettura della risposta del server, per scongiurare la possibilità di troncare un messaggio solo perché molto lungo.

2.3 Chat

Il sistema di gestione delle **chat di progetto** è la parte più delicata ma anche più sofisticata del client. A grandi linee, si è pensato di porre **un thread in ascolto su ogni chat di progetto**, che mantiene in una coda tutti i messaggi in arrivo da altri utenti (o anche dal server, che può notificare sulla chat eventi

riguardanti il progetto stesso) e che l'utente può leggere in qualsiasi momento. Nel seguito viene spiegato nel dettaglio il funzionamento di questo meccanismo, attraverso un approccio "bottom-up" partendo dalla struttura dati che mantiene i messaggi in memoria fino alla ThreadPool che gestisce invece i threads.

2.3.1 Coda dei messaggi

La struttura dati più adatta a salvare i **messaggi** in ingresso è di fatto una **coda**, essendo una struttura **FIFO** che permette una lettura ordinata partendo dal messaggio che da più tempo si trova nella coda fino a quello inserito più di recente: i nuovi messaggi vengono inseriti in fondo alla coda, mentre la lettura comincia proprio dalla testa. Definita l'interfaccia **Queue**, l'implementazione utilizzata è una **ConcurrentLinkedQueue**. Le motivazioni sono molteplici: è una struttura **unbounded**, che di conseguenza non pone limiti teorici sulla quantità di messaggi che possono essere immagazzinati e non richiede di specificare una capacità massima iniziale. E' **thread-safe**, e questo è molto importante in quanto la coda dei messaggi viene utilizzata sia dal thread che rimane in ascolto, sia dal thread *Main* che va a leggere i messaggi quando l'utente lo richiede; per questo motivo deve essere garantito un accesso concorrente sicuro che non interferisca con le fasi di lettura e scrittura da parte di threads diversi. L'essere una struttura di "linked nodes" vuol dire che ogni messaggio è di fatto collegato al successivo, mantenendo un ordine sequenziale tra messaggi e permettendo un accesso efficiente e ordinato in fase di lettura. L'implementazione "under the hood" di questa struttura è garantita essere wait-free, per cui ogni metodo ritorna immediatamente un risultato, senza attese. I principali metodi utilizzati sono: **add**, per inserire in coda nuovi messaggi, **poll**, per recuperare e contestualmente rimuove il messaggio che si trova in testa e **isEmpty**, per continuare ad eseguire il metodo poll fino a che non ci sono più messaggi da leggere. Come intuibile, una volta letti, i messaggi vengono rimossi dalla coda e non più accessibili: questo comportamento deriva da una scelta personale di implementazione dovuto all'assenza di un controllo centralizzato dei messaggi. Tuttavia può essere facilmente esteso attraverso un puntatore all'ultimo messaggio letto dall'utente e da cui ripartire in una successiva fase di lettura, senza rimuoverli dalla coda. Infine, i messaggi vengono rappresentati attraverso delle stringhe, formattate sia dal client che dal server per contenere non solo il contenuto del messaggio ma anche il nome del mittente e l'ora di invio.

```
Luca@WORTH > readChat esameReti
(18:59) luca: Quindi questa e' la chat del progetto!
(18:59) giulia: Si! Possiamo comunicare rimanendo su WORTH
(18:59) WORTH: giulia has moved card Chat from INPROGRESS to DONE
Luca@WORTH >
```

Figure 2.3: Scambio di messaggi sulla chat tra due membri di un progetto, con notifica dal server

2.3.2 Chat listeners

ChatListener è una classe che estende **Runnable** e si configura di conseguenza come un task eseguibile da un qualsiasi thread. Come suggerisce il nome, permette ad un thread di mettersi **in ascolto** sulla configurazione multicast della chat di un progetto e salvare nella propria coda i messaggi in entrata. Ogni task salva localmente i dati relativi alla configurazione multicast di quel particolare progetto, quali indirizzo IP e porta che usa poi per ricevere gli aggiornamenti (successivamente verrà discusso il modo in cui il client recupera queste informazioni), e la propria ConcurrentLinkedQueue precedentemente discussa.

