Laboratorio di Programmazione di Reti
Progetto di fine corso (A.A. 20-21)

Docente: Laura Ricci
Autore del progetto: Alessio La Greca (Corso A), matricola 581578

Indice

Introduzione e considerazioni sul progetto

Questa relazione è da considerarsi come un completo manuale di istruzioni per l'utilizzo e la comprensione di Worth (Versione Alessio La Greca©), suddiviso in due grandi parti: la prima dedicata agli utenti finali (Capitolo 1), che spiega in breve i servizi offerti da Worth e illustra l'interfaccia grafica. La seconda (tutti gli altri capitoli) dedicata agli sviluppatori, così da permettere loro di comprendere le scelte progettuali e implementative, e farsi un'idea precisa di come funzioni Worth "dietro le quinte", salvandogli quindi la fatica di dover interpretare il codice sorgente (che risulta comunque opportunamente commentato e accessibile agli sviluppatori).

Parlando del progetto in quanto tale invece: è stata sinceramente un'esperienza formativa, con momenti di sfida controbilanciati da altri di grande soddisfazione, che hanno reso nel complesso lo sviluppo qualcosa di spesso divertente e stimolante. Ciò di cui vado più orgoglioso sono sicuramente lo studio sulle lock a grana fine trattate nei paragrafi 4.3 e 4.4 e l'uso e il riciclo degli indirizzi di multicast, trattati nel paragrafo 2.3. Inoltre, grazie ai continui esercizi settimanali condotti durante il corso, paradossalmente la parte più difficile è stata l'interfaccia grafica, che soprattutto nella parte di feedback in tempo reale mi ha dato non pochi grattacapi, ma che alla fine sono riuscito a gestire come volevo. Spero davvero che questa relazione, combinata al codice commentato, possa far trasparire il modello concettuale del mio sviluppo di Worth.

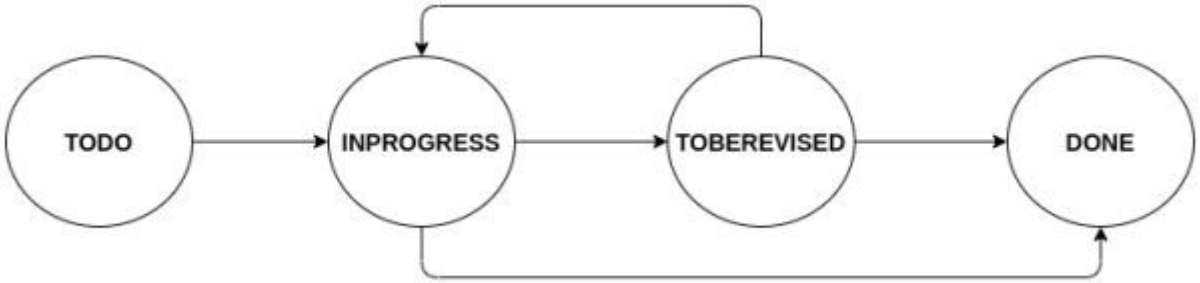
Il progetto è stato sviluppato usando Eclipse su Windows 10, versione Java 13. L'interfaccia grafica si basa su Swing e per la serializzazione ho deciso di usare Gson (versione 2.8.6). Il file pom è comunque compreso nella consegna.

Capitolo 1: Panoramica per gli utenti

1.1 – Una breve panoramica di Worth

Grazie per aver scelto Worth, generico utente! Con Worth sarai in grado di organizzare e condividere progetti seguendo il metodo Kanban, per coordinarti con i tuoi colleghi di lavoro o

