RELAZIONE PROGETTO “WORTH”, Laboratorio di Reti A a.a. 2020/2021

Gabriele Masciotti 578318

In questo elaborato riporto una breve relazione sul progetto assegnato, sviluppata partendo da una presentazione del contenuto dell’archivio consegnato, fino alla descrizione dettagliata delle fasi di esecuzione dei programmi in oggetto.
Per favorire la leggibilità organizzerò lo scritto in sezioni.

1. CONTENUTO DELL’ARCHIVIO COMPRESSO

Il contenuto dell’archivio consegnato costituisce tutto il necessario per poter eseguire il programma oggetto del progetto. All’interno della cartella “*WORTH_server*” sono presenti i file con il codice che implementa, appunto, il server del servizio; in particolare troverete:

- file “***Card.java***”, contenente l’implementazione del tipo “card”, i cui oggetti sono caratterizzati da un nome, una descrizione, un nome di progetto a cui la card appartiene, una storia (successione di liste visitate dalla card) e un nome utente creatore, con tanto di metodi utili per ottenere informazioni sulle card di progetto (tra i quali particolarmente utili sono il metodo per reperire la lista in cui una card si trova in un dato istante, l’autore della card, la sua storia);
- file “***Client_UserUpdates_Interface.java***”, contenente l’interfaccia del programma client utilizzata dal server per chiamare i metodi di aggiornamento con callback delle strutture dati locali del client (per dettagli su questi aggiornamenti vedere sezioni successive);
- file “***Project.java***”, contenente la definizione del tipo “progetto”. Un progetto WORTH è identificato univocamente da un nome, possiede una lista di membri, quattro liste in cui sono distribuiti oggetti di tipo card in base alla fase di lavorazione da parte degli utenti in cui le card si trovano, un indirizzo multicast utilizzato per il servizio di chat, un flag booleano che indica se il progetto è in eliminazione oppure no (modificabile oppure no) e una lista di utenti che hanno inviato delle richieste di eliminazione (se presenti). I primi tre metodi (la cui parola chiave è “persistent”) sono utilizzati al momento dell’attivazione del server (all’inizio della sua esecuzione) per ricostruire lo stato del sistema leggendo i file memorizzati nel file system (per maggiori dettagli vedere sezioni successive). Tra gli altri metodi, quelli di maggior rilevanza sono senz’altro quello che restituisce la lista di tutte le card di progetto, il metodo che controlla se uno spostamento di una card può essere effettuato (cioè se rispetta i vincoli dettati nella specifica), il metodo che controlla se il progetto può essere eliminato (se tutte le card sono nella lista “done”) e quelli che permettono di inserire/rimuovere un membro e aggiungere/annullare una richiesta di eliminazione.;
- file “***Registration_Interface.java***”, che costituisce l’interfaccia in cui sono definiti i metodi dello stub RMI di registrazione esportato dal server (condivisa con i client), i quali consentono agli utenti di registrarsi al servizio con le proprie credenziali e ai programmi client di iscriversi (e disiscriversi) per ricevere callback di aggiornamento dal server;
- file “***Registration_Service.java***”, che implementa l’interfaccia precedente e costituisce l’oggetto con cui il server gestisce gli iscritti al servizio WORTH. Tra le variabili di istanza troviamo un object mapper utilizzato per le scritture su file (per rendere persistenti le informazioni sul file system), un “data base” (hash table di nomi utente e password) di utenti registrati al sistema, un “data base” (hash table di nomi utente e stati) contenente i nomi degli utenti e i loro stati nel sistema (online o offline), l’oggetto di tipo File con il riferimento al file contenente la lista persistente di membri, una lista di interfacce di clients che devono essere notificati con callback e un produttore di immagini hash (utilizzato per proteggere le password da accessi non autorizzati al file degli utenti registrati). I dettagli relativi ai metodi definiti in questa classe sono forniti più tardi; concentrerei invece l’attenzione sul costruttore, il quale inizializza il produttore di immagini hash utilizzando

l'algoritmo "SHA-1" e tenta di creare il file degli utenti iscritti. Se non riesce (il file esiste già) lo legge utilizzando l'object mapper e inizializza il data base di utenti e di stati (che saranno tutti impostati ad offline all'inizio).;

- file "***setConnectionParameters.java***", che definisce la finestra iniziale in cui poter scegliere i parametri di connessione del server;
- file "***WORTH_Server_main.java***", in cui è implementato il programma server.

All'interno della cartella "*WORTH_client*" sono presenti i file con il codice che implementa, appunto, il client con cui si accede al servizio; in particolare troverete:

- file "***Chats.java***", in cui è definito il tipo Chats, i cui oggetti costituiscono insiemi di chat di progetto di cui gli utenti sono membri. Ogni client, all'inizio della sua esecuzione, crea un proprio oggetto di tipo Chats e lo utilizza per gestire tutte le chat da cui l'utente vuole ricevere i messaggi. In questo oggetto sono memorizzati: una lista di indirizzi multicast (uno per ogni chat), una lista di messaggi ricevuti e inviati (una per ogni chat) che saranno persistiti fino al termine dell'esecuzione del programma, e due liste (chatAdded e chatRemoved) utilizzate per sincronizzare l'esecuzione tra il thread main e il thread di ricezione dei messaggi. Questo oggetto è infatti condiviso con il thread di cui al punto successivo;
- file "***ChatMessageReceiver.java***", contenente il task runnable che viene eseguito dal thread di ricezione dei messaggi. Come descritto in seguito infatti, il client attiva un thread che si occupa di ricevere, e mostrare all'utente, i messaggi provenienti da tutte le chat di progetto di cui è membro;
- file "***Registration_Interface.java***", interfaccia dello stub RMI esportato dal server e utilizzato dal client per richiedere le operazioni di registrazione di nuovi utenti e iscrizione al servizio di notifica callback (dettagli forniti in seguito);
- file "***Client_UserUpdates_Interface.java***", interfaccia esportata dal client, contenente i metodi che il server invoca quando deve effettuare le callback. Tali metodi sono responsabili dell'aggiornamento delle strutture dati locali al client, come descritto al punto successivo;
- file "***UserUpdates_ROC.java***", oggetto remoto che implementa l'interfaccia del punto precedente e, come si evince dal nome, viene utilizzato per gestire ed aggiornare le strutture dati locali del client. Le variabili di istanza di questo tipo di dato includono: la lista di tutti gli utenti registrati al servizio e il loro stato (lista restituita dal server al momento della registrazione per callback da parte del client, come descritto in seguito), una lista di membri di progetto e il loro stato, utilizzata per mostrare la situazione in tempo reale dei membri del progetto di cui l'utente ha la finestra della chat aperta al momento, ed un riferimento a quest'ultima (come detto al punto dedicato alle finestre della GUI). Tutti i metodi definiti in questa classe saranno descritti nei dettagli in seguito.;
- file "***waitUserAction.java***", un monitor necessario per attendere in attesa passiva gli input dell'utente nelle finestre della GUI. Il tipo contiene soltanto un campo booleano il quale costituisce un flag che indica se un utente ha premuto un pulsante in una finestra e quindi il programma può effettuare il relativo task. Il client, all'inizio della sua esecuzione, crea un oggetto di questo tipo e lo condivide con tutte le finestre che verranno attivate nel corso dell'utilizzo del programma da parte dell'utente; queste ultime utilizzeranno il metodo "setUserDone" per segnalare al thread main, in attesa, che l'utente ha effettuato la sua scelta;
- file "***WORTH_Client_main.java***", in cui è implementato il programma del client;
- tutti gli altri file contengono il codice delle finestre con cui è realizzata la GUI per questo progetto. In particolare il programma utilizza:
  - una finestra di partenza definita in "***BeforeWeStart.java***", che consente di scegliere il metodo di accesso al servizio WORTH (come specificato in seguito infatti, il client può essere collegato ad un server che è in esecuzione sulla sua stessa macchina oppure in un'altra, collegata alla stessa rete);