Il metodo **run**, che definisce il comportamento del thread, va sostanzialmente ad aprire una **MulticastSocket** sulla porta specificata, "entra" nel gruppo multicast definito dall'indirizzo IP e si pone in uno stato idle, **in attesa** di nuovi messaggi da recuperare tramite la funzione **receive**, che è di fatto bloccante. La dimensione del pacchetto che andrà a contenere il messaggio è fissa per questioni di semplicità: risulta però facilmente estendibile inviando prima di ogni messaggio la sua dimensione, ed allocando un **Data-gramPacket** proprio della dimensione del prossimo messaggio, risultando più efficiente in termini di memoria occupata. Il thread può essere **terminato** dall'esterno grazie ad un riferimento alla Future generata al momento dell'esecuzione del task, nel caso in cui il progetto venga cancellato e la chat non debba più essere disponibile.

2.3.3 ThreadPool

Per una migliore e più efficiente gestione dei threads listener è stata utilizzata una ThreadPool, in particolare una **CachedThreadPool**. La scelta di utilizzare proprio quest'ultima implementazione è motivata principalmente dal fatto che permette di creare threads tanti quanti se ne richiede: non viene infatti specificato un numero massimo di core threads ma ne garantisce un numero indefinito a priori, nonché il riuso di quelli precedentemente costruiti e disponibili al momento di nuove **submit**. Inoltre, la terminazione dei threads che non stanno più svolgendo la propria attività dopo lo scadere di un timeout di sessanta secondi garantisce che la pool non consumi risorse quando non ci sono listeners attivi. Gestire dei "raw" threads, senza l'ausilio di una pool avrebbe permesso di maneggiarne lo stato con più facilità, ma lo stesso obiettivo è stato raggiunto anche attraverso la ThreadPool, grazie all'uso delle Future. La creazione di un nuovo thread listener procede per step:

1. viene creata la coda dei messaggi, ed un riferimento è salvato nel dizionario *chats*;
2. un nuovo task ChatListener viene creato e salvato poi nel dizionario *projectsMulticast*, utilizzando i dati di multicast e la coda dei messaggi;
3. il task viene sottomesso alla pool che ritorna un oggetto Future relativo a quel listener, e che viene salvato nel dizionario *chatListeners*;

Attraverso il riferimento **Future** relativo ad ogni ChatListener, è possibile controllare lo stato del thread ed eventualmente interromperlo, qualora il progetto venisse cancellato e non fosse più possibile ricevere ed inviare messaggi sulla sua chat. Questo grazie al metodo **cancel** offerto dall'interfaccia Future, che prova a cancellare il task dalla coda della pool se non è stato ancora eseguito, oppure se il parametro booleano *mayInterruptIfRunning* del metodo ha valore *true* si tenta di interrompere il thread anche se è già stato eseguito. Essendo il thread, generalmente, in uno stato di idle ma bloccato sulla receive, viene "sbloccato" inviando un messaggio di shutdown, che porterà il thread a controllare se il suo stato è passato su "interrotto" grazie al metodo **isInterrupted**, e in caso affermativo la MulticastSocket viene chiusa e il thread termina, rilasciando le risorse.

2.3.4 Recupero delle informazioni di multicast

Le informazioni necessarie all'utilizzo delle chat di progetto sono generate lato server al momento della creazione dello stesso. Queste informazioni persistono sul server ma devono essere in qualche modo comunicate ai clients che ne hanno bisogno. Si è deciso tuttavia, essendo l'invio e la ricezione dei messaggi un evento gestito interamente da una connessione multicast in cui non viene coinvolto il server, **di ridurre al minimo la dipendenza da quest'ultimo**. Contestualmente al login ogni client riceverà le informazioni multicast per ogni progetto di cui l'utente è membro, che verranno immediatamente salvate

in locale nelle rispettive strutture dati e utilizzate per la creazione dei threads listener, che subito dopo entreranno in esecuzione per salvare, da questo momento fino al logout o alla chiusura dell'applicazione, tutti i messaggi che arriveranno in quella chat. Nel caso in cui l'utente venga aggiunto come membro di un progetto da un altro utente, le informazioni di multicast verranno inviate dal server contestualmente ad una **notifica**, senza alcuna azione da parte del client. Quest'ultimo provvederà alla creazione del rispettivo thread listener, nonché ad avvisare l'utente di essere stato aggiunto al *progetto P* da parte dell'utente *U*. Un meccanismo simile viene messo in moto quando uno dei membri di un progetto ne richiede la cancellazione: se i criteri per procedere sono rispettati, il server cancella il progetto e **notifica tutti i membri** dell'accaduto, ed i clients potranno agire di conseguenza interrompendo il thread listener di quel progetto in quanto, di fatto, non esiste più la chat corrispondente.