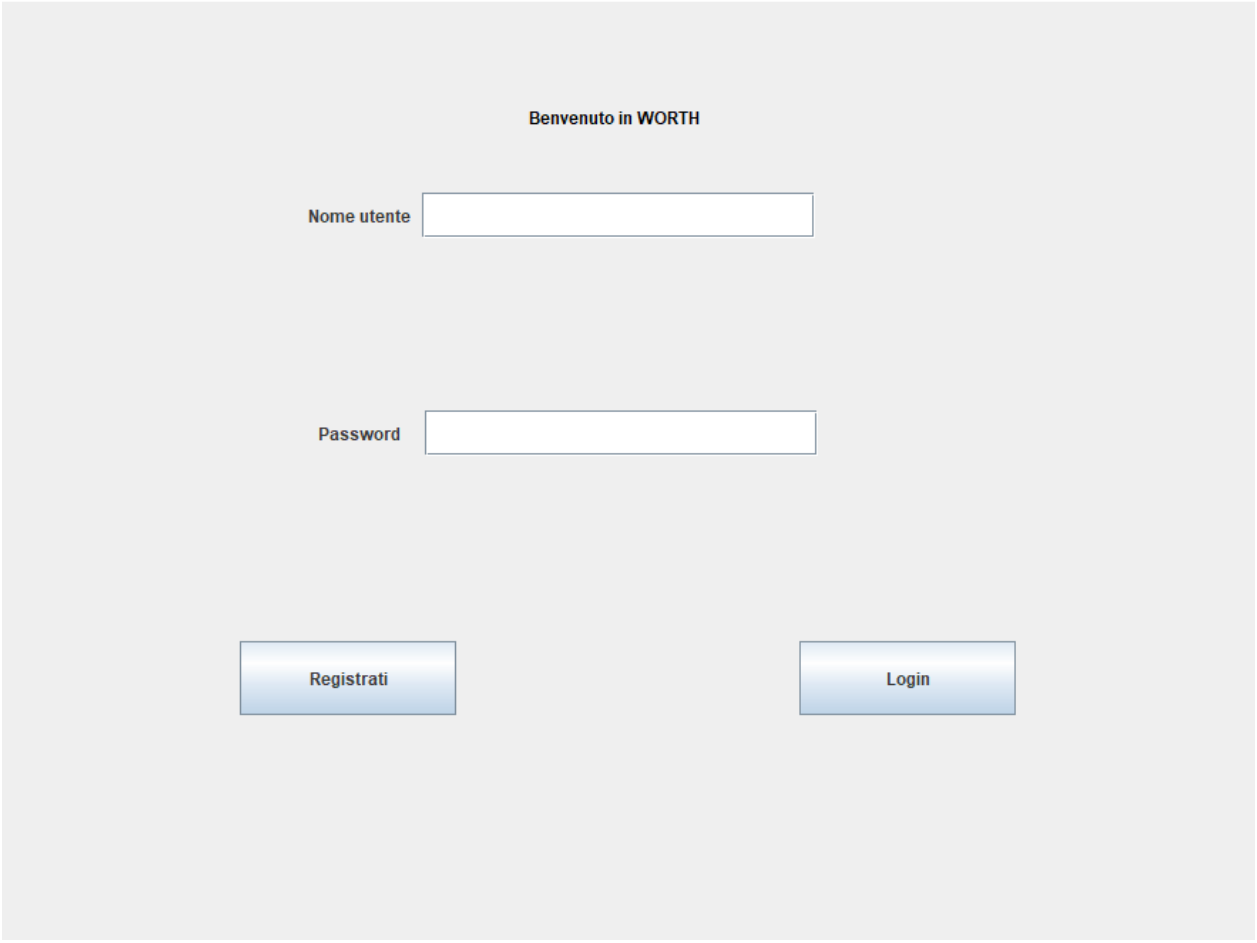
amici nella gestione di progetti arbitrariamente complessi. A causa del Covid infatti le riunioni in ufficio non sono più possibili (e diciamocelo, non erano nemmeno così tanto stimolanti), e bisogna ripiegare sui mezzi tecnologici per sopperire a questa mancanza di comunicazione diretta.
Quando accederai a Worth per la prima volta dovrai innanzitutto registrarti scegliendo un nome utente della lunghezza massima di 20 caratteri e una password, che verrà opportunamente cifrata dal nostro sistema per garantire la sicurezza delle tue informazioni personali. In Worth un nome utente è univoco, per cui, se il nome che inserisci esiste già, ti verrà chiesto di modificarlo o sceglierne un altro. Una volta registrato ti sarà possibile effettuare il login, che ti porterà alla pagina iniziale di Worth. Da qui potrai visualizzare tutti gli utenti registrati a Worth e controllarne il loro stato, visualizzare solo gli utenti attualmente online, creare un nuovo progetto o visualizzare i progetti di cui fai parte. Attento: il nome di un progetto è univoco in tutto Worth, pertanto se provi a creare un progetto che ha lo stesso nome di uno già esistente, ti verrà chiesto di cambiarlo.
Una volta entrato nel menu a scorrimento di visualizzazione dei progetti, fai doppio click sul nome di un progetto per arrivare alla sua pagina. Qui potrai visualizzare i membri del progetto, aggiungerne di nuovi, creare, modificare e visualizzare le card, accedere alla chat e cancellare il progetto. Considera una card come un post-it che racchiude al suo interno un compito da svolgere: ogni card ha un nome (che ha lunghezza massima 50 caratteri, è univoco all'interno di un progetto e non può essere uguale al nome del progetto), una descrizione (che non non può essere modificata una volta che la card è stata creata) e uno stato. Ci sono quattro possibili stati in cui può trovarsi una card: To do, In progress, To be revised e Done. Le card possono essere spostate da uno stato all'altro rispettando il seguente diagramma:



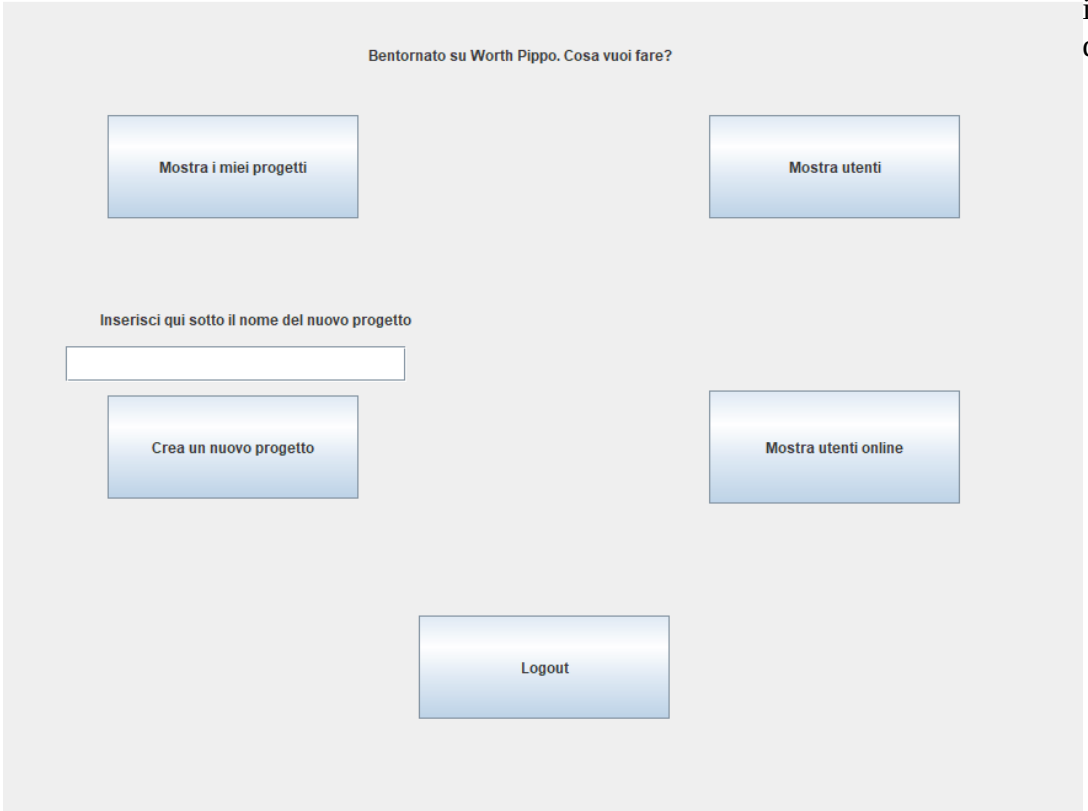
È inoltre possibile reperire la history di una card, ovvero i vari spostamenti di stato che sono stati effettuati su di essa. Per visualizzare le informazioni di una card (nome, descrizione e stato), clicca sul pulsante "mostra le card di questo progetto", e dalla lista che compare, fai doppio click sulla card che ti interessa. Se e solamente se tutte le card del progetto si trovano nello stato "Done", il progetto può essere cancellato da uno qualsiasi dei suoi membri. Inoltre, con il pulsante in basso a sinistra "vai alla chat" ti sarà possibile accedere alla chat del progetto, dove potrai visualizzare in tempo reale i messaggi inviati e inviarne a tua volta. Il massimo numero di caratteri è 490, e il counter x/490 situato a destra nell'interfaccia ti aiuterà a tenere traccia del numero di caratteri inseriti. Nota bene: nel momento in cui effettui il login comincerai a ricevere i messaggi inviati su tutte le chat di cui fai parte, ma nel momento in cui effettui il logout, al tuo prossimo login non rivedrai quei messaggi. Questo è in analogia a ciò che accade nella realtà: se sei ad una riunione con i tuoi colleghi e siete tutti nella stessa stanza, puoi sentire cosa hanno da dire gli altri. Ma se fai una capatina al bagno, ti perderai la discussione che i tuoi colleghi hanno portato avanti nel frattempo.
Qualora inoltre dovessi richiedere un'operazione impossibile da soddisfare, Worth ti notificherà l'esito negativo con un popup. Se inoltre provi ad effettuare un'operazione su un progetto che non esiste più (magari sei entrato nella pagina di un progetto, sei andato al bagno, nel frattempo un altro membro lo cancella, e quando torni vuoi visualizzare le card del progetto), Worth ti riporterà alla pagina iniziale.
Detto questo, divertiti con Worth!