- una finestra di accesso (“***Access_Frame.java***”) che semplicemente si serve di due Text Fields per raccogliere le credenziali dell’utente al momento dell’accesso al servizio;
- una finestra di home page del servizio WORTH (“***Home_Page_Frame.java***”) con cui l’utente interagisce per richiedere i servizi messi a disposizione (creare un progetto, visualizzare il menù di un progetto, visualizzare lista di utenti e progetti di cui è membro, …);
- una finestra di impostazioni del profilo utente definita nel file “***Profile_Settings.java***”, in cui l’utente, oltre a visualizzare informazioni generiche sul suo stato nel sistema, ha la possibilità di: disattivare le notifiche relative ai messaggi di chat (per ulteriori dettagli vedere sezioni successive della relazione), modificare la propria password di accesso, eliminare definitivamente il proprio account.;
- una finestra che visualizza il menù di un progetto (di cui l’utente è membro ovviamente) il cui codice è contenuto in “***ProjectMenu.java***”, la quale organizza una serie di pulsanti e aree di testo con cui sono implementate tutte le funzioni richieste da specifica (creare, card, visualizzare i dettagli e la storia delle card, spostare le card di lista in lista, …). Funzionalità aggiuntive (rispetto a quanto richiesto nella specifica) sono l’eliminazione di una card di progetto non più necessaria e la possibilità di visualizzare una *lavagna kanban*, per avere un’immagine della situazione attuale del progetto su cui i membri stanno lavorando.;
- dalla finestra del menù di progetto si accede a quella della chat (“***ChatFrame.java***”), ultima non per importanza, in quanto si occupa di mostrare all’utente sia la chat associata al progetto, sia la situazione in tempo reale dei membri del progetto nel sistema (sfruttando aggiornamenti callback del server, come descritto in seguito nei dettagli).;
- tutti gli altri file contengono il codice delle finestre ausiliarie (su cui non è importante soffermarsi molto) che sono presentate all’utente nel momento in cui il programma ha la necessità di avvertirlo di qualcosa di imminente o di accertarsi che abbia realmente intensione di eseguire l’azione richiesta.

** *Tutti i file di implementazione delle suddette finestre contengono codice generato automaticamente dal sistema IDE, su cui invito a non soffermarsi con attenzione perché di relativa importanza. Questo discorso non vale per la finestra della chat e dei membri, di cui tratteremo nei dettagli più avanti*.

Infine troverete la cartella “***icone***” in cui sono posizionate tutte le icone utilizzate nelle finestre del programma client.

In entrambe le cartelle sopra citate (WORTH_server e WORTH_client) sono riportate per comodità anche le ***librerie jackson*** utilizzate da entrambi i programmi.

2. COMPILAZIONE ED ESECUZIONE DEI PROGRAMMI

Per far girare i programmi server e client di cui tratteremo è sufficiente seguire le semplici istruzioni seguenti:

1. decomprimere l’archivio consegnato in una cartella a piacimento;
2. **assicurarsi di mantenere sempre la struttura** descritta nella sezione precedente: all’interno della cartella WORTH_client devono essere presenti tutti i file sorgente, la cartella “icone” e le tre librerie jackson; all’interno della cartella WORTH_server devono essere presenti tutti i file sorgente e le tre librerie jackson.;
3. aprire un paio di terminali (o più se si vogliono utilizzare più istanze del programma client per far accedere al servizio più utenti contemporaneamente) e posizionarsi nelle cartelle sopra citate;
4. compilare i programmi riportando anche le librerie usate. Per comodità riporto i comandi da utilizzare per un pratico copy-paste (sempre apprezzato):
   I. *compilazione del server*: javac -cp .:jackson-core-2.9.7.jar:jackson-annotations-2.9.7.jar:jackson-databind-2.9.7.jar WORTH_Server_main.java

   II. *compilazione del client:* javac -cp .:jackson-core-2.9.7.jar:jackson-annotations-2.9.7.jar:jackson-databind-2.9.7.jar WORTH_Client_main.java
5. eseguire i programmi compilati ancora una volta includendo le librerie. Anche ora riporto i comandi da utilizzare:
   I. *esecuzione del server*: java -cp .:jackson-core-2.9.7.jar:jackson-annotations-2.9.7.jar:jackson-databind-2.9.7.jar WORTH_Server_main
   II. *esecuzione del client*: java -cp .:jackson-core-2.9.7.jar:jackson-annotations-2.9.7.jar:jackson-databind-2.9.7.jar WORTH_Client_main
6. l'esecuzione del server provocherà la creazione di file necessari per garantire la persistenza delle informazioni (maggiori dettagli nelle prossime sezioni). **Non modificare mai tali file** per garantire il corretto funzionamento del sistema.