3 User

La classe **User** identifica un utente del sistema WORTH. Questo possiede uno **username**, che lo riconosce *univocamente* nel sistema, una **password**, scelta al momento della registrazione ed utilizzata ad ogni accesso per offrire un meccanismo di autenticazione, un **salt**, usato per l'hashing della password e completamente trasparente all'utente, ed infine uno **status**, ovvero lo stato dell'utente che può essere Online oppure Offline. Al momento della registrazione, l'utente ancora ospite deve fornire un nome utente, di cui ne sarà controllata l'univocità nel sistema, ed una password di accesso. Questa verrà trattata dal server con la massima **confidenzialità**. In primo luogo viene generato ed associato all'utente un salt, attraverso un generatore **crittograficamente sicuro**; questo diventa un parametro della funzione di hashing che sfrutta l'algoritmo **SHA-256** per calcolare un'immagine one-way a 256 bit della password "mischiata" al salt. Solo a questo punto, l'hash della password verrà salvato sul disco ed utilizzato in tutti i tentativi di accesso successivi. Il processo di autenticazione prevede di effettuare lo stesso procedimento di hashing sulla password fornita al login, e **confrontare** il risultato ottenuto con l'impronta della password scelta originariamente.

4 Project

Un **progetto** ha un **nome**, scelto dall'utente e *univoco* nel contesto globale del sistema, ed una **lista di membri** che ne fanno parte. L'utente che crea il progetto ne diventa automaticamente membro. Inoltre, mantiene **quattro liste** separate, una per ogni sezione in cui una card può trovarsi in un certo istante di tempo. Seguono poi le informazioni di **multicast**, indirizzo IP e porta. Le liste sono state implementate, appunto, come liste ed in particolare come **ArrayList**. La classe mette a disposizione metodi per interagire con il progetto e le sue componenti appena illustrate. Oltre ai vari getters/setters, sono presenti metodi ausiliari, tra cui **isMember**, per verificare se un utente del sistema WORTH è membro di quel particolare progetto, o ancora **getMulticastInfo**, che ritorna le informazioni multicast insieme al nome del progetto, in un formato specifico e utilizzato per la trasmissione di queste informazioni verso i clients. Degno di nota è il metodo **moveCard**, che permette di spostare una card dalla lista in cui si trova attualmente ad un'altra: qui vengono fatti i controlli necessari a impedire spostamenti non permessi, come specificato dal seguente schema:

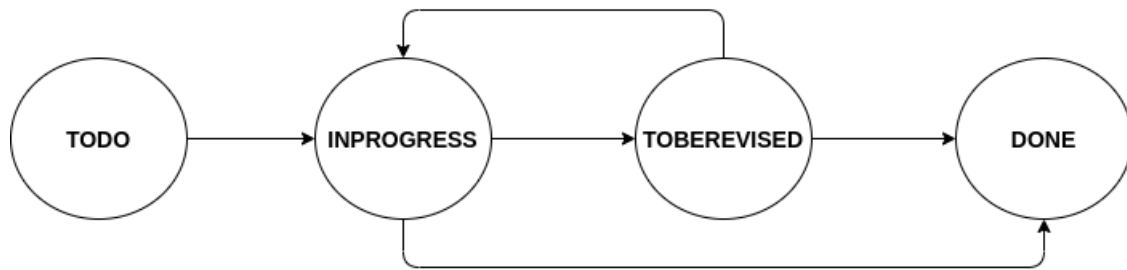


Figure 2.4: Vincoli per lo spostamento di una card da una lista ad un'altra

Ad ogni movimento, viene aggiornata anche **la storia dei movimenti** della card interessata e il server invia un messaggio di notifica sulla chat del progetto. Inoltre, sono state specificate eccezioni personalizzate che identificano possibili eventi di errore, quali ad esempio *CardNotFoundException* oppure *IllegalCardMovementException*. La struttura del progetto persiste sul sistema con una directory omonima, posta all'interno della parent-directory "Projects/", che contiene il file di configurazione JSON e tutte le cards aggiunte.