3. INTERAZIONE CON I PROGRAMMI

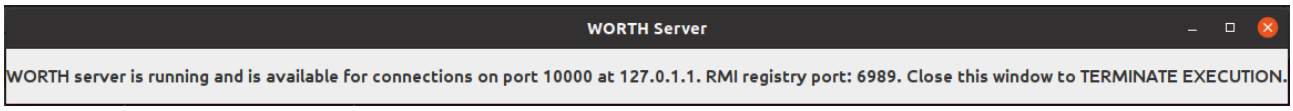
Come già intuito, entrambi i programmi fanno uso di una semplice **interfaccia grafica**; questo fa sì che l'utente possa scordarsi il terminale in background ed interagire in modo pratico con il sistema.
Al *lancio del programma server* viene presentata una finestra che consente di scegliere i parametri di connessione desiderati; in particolare si può scegliere di fare in fretta e mantenere i parametri impostati di default (mostrati nella finestra) oppure sceglierli a piacimento. Dopo aver scelto come eseguire la connessione, se non si verificano errori, viene aperta una piccola finestra in alto che mostra le impostazioni del server (da utilizzare per connettere i client se non si trovano in esecuzione nello stesso pc del server). La chiusura di questa finestra comporta la terminazione del programma server.
Al *lancio del programma client* viene (come già spiegato) aperta la finestra di connessione in cui l'utente specifica se si vuole connettere ad un server locale (in esecuzione sulla stessa macchina) oppure ad un server "remoto" (in esecuzione in un'altra macchina, connessa alla stessa rete ovviamente) inserendone i giusti riferimenti. Se la "connessione" viene stabilita con successo il programma presenta la pagina di accesso al servizio (maggiori dettagli nelle sezioni successive).
A questo punto l'utente interagisce con le varie schermate richiedendo i servizi offerti dal sistema.
** *I programmi sono progettati per gestire tutti i possibili errori/comportamenti anomali commessi dall'utente durante l'utilizzo; se dovesse tuttavia verificarsi un errore non previsto (per esempio un errore di rete), il programma (sia server che client) mostrerebbe una finestra in cui viene riassunto il tipo di errore e chiesto di riavviare*.

4. ISTRUZIONI PER L'ESECUZIONE SU MACCHINE DIVERSE

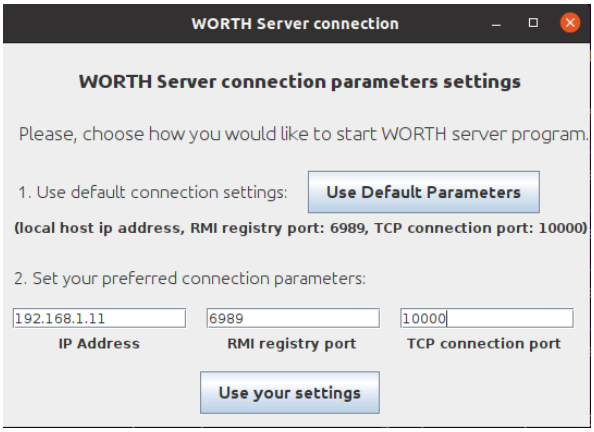
Il funzionamento dei programmi è garantito se avviene su un pc singolo. Se avviene su pc diversi (cioè ci sono dei client che sono in esecuzione su computer differenti da quello in cui è in esecuzione il server) il funzionamento è garantito soltanto se i computer sono connessi alla stessa rete e se le impostazioni di ciascuna macchina lo consentono. Nelle righe successive farò riferimento al sistema operativo Ubuntu (negli altri sistemi operativi non è garantito il funzionamento "remoto" del progetto; è comunque garantito il funzionamento normale in localhost).
Andando nello specifico, è importante controllare l'indirizzo utilizzato per la connessione del server; questo può essere diverso a seconda della macchina che si utilizza (seguono degli esempi).
Digitare il comando < ip address > in un terminale per visualizzare le impostazioni degli indirizzi di rete.

```
casa@casa-HP-Pavilion-g6-Notebook-PC:~/Scrivania/client$ ip address
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eno1: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc fq_codel state DOWN group default qlen 1000
    link/ether 2c:27:d7:e1:62:07 brd ff:ff:ff:ff:ff:ff
    altname enp5s0
3: wlp4s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen 1000
    link/ether cc:af:78:14:e4:fc brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.11/24 brd 192.168.1.255 scope global dynamic noprefixroute wlp4s0
       valid_lft 79312sec preferred_lft 79312sec
    inet6 fe80::d37:3ee2:1983:c26d/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
casa@casa-HP-Pavilion-g6-Notebook-PC:~/Scrivania/client$
```

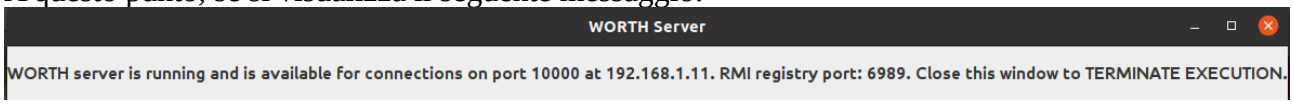
In un pc con una situazione simile a quella nell'immagine precedente (con più interfacce di rete, 3 in questo caso) la scelta dell'indirizzo ip da assegnare al programma potrebbe non essere effettuata.
Più chiaramente, lanciando il programma server mantenendo le impostazioni di default (e quindi l'indirizzo di localhost) si può ottenere questo risultato:



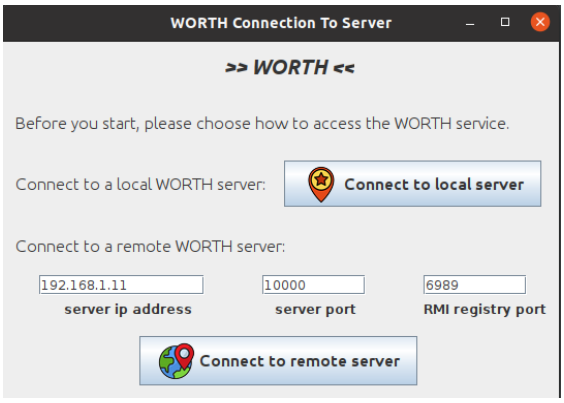
Come si nota, il server è in attesa di richieste di connessione all'indirizzo di loopback, rendendo ovviamente impossibile collegarsi da una macchina diversa.
In questo caso si può tentare di preparare il server scegliendo l'altro indirizzo ip disponibile manualmente. Digitare quindi i parametri corretti di connessione, come mostrato di seguito:



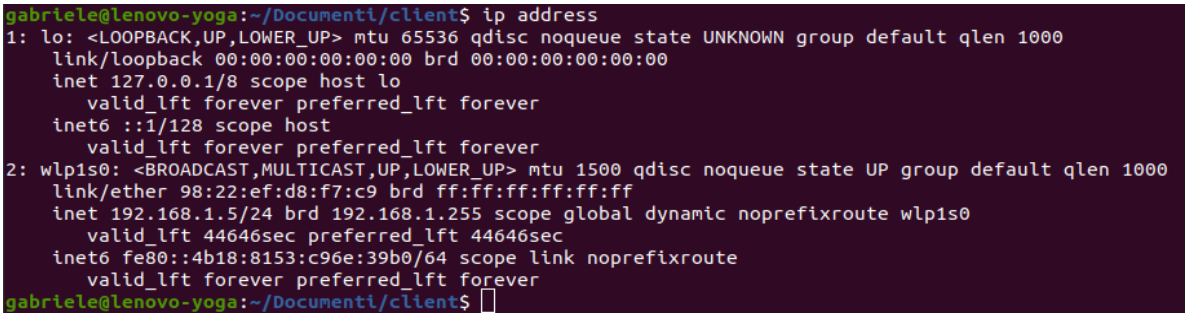
e premere il pulsante "Use your settings".
A questo punto, se si visualizza il seguente messaggio:



si può ragionevolmente ben sperare di poter utilizzare il servizio anche da remoto.
Ovviamente nei clients dovremo effettuare la connessione al server specificando i parametri corretti come mostrato subito sotto (anche in quelli in esecuzione nello stesso pc del server!!):



per poi premere il pulsante “Connect to remote server”.
Se tutto questo non dovesse funzionare, utilizzare il servizio in localhost.
In una macchina come quella nell’immagine qui sotto, ad esempio, non è stato necessario impostare manualmente l’indirizzo ip, poiché è stato automaticamente assegnato al programma l’indirizzo reale corretto.



(come si nota, qui è presente una sola interfaccia di rete diversa da quella di loopback)

*** Questo problema non si pone nel programma client, in quanto l’indirizzo ip è settato selezionando quello a cui è connessa la socket TCP con il server; impostando correttamente l’indirizzo del server, quindi, è garantito che il client utilizzi il giusto indirizzo di rete*.

Se si desidera utilizzare il servizio da “remoto” in ogni caso NON È GARANTITO il funzionamento della chat di progetto. Questo dipende dalle impostazioni del modem al quale si è collegati (IGMP, multicasting, ...).

*** In generale è da prestare estrema attenzione nell’impostare il giusto indirizzo ip per permettere il funzionamento dell’udp multicast utilizzato per implementare il servizio di chat, anche da localhost (con indirizzo impostato a loopback, come visto nella seconda delle immagini precedenti, ad esempio, i datagrammi non vengono ricevuti dal thread di ricezione dei messaggi! Procedere quindi in questo caso a settare l’indirizzo reale seguendo il procedimento illustrato sopra).*

5. ILLUSTRAZIONE DELL’ESECUZIONE DEI PROGRAMMI
In questa sezione dell’elaborato illustro brevemente il funzionamento dei programmi client e server di WORTH e come essi interagiscono fra di loro.
Tenterò quanto più possibile di seguire il flusso normale di esecuzione per rendere il tutto comprensibile. Ad ogni modo, consiglio di leggere quanto segue tenendo a portata di mano il codice consegnato e prendendo in considerazione i numerosi commenti a fianco ad esso.

5.1 WORTH SERVER
Il programma server di WORTH (codice contenuto nel file “WORTH_Server_main.java”), come già detto precedentemente, all’inizio della sua esecuzione mostra la finestra di **impostazione dei parametri di connessione**, in cui l’utente può scegliere se mantenere i parametri impostati di default o impostarne di nuovi a suo piacimento (*tenere in considerazione gli accorgimenti illustrati nella sezione 4*).
Una volta che l’utente ha effettuato la scelta, se non si riscontrano errori, il server comincia il suo **procedimento di inizializzazione del servizio**. In particolare il programma:

- ✔ crea un nuovo oggetto di tipo “Registration_Service” che costituisce l’oggetto remoto che verrà esportato per consentire agli utenti di iscriversi al servizio con le proprie credenziali utilizzando RMI del metodo “register”. Il server utilizza questo oggetto come data base degli utenti iscritti. Infatti, oltre al metodo remoto che consente di effettuare la registrazione, troviamo i metodi utili per eseguire login, logout, modifica della password di accesso e eliminazione di un account. La lista degli utenti registrati al servizio è resa persistente su file system nel file “*registeredUsers.json*”, che viene creato nella cartella di lavoro del server, e contiene una serializzazione, realizzata dall’object mapper, dell’hash table “*usersDB*”.

Inoltre l'oggetto è responsabile dell'effettuazione delle notifiche callback ai client iscritti al servizio (come vedremo meglio più avanti).

- ✔ esporta l'oggetto remoto di cui sopra in modo da renderlo disponibile per le chiamate esterne da parte dei clients; l'esportazione avviene utilizzando l'indirizzo ip impostato durante la fase di avvio. Se l'utente non effettua i controlli di cui abbiamo trattato nella sezione 4, l'oggetto potrebbe essere esportato utilizzando l'indirizzo di loopback, rendendo il servizio utilizzabile soltanto da localhost. Inserendo un indirizzo ip diverso (associato dal modem alla macchina su cui si sta lavorando ovviamente) invece, l'istruzione "System.setProperty("java.rmi.server.hostname", myIp);" provvede ad impostare l'indirizzo di rete corretto.
- ✔ crea un registro sulla porta selezionata nella fase di avvio e vi registra lo stub ottenuto dall'esportazione precedente. Successivamente apre un nuovo canale con cui comunicherà con i clients, connettendo il relativo socket nel rispetto delle impostazioni fornite in fase di avvio, apre un nuovo selettore e vi registra il canale appena aperto.
- ✔ dopo una serie di dichiarazioni di variabili ausiliarie utilizzate nel corso dell'esecuzione, il programma comincia la fase di **recupero delle informazioni persistenti**. In particolare il server accede alla cartella "*WORTH_projects_serverDirectory*", creata nella cartella di lavoro del server (dove sono situati i file con il codice), in cui si trovano tutte le cartelle dei vari progetti creati in precedenti esecuzioni dei programmi. All'interno di ciascuna cartella di progetto viene creato un file dal nome "*members.json*" in cui è contenuta una serializzazione della lista di membri di quel progetto, e una cartella "*cards*" all'interno della quale sono creati i file relativi alle cards di progetto create dagli utenti. Nei file delle cards sono salvate le serializzazioni delle storie delle cards (successione di liste visitate). Altri due file sono utilizzati per persistere informazioni: il file "*deleters.json*", uno per ogni progetto, che mantiene la lista di utenti che hanno inviato delle richieste di cancellazione per quel progetto, e il file unico "*noviceMembers.json*", che contiene la lista di utenti che sono stati aggiunti come nuovi membri di progetti e che devono ancora sincronizzare i loro accounts (per maggiori dettagli vedere punti successivi).
- ✔ per ogni progetto da persistere, il programma genera un indirizzo di multicast da utilizzare per il servizio di chat. La generazione dell'indirizzo è effettuata in modo puramente casuale, con lo scopo di utilizzare tutto il range di indirizzi disponibili (con primo byte da 225 a 239). ** *Essendo gli indirizzi generabili in numero maggiore di 250 milioni e tenuto conto dell'ambito ristretto di utilizzo del progetto, così per come esso è stato definito, non ho intenzionalmente tenuto conto del problema relativo al loro esaurimento (che quindi comporterebbe il malfunzionamento del server). Gli indirizzi sono riutilizzabili una volta che il relativo progetto è stato definitivamente eliminato dal sistema.*
- ✔ a questo punto il server è pronto a ricevere richieste di connessione dai clients. Tutte le richieste che provengono dai clients sono gestite utilizzando la tecnica di **multiplexing dei canali mediante il selettore di NIO** aperto in fase di inizializzazione, con lo scopo di favorire un più efficiente utilizzo delle risorse, rendere il programma più performante e facilitare la gestione dei task richiesti dagli utenti, rispetto ad un semplice approccio multithreaded.

Dopo aver stabilito la connessione TCP con ciascun client, questa connessione viene utilizzata per lo scambio delle informazioni tra i due programmi. Nello specifico, il client (il cui canale viene subito impostato dal server con operazione di interesse "read") invia dei codici di operazione che vengono riconosciuti ed elaborati dal programma server. Dopo una prima elaborazione del server da parte del "ramo readable" ***l'attachment*** della chiave associata al client viene utilizzato per memorizzare le informazioni necessarie al "ramo writable" per completare l'operazione richiesta e restituire il risultato al client (l'elenco delle operazione di write (writeOp) sarà illustrato in seguito).
Le **operazioni** che i clients possono richiedere al server sono riassunte nell'elenco seguente:

(operazione: 0) Richiesta l'operazione di ***login*** dall'utente. Il server recupera le credenziali dell'utente dal buffer di ricezione e invoca il metodo "*checkUserLogin*" dell'oggetto di tipo

"*Registration_Service*", il quale controlla che l'utente esista e che la password coincida. L'esecuzione passerà poi al "ramo writable" (codice writeOp 0) che restituirà il risultato al client.

(operazione: 1) Operazione utilizzata dal client, dopo un login effettuato con successo, per richiedere al server la lista di ***riferimenti alle chat*** dei progetti di cui l'utente (che ha effettuato il login) è membro. Dettagli forniti nell'elenco successivo al writeOp -8.

(operazione: 2) Richiesto il ***logout*** dal servizio. Il server imposta ad offline lo stato dell'utente e chiude la connessione con il client (cancellando anche la relativa chiave).

(operazione: 3) Operazione richiesta dal client dopo aver effettuato una registrazione con successo. Il server semplicemente imposta ad online lo stato dell'utente nel sistema e memorizza nell'attachment il nome dell'utente (il nome dell'utente che utilizza il client a cui è associata la chiave deve essere sempre contenuto nell'attachment come spiegato alla fine di questo elenco).

(operazione: 4) Questo codice è inviato dal client quando l'utente desidera ***creare un nuovo progetto***. Se il progetto può essere creato (non esiste già un progetto con lo stesso nome) il server crea una nuova cartella di progetto e vi inserisce un file di membri "members.json" che conterrà inizialmente soltanto il nome dell'utente creatore. Genera inoltre un indirizzo per il servizio di chat e lo restituisce al client tramite operazione writeOp -4 (dettagli in seguito).

(operazione: 5) L'utente richiede di ***visualizzare il menù di un progetto***. Il server si occupa di controllare che il progetto esista, che l'utente ne sia membro e che non abbia inviato una richiesta di eliminazione per quel progetto (che comporta non poterne visualizzare più il menù, come descritto più tardi qui sotto).

(operazione: 6) L'utente ha richiesto la ***lista di progetti*** di cui è membro (maggiori dettagli al writeOp -2).

(operazione: 7) Codice utilizzato per identificare l'operazione di ***cancellazione dell'account*** WORTH. Se la password spedita dal client corrisponde con quella dell'utente che ha richiesto l'operazione, il server procede con la rimozione di quest'ultimo dal sistema. In particolare, l'utente viene rimosso da tutte le liste di membri dei progetti e dai data base del server. Se l'utente da rimuovere è l'unico membro di alcuni progetti o l'unico membro a non aver ancora inviato una richiesta di cancellazione per alcuni progetti, anche questi vengono definitivamente eliminati.

(operazione: 8) Operazione di ***cambio della password***. Viene effettuato soltanto se la vecchia password specificata dall'utente coincide.

(operazione: 9) Se il progetto non è in stato di eliminazione e non esiste una card con il nome specificato, ***crea una nuova card*** di progetto. La card viene inserita nella lista todo con nome creatore quello dell'utente che ha richiesto l'operazione.

(operazione: 10) Operazione richiesta se l'utente vuole visualizzare le ***informazioni relative ad una card*** di progetto (nome, progetto di appartenenza, autore e lista corrente).

(operazione: 11) Richiesta la ***lista delle cards*** di un progetto (dettagli al writeOp -6).

(operazione: 12) L'utente ha richiesto lo ***spostamento di una card*** di progetto da una lista ad un'altra. Il server controlla che lo spostamento possa essere eseguito (nel rispetto dei vincoli dettati dalla specifica) invocando il metodo "*moveCard*" della classe Project.

(operazione: 13) Richiesta la visualizzazione della ***storia*** di una card.