5 Card

La struttura di una **card** è di per sé molto semplice: ha un **nome**, *univoco* nel contesto del progetto di cui fa parte, una **descrizione**, una **storia di movimenti** ed un riferimento alla **lista** in cui si trova attualmente. Offre, oltre a getters/setters ovvi, un metodo per l'aggiornamento della sezione attuale che si riflette anche sulla storia dei movimenti, nonché la possibilità di restituire le informazioni della card attraverso un override del metodo **toString**. Persiste sul sistema in un file JSON contenente le informazioni appena elencate.

6 Multicast

La gestione della struttura multicast del sistema è affidata proprio a questa classe, che porta in memoria il contenuto presente nel corrispondente file JSON. Trovano spazio **l'ultimo indirizzo IP assegnato**, in modo da avere un riferimento per la generazione di nuovi indirizzi, e una lista di indirizzi **rilasciati**, ovvero precedentemente assegnati a progetti che sono stati poi cancellati. La generazione di un nuovo indirizzo è **incrementale**, e sfrutta tutto il range disponibile di indirizzi multicast della classe D, ovvero dal 224.0.0.1 al 239.255.255.255. E' stato implementato un meccanismo di riuso ma non uno di **reset**. Questo perché, appunto, essendo presente il riuso degli indirizzi IP, se il sistema raggiunge l'ultimo indirizzo disponibile vuol dire che tutti quelli precedenti sono ancora in uso, ed un reset totale andrebbe a sovrascrivere e a generare conflitti. In un contesto reale, un caso estremo come questo potrebbe essere scongiurato revocando gli indirizzi IP dei progetti che non vengono utilizzati per più di un certo quantitativo di tempo.

7 Funzionalità aggiuntive

Nonostante si rimandi continuamente al codice per curiosare le implementazioni, sono riportate qui alcune funzionalità interessanti del sistema WORTH.

7.0.1 moveCard shortcut

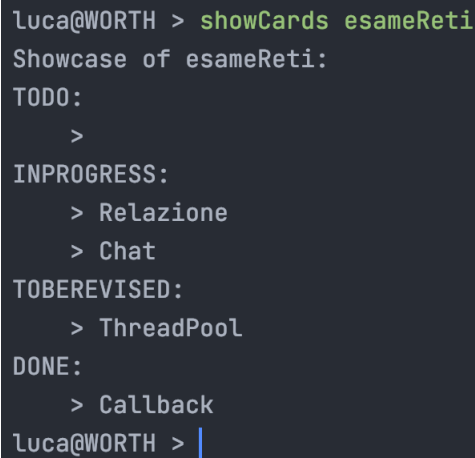
Il comando **moveCard** permette di spostare una card dalla lista in cui si trova attualmente in un'altra lista, purché l'operazione rispetti determinati criteri. Tra tutti quelli disponibili, questo comando risulta il più verboso, in quanto richiede il comando in sé, il nome del progetto, il nome della card, la lista di partenza e quella di destinazione. E' stata aggiunta la possibilità di utilizzare i **numeri** da 1 a 4 per identificare le liste, invece che il loro nome per esteso. I due comandi seguenti, ad esempio, sono del tutto equivalenti:

```
moveCard esameReti relazione todo inprogress
moveCard esameReti relazione 1 2
```

Un'altra possibile scorciatoia, atta a ridurre il numero di argomenti necessari al comando, potrebbe prevedere il caso in cui venga specificata una sola lista: si potrebbe quindi assumere che quella mancante sia la lista di partenza e quella esplicitata invece sia quella di destinazione. La card verrebbe dunque cercata tra tutte le quattro liste e, come di consueto, ne verrebbero controllati i vincoli prima di effettuare lo spostamento nella nuova lista.

7.0.2 showCards tabellare

E' possibile visualizzare le cards del progetto in due modalità: **lista** o **tabella**. La prima prevede una struttura **verticale**, che riporta quindi le liste dalla prima all'ultima, con le relative cards.



```
Luca@WORTH > showCards esameReti
Showcase of esameReti:
TODO:
  >
INPROGRESS:
  > Relazione
  > Chat
TOBEREVISED:
  > ThreadPool
DONE:
  > Callback
Luca@WORTH > |
```

Figure 2.5: Visualizzazione di tutte le cards di un progetto in modalità lineare

La modalità di visualizzazione **tabellare** prevede invece una struttura orizzontale, ed è attivabile aggiungendo il parametro "*table*" al comando showCards, che seguirà ovviamente il nome del progetto.

```
showCards esameReti // Modalita' lineare
showCards esameReti table // Modalita' tabellare
```

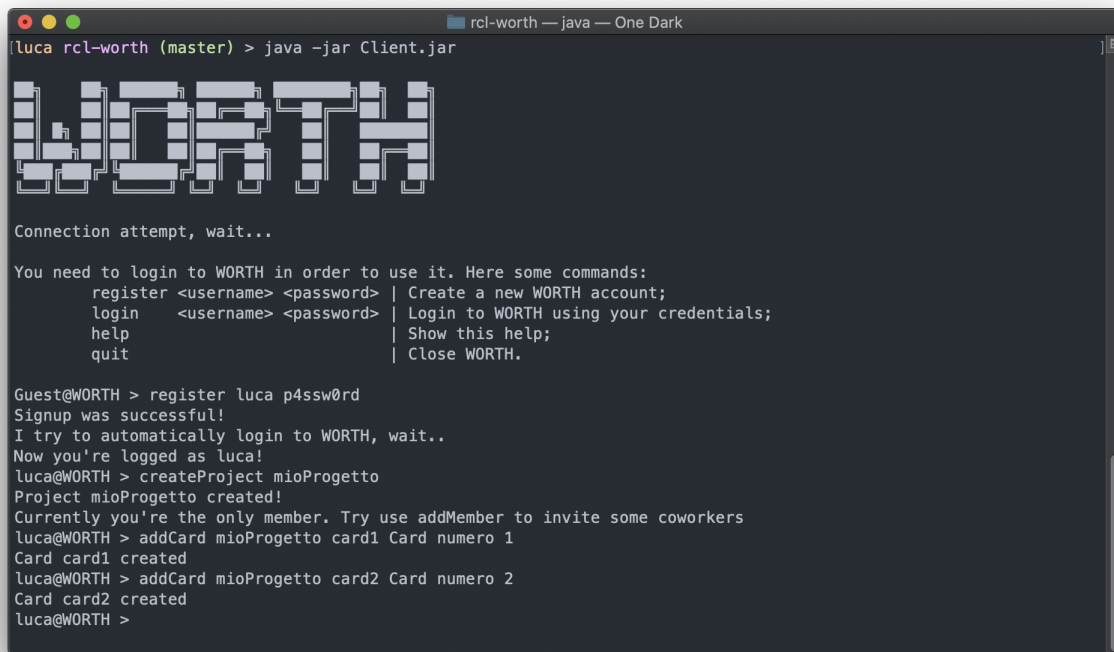


```
luca@WORTH > showCards esameReti table
Showcase of esameReti:
|          TODO          |      INPROGRESS      |      TOBERESIVED      |      DONE      |
|                        | Relazione            | ThreadPool             | Callback        |
|                        | Chat                 |                        |                 |
luca@WORTH > |
```

Figure 2.6: Visualizzazione di tutte le cards di un progetto in modalità tabellare