(operazione: 14) L'utente vuole inserire una ***richiesta di cancellazione del progetto***. L'operazione può essere effettuata soltanto se tutte le card di progetto sono nella lista done (come da specifica). Il server mette il progetto in stato di eliminazione ad aggiunge il membro tra i "deleters" (aggiornando il file "*deleters.json*"). Se tutti i membri hanno richiesto l'eliminazione del progetto (l'utente in questione è l'ultimo ad inserire la propria richiesta) allora questo viene definitivamente eliminato. Un progetto in stato di eliminazione rimane tale fin quando o viene eliminato o tutti i membri annullano l'invio delle loro richieste di eliminazione. Un progetto in stato di eliminazione non è più modificabile, ossia non è possibile ne creare o eliminare delle card ne aggiungere nuovi membri. In seguito

all'invio della richiesta di eliminazione l'utente non può più visualizzare il menù del progetto, a meno che non decida di annullare l'invio. Affinché un progetto torni ad essere modificabile, tutti i membri devono annullare le proprie richieste di eliminazione.

(operazione: 15) Richiesta l'***eliminazione di una card*** di progetto non più utile. Se il progetto non è in stato di eliminazione e la card esiste, allora chiunque può richiederne l'eliminazione (a patto che sia membro del progetto ovviamente).

(operazione: 16) Codice che viene inviato dal client quando è pronto a ricevere una lista serializzata tramite la rete, dopo aver allocato un buffer di ricezione delle giuste dimensioni (maggiori dettagli alla fine di questo elenco).

(operazione: 17) L'operazione che ***aggiunge un membro ad un progetto***. Se il progetto non è in stato di eliminazione, l'utente da aggiungere è registrato nel sistema e non è già membro del progetto, il nuovo membro viene inserito nella lista dei "*noviceMembers*" (aggiornato il file "*noviceMembers.json*"). I "*novice members*" sono i membri novizi, cioè quei membri che sono stati appena aggiunti a dei progetti e non hanno ancora sincronizzato la loro lista dei progetti (richiesto l'operazione successiva).

(operazione: 18) L'utente vuole ***sincronizzare la sua lista di progetti***, cioè vuole controllare se qualcuno lo ha aggiunto come nuovo membro di qualche nuovo progetto. Il server invia al client la lista dei riferimenti alle chat degli eventuali nuovi progetti, così che l'utente possa cominciare a ricevere i messaggi per quei progetti (maggiori dettagli nel writeOp -10).

(operazione: 19) Il programma client richiede la ***lista di membri di un progetto*** (writeOp -11).

(operazione: 20) L'utente desidera ***abbandonare un progetto***. L'operazione può essere effettuata con successo soltanto se l'utente non è l'unico membro del progetto. Se l'utente che vuole abbandonare è l'unico a non aver ancora inviato una richiesta di cancellazione per il progetto, quest'ultimo viene definitivamente eliminato dal sistema.

(operazione: 21) L'utente ha richiesto di ***annullare l'invio della sua richiesta di eliminazione*** per un progetto. Se per quando conferma il progetto non è ancora stato eliminato (non tutti i membri hanno ancora inviato la loro richiesta di eliminazione), il server spedisce al client l'indirizzo della chat del progetto, in modo tale che l'utente possa ricominciare a ricevere i messaggi del progetto non ancora eliminato.

*** Per tutte le operazioni sopra in cui è necessario che il server invii al client una lista di oggetti di dimensione variabile, avvengono due comunicazioni consecutive tra i due programmi. Nella prima comunicazione il server invia al client la lunghezza della lista richiesta, permettendo al client di allocare un buffer di ricezione delle giuste dimensioni. Nella seconda comunicazione (che avviene quando il client richiede al server l'operazione con codice 16, come visto nell'elenco sovrastante) il server invia finalmente la lista. In tutte le altre richieste il problema non si pone poiché i vincoli imposti dal programma client garantiscono che le informazioni possano essere contenute nei normali buffer utilizzati per le operazioni di rete*.

Se al momento della lettura il server non riceve alcun dato dal client, procede con la chiusura della connessione con esso. In particolare lo deregistra dal servizio di notifica callback, **chiude la connessione** TCP, cancella la chiave relativa al suo canale NIO e imposta lo stato del suo utente ad offline (è anche a questo che serve il nome nell'attachment delle chiavi).
Nel processare le varie richieste, il server effettua dei **controlli di sicurezza**. Nelle operazioni in cui c'è da modificare il file system (e quindi modificare un progetto) il programma si accerta che il client da cui riceve quelle richieste sia effettivamente un client "legale" del servizio. Nello specifico, quando l'utente richiede ad esempio di creare una nuova card, non è possibile che questo non sia un membro del progetto in cui la vuole creare, perché il controllo è già stato effettuato al momento della visualizzazione del menù. Per questo motivo il server effettua questo tipo di controllo:

```
if(projects.get(prjName) == null || !projects.get(prjName).checkMember(name)){      //controlli di sicurezza
    regService.unRegisterForNotification(name);    //client rimosso dalla lista di avvisi callback
    regService.logOut(name);           //imposta ad offline lo stato dell'utente
    client.close();            //il client fraudolento viene buttato fuori dal servizio
    chiave.cancel();
    continue;
}
```

(se il client è quello normale, il codice dell'immagine non viene MAI eseguito)

per evitare che un malintenzionato possa stabilire una connessione TCP col server e modificare i progetti di cui non è membro.

Quando il client è pronto per **operazioni di scrittura**, il server agisce diversamente in base al writeOp memorizzato nei primi 4 bytes dell'attachment della relativa chiave, impostato dal "ramo readable". Vediamo molto brevemente quali sono le possibili operazioni che esegue:

(writeOp 0) spedisce l'output del controllo di login al client. Se il login è avvenuto con successo, memorizza il nome utente nell'attachment altrimenti chiude la connessione col client;

(writeOp -1) operazione effettuata molto di frequente quando il server deve spedire tramite la rete al client un risultato di operazione che sia un intero (0 per il successo, !=0 per insuccesso …);

(writeOp -2) il server prepara la lista di progetti di cui l'utente è membro e ne invia la lunghezza al client (come detto prima). La lista viene memorizzata nell'attachment e inviata al client nella successiva operazione di scrittura (dopo che il client ha richiesto l'operazione 16) nel writeOp -9;

(writeOp -3) il server restituisce al client il risultato della cancellazione dell'account WORTH. Se la cancellazione è avvenuta, chiude la connessione con il client;

(writeOp -4) operazione con cui il server invia al client l'indirizzo di multicast di una chat di progetto. Utilizzata al momento della creazione di un nuovo progetto o all'annullamento di una richiesta di cancellazione per un progetto (di cui il client deve ricominciare a ricevere i messaggi);

(writeOp -5) il server invia al client le informazioni relative ad una card di progetto. L'invio avviene immediatamente senza passare per il writeOp -9 poiché i limiti imposti dal programma garantiscono che i dati possano essere contenuti nei buffer normali con cui vengono gestite le operazioni di rete.;

(writeOp -6) il server prepara la lista delle cards di un progetto e ne invia la lunghezza al client (la lista, memorizzata nell'attachment, verrà inviata successivamente nel writeOp -9);

(writeOp -7) il server invia al client la lunghezza della storia di una card (la storia, memorizzata nell'attachment, verrà inviata successivamente nel writeOp -9);

(writeOp -8) operazione richiesta dal client dopo il login. Il server prepara la lista di riferimenti alle chat di progetto di cui l'utente è membro e ne invia la lunghezza (la lista, memorizzata nell'attachment, verrà inviata successivamente nel writeOp -9);

(writeOp -9) codice di operazione in cui il server invia al client la lista serializzata nell'attachment (lista di indirizzi delle chat, lista di progetti, lista di cards, storia di una card, ...). L'operazione viene effettuata dopo che il client ha allocato un buffer di ricezione delle giuste dimensioni e ha richiesto l'operazione del server con codice 16 (come visto nell'elenco precedente);

(writeOp -10) il server sincronizza la lista di progetti di cui un utente è membro. In particolare controlla se l'utente è un "*novice member*" di qualche progetto, e se lo è, lo rende un membro effettivo conferendogli tutti i diritti. Il server prepara la lista dei riferimenti alle chat dei nuovi progetti dell'utente e ne invia al client la lunghezza (la lista, memorizzata nell'attachment, verrà inviata successivamente nel writeOp -9);

(writeOp -11) il server prepara la lista di membri di un progetto e ne invia la lunghezza al client (la lista, memorizzata nell'attachment, verrà inviata successivamente nel writeOp -9);

(writeOp -12) il server invia al client il numero delle richieste rimanenti per la cancellazione del progetto che l'utente vorrebbe visualizzare (quando l'utente tenta di aprire il menù di un progetto per cui ha inviato una richiesta di cancellazione).

Per tutte le operazioni di modifica di un progetto (aggiunta, spostamento, rimozione di una card, aggiunta o rimozione di un membro o invio, annullamento di una richiesta di cancellazione) il server recupera l'indirizzo di multicast della chat di quel progetto e vi **invia la relativa notifica** ai membri.
Per tutte le operazioni in cui viene modificato lo stato di un utente nel sistema (login, logout, cancellazione dell'account, aggiunto ad un nuovo progetto, abbandono di un progetto, …) il server effettua delle **notifiche callback** ai client attivi al momento (come vedremo in dettaglio qui sotto).

5.2 WORTH CLIENT
Arrivati a questo punto della relazione, gran parte delle funzionalità del client sono già state descritte. Essendo il codice organizzato come una serie di cicli while per gestire gli spostamenti tra le finestre della GUI, direi di procedere analizzando le operazioni più importanti svolte dal programma finestra per finestra.
5.2.1 Inizio dell'esecuzione e finestra di connessione al server
Come già detto, all'inizio dell'esecuzione il programma presenta all'utente una finestra di connessione al server grazie alla quale si può scegliere se connettersi ad un server in esecuzione sulla stessa macchina del client oppure ad un server "remoto" (in esecuzione su un computer diverso ma connesso alla stessa rete). *Attenzione*: se si modificano i parametri di connessione del server (indirizzo ip soprattutto) è sempre consigliato riportare tali parametri in questa finestra e premere "Connect to remote server" invece di cliccare direttamente sul pulsante "Connect lo local server", anche se il server è in esecuzione sullo stesso computer.
Effettuata la scelta in questa finestra, il programma tenta di acquisire il registro del server e di ottenere lo stub remoto di registrazione al servizio. Se non si riscontrano problemi di connessione viene visualizzata la finestra successiva.
5.2.2 Finestra di accesso al servizio WORTH
Dopo aver creato l'oggetto remoto (*callBackReceiver*) da esportare per ricevere gli aggiornamenti callback sullo stato degli utenti nel sistema da parte del server e un oggetto di tipo "*Chats*" per gestire le chat di progetto di cui l'utente è membro, viene visualizzata questa finestra, in cui quest'ultimo può scegliere se effettuare una registrazione con le proprie credenziali nel sistema oppure accedere al proprio account esistente.
Dopo aver effettuato tutti i controlli necessari (che invito a vedere direttamente nel codice per non dilungarmi più di quanto già ho fatto) il client richiede al server l'operazione di registrazione (RMI) o di login (tramite la connessione TCP stabilita).
Se l'operazione richiesta dell'utente può essere completata con successo (il server non comunica nessun errore) in entrambi i casi (sia dopo una registrazione che dopo un login) il programma imposta la proprietà di sistema ***System.setProperty("java.rmi.server.hostname", myIP)***, in cui myIP è l'indirizzo locale a cui è connessa la socket TCP col server. Questo evita che alcuni dispositivi possano esportare il proprio *callBackReceiver* utilizzando l'indirizzo di loopback (come visto nelle sezioni precedenti) rendendo impossibili le callback del server. Dopodiché il programma effettua la registrazione al servizio di notifica callback, appunto, e riceve in risposta dal server, come da specifica, la lista di tutti gli utenti registrati al servizio (ammissibile considerando l'ambito ristretto di utilizzo del progetto, così per come esso è stato definito).
Se l'operazione che si è effettuata era di login, allora il client richiede al server la lista dei riferimenti di tutte le chat di progetto di cui l'utente è membro (*l'utente online riceve sempre tutti i messaggi, da tutte le chat a cui appartiene*).
Dopo aver attivato il *thread di ricezione dei messaggi*, di cui parleremo nei dettagli tra poco, visualizza la pagina successiva.
5.2.3 Home Page del sistema WORTH
È la finestra principale di WORTH dalla quale si accede a tutti i servizi messi a disposizione. In particolare si possono creare nuovi progetti, visualizzare il menù di progetti esistenti, richiedere la lista dei progetti di cui si è membri, richiedere la lista di tutti gli utenti del sistema e la lista dei soli utenti online al momento. Queste ultime due operazioni non richiedono una comunicazione con il

server ma sfruttano la struttura dati locale del programma client (*callBackReceiver*) che viene aggiornata tramite le operazioni di callback.
Le altre operazioni che si possono effettuare sono: richiedere la sincronizzazione della propria lista di progetti (per controllare se qualcuno ci ha aggiunti come nuovi membri, e se si, diventarlo in modo effettivo), visualizzare la finestra di impostazioni del profilo (di cui parleremo alla fine di questa sezione) ed eseguire il logout dal servizio.