3. Screenshots

Si riportano alcune immagini del sistema in esecuzione:



```
luca rcl-worth (master) > java -jar Client.jar

WORTH

Connection attempt, wait...

You need to login to WORTH in order to use it. Here some commands:
  register <username> <password> | Create a new WORTH account;
  login    <username> <password> | Login to WORTH using your credentials;
  help     | Show this help;
  quit     | Close WORTH.

Guest@WORTH > register luca p4ssw0rd
Signup was successful!
I try to automatically login to WORTH, wait..
Now you're logged as luca!
luca@WORTH > createProject mioProgetto
Project mioProgetto created!
Currently you're the only member. Try use addMember to invite some coworkers
luca@WORTH > addCard mioProgetto card1 Card numero 1
Card card1 created
luca@WORTH > addCard mioProgetto card2 Card numero 2
Card card2 created
luca@WORTH >
```

```
luca@WORTH > listProjects
These are the projects you are a member of:
- mioProgetto
luca@WORTH > showCards mioProgetto
Showcase of mioProgetto:
TODO:
    > card1
    > card2
INPROGRESS:
    >
TOBEREVED:
    >
DONE:
    >
luca@WORTH > showCards mioProgetto table
Showcase of mioProgetto:
|      TODO      |      INPROGRESS      |      TOBERESIVED      |      DONE      | | | |
| card1          | |                     | |                     | |               |
| card2          | |                     | |                     | |               |
luca@WORTH > showCard mioProgetto card1
Some information about the card1 card:
- Name > card1
- Description > card1
- Currently in > TODO
luca@WORTH > moveCard mioProgetto card1 TODO INPROGRESS
card1 moved from TODO to INPROGRESS
luca@WORTH > moveCard mioProgetto card1 2 4
card1 moved from 2 to 4
luca@WORTH >
```

```
luca@WORTH > listUsers
• luca
• paolo
luca@WORTH >
Ding! You have been added to the project GamblingOnBitcoin of paolo
luca@WORTH > listProjects
These are the projects you are a member of:
- mioProgetto
- GamblingOnBitcoin
luca@WORTH > sendChatMsg GamblingOnBitcoin Dobbiamo fare un sacco di soldi!
Message sent!
luca@WORTH > readChat GamblingOnBitcoin
(19:20) luca: Dobbiamo fare un sacco di soldi!
(19:21) paolo: Vedremo!!!
luca@WORTH > addCard GamblingOnBitcoin fareSoldi Ma... come?
Card fareSoldi created
luca@WORTH > showCards GamblingOnBitcoin table
Showcase of GamblingOnBitcoin:
|      TODO      |      INPROGRESS      |      TOBERESIVED      |      DONE      |
| fareSoldi      | |                     | |                     | |               |
luca@WORTH > moveCard GamblingOnBitcoin fareSoldi TODO INPROGRESS
fareSoldi moved from TODO to INPROGRESS
luca@WORTH > moveCard GamblingOnBitcoin fareSoldi 2 4
fareSoldi moved from 2 to 4
luca@WORTH > showCards GamblingOnBitcoin table
Showcase of GamblingOnBitcoin:
|      TODO      |      INPROGRESS      |      TOBERESIVED      |      DONE      |
|               | |                     | |                     | | fareSoldi     |
luca@WORTH >
```

```
rcl-worth — java — One Dark
luca@WORTH > showCards GamblingOnBitcoin table
Showcase of GamblingOnBitcoin:
|          TODO          |          INPROGRESS          |          TOBERESIVED          |          DONE          |
|          |          |          |          fareSoldi          |          |
luca@WORTH > Ding! paolo has canceled the project GamblingOnBitcoin
luca@WORTH > listProjects
These are the projects you are a member of:
- mioProgetto
luca@WORTH > showCards GamblingOnBitcoin
Can't found GamblingOnBitcoin, are you sure that exists? Try createProject to create a project
luca@WORTH > showMembers mioProgetto
These are the members of the project mioProgetto:
- luca
luca@WORTH > listOnlineUsers
● luca
● paolo
luca@WORTH >
```

```
rcl-worth — -bash — One Dark
luca@WORTH > help
createProject <projectName>          | Create a new project;
addMember <projectName> <memberUsername> | Add a new member in a project;
showMembers <projectName>          | Shows the current members of a project;
addCard <projectName> <cardName> <cardDesc> | Create and assign a new card to a project;
showCard <projectName> <cardName>          | Shows information about a card assigned to a project;
showCards <projectName> [table]          | Shows all cards assigned to a project;
moveCard <projectName> <cardName> <from> <to> | Move a card from a list to another;
getCardHistory <projectName> <cardName>      | Shows the history of a card;
readChat <projectName>                  | Read message sent in the project chat;
sendChatMsg <projectName> <message>          | Send a message in the project chat;
cancelProject <projectName>              | Cancel a project (Warning, it's not reversible);
listUsers                               | List all WORTH users and their status;
listOnlineUsers                         | List only users that are currently online;
logout                                  | Logout from your WORTH account;
help                                    | Show this help;
quit                                    | Logout and close WORTH.

luca@WORTH > logout
You have been logged out successfully
Guest@WORTH > quit
Hope to see you soon!
luca rcl-worth (master) >
```