5.2.4 Menù di un progetto WORTH

Pagina in cui si gestiscono le operazioni relative al lavoro su di un progetto. Sono permesse tutte le attività che ci si aspetta di poter eseguire su un progetto: creazione di una nuova card, spostamento di una card, visualizzazione della finestra di informazioni di una card, ottenere la lista delle card di progetto, ottenere la storia di una card, eliminare una card di progetto, eliminare tutto il progetto; di cui abbiamo già parlato.
Da questa finestra si possono aprire altre due finestre. L'utente può richiedere di visualizzare la *lavagna kanban del progetto*, con la quale il sistema fornisce un'immagine della situazione attuale del progetto, mostrando la composizione delle quattro liste di cards. Inoltre si può aprire la finestra della chat e dei membri di cui parliamo adesso.

5.2.5 Finestra della chat e dei membri di un progetto

Quando l'utente richiede l'apertura della finestra della chat e dei membri il programma richiede al server la lista di tutti i membri del progetto in questione e la utilizza per impostare la struttura dati "*projectMembers*" dell'oggetto "*callBackReceiver*" che verrà aggiornata con notifiche callback da parte del server (in caso di aggiunta di un membro, abbandono da parte di un membro, login/logout di un membro, …). In seguito il client crea una finestra "*ChatFrame*", setta un riferimento a quest'ultima sia nell'oggetto "*callBackReceiver*" sia nel thread di ricezione dei messaggi, e poi invoca il metodo *callBackReceiver.setMemberStatusAtStart*() che visualizza la lista dei membri del progetto nella finestra appena aperta con tanto di stato nel sistema (online/offline); la ricezione di callbacks da parte del server provoca un aggiornamento in tempo reale di questa lista visualizzata. Il riferimento alla finestra nel thread di ricezione dei messaggi serve per poter visualizzare i messaggi nella chat nel momento in cui vengono ricevuti.
All'uscita dalla finestra della chat i riferimenti ad essa nell'oggetto "*callBackReceiver*" e nel thread di ricezione messaggi sono resettati dal programma.
Da questa finestra l'utente può aggiungere un altro utente registrato al servizio come membro del progetto o abbandonare il progetto a cui non desidera più partecipare.
Infine ogni membro può spedire messaggi in chat. L'invio dei messaggi da parte dell'utente è implementato nel codice della finestra (file "*ChatFrame.java*") con il metodo "*sendButtonActionPerformed*", invocato quando il pulsante "*Send message*" viene premuto. Il programma non fa altro che aprire una DatagramSocket e inviare il pacchetto UDP all'indirizzo di multicast della chat.
Nella chat sono visualizzati anche gli avvisi di sistema (del programma che l'utente sta utilizzando) e del server.
L'inserimento dei messaggi nel documento che rappresenta la chat è effettuato da tre thread differenti (il main, il thread di ricezione dei messaggi e il sotto-thread che gestisce la finestra); il corretto accodamento dei messaggi è garantito dall'acquisizione della lock creata appositamente.

THREAD DI RICEZIONE DEI MESSAGGI

Come già spiegato, il programma client fa partire l'esecuzione di un thread che ha il solo compito di ricevere i messaggi provenienti da tutte le chat di progetto di cui un utente fa parte. Questo thread utilizza la struttura "*chats*" condivisa con il thread main, in cui inserisce il testo dei messaggi ricevuti. È realizzato impiegando il multiplexing dei canali con un selettore di NIO; nel selettore è infatti registrata una serie di DatagramChannel, ciascuno dei quali è impostato per ricevere i messaggi UDP multicast provenienti da una chat. La motivazione di questa scelta di implementazione è sicuramente da far risalire al miglior utilizzo delle risorse e alla maggior efficienza e tempestività di gestione della ricezione e visualizzazione dei messaggi in tutte le chat,

rispetto ad un approccio multithreaded, per esempio con un thread attivo per ogni chat di progetto. Se vogliamo, un'ulteriore motivazione è senz'altro che quest'ultimo approccio, seppur accettabile tenendo conto dell'ambito di utilizzo del sistema, non sarebbe stato adatto ad una possibile estensione futura del servizio WORTH (un conto è, ad esempio, modificare l'algoritmo di generazione degli indirizzi di multicast, un conto è modificare tutto il processo di ricezione dei messaggi).
Quando il programma client deve aggiornare la lista dei progetti da cui ricevere messaggi (l'utente è stato aggiunto a nuovi progetti o ne ha abbandonati/eliminati altri) utilizza la struttura dati condivisa per sincronizzarsi col thread di ricezione. Nello specifico, utilizza le due liste "chatAdded" e "chatRemoved" per comunicare al thread quali sono gli indirizzi di multicast da cui vuole cominciare a ricevere messaggi o per i quali interromperne la ricezione. I canali da chiudere sono individuati utilizzando il nome del progetto associato alla chat (che rimane sempre memorizzato nell'attachment delle chiavi registrate nel selettore).
Quando il thread riceve un messaggio da una chat di progetto lo mostra all'utente; se l'utente ha la finestra di quella chat aperta, il thread aggiunge semplicemente il messaggio in chat, se invece l'utente non sta visualizzando la chat in quel momento, il thread genera una notifica di avviso (a meno che l'utente non le abbia disattivate nella finestra di impostazioni del profilo).

5.2.6 Finestra di impostazioni del profilo WORTH

Concludo questo elaborato trattando della finestra di impostazioni del profilo, aggiuntiva rispetto a quanto richiesto dalle specifiche di progetto, in cui un utente ha la possibilità di: disattivare le notifiche di ricezione dei messaggi (il thread di ricezione smette di avvertire l'utente quando egli riceve messaggi da chat che non sta visualizzando in quel momento; i messaggi vengono comunque mostrati all'utente nel momento in cui accede alle rispettive chat, prima di terminare il programma), modificare la propria password ed eliminare definitivamente il proprio account. Per effettuare queste ultime due operazioni l'utente deve indicare la propria password corrente per ovvie ragioni di sicurezza.