1.2 – L'interfaccia grafica

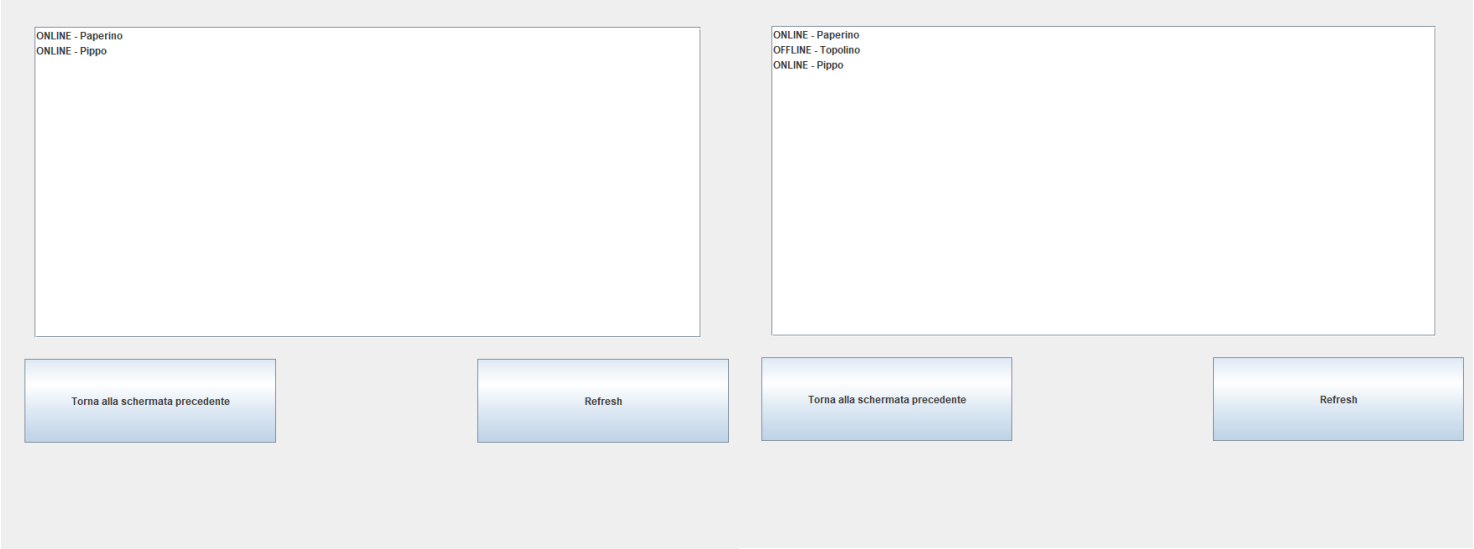
Per offire un’esperienza utente senza precedenti, i nostri *esperti* grafici di Worth hanno progettato una nuova tipologia di interfaccia grafica Facile, Altamente Intuitiva, Lussuosa, Silenziosa, Unica e Ordinata: l’interfaccia FAILSUO!
Con FAILSUO, muoversi su Worth non è mai stato così semplice. Quando eseguirai il client Worth, verrai portato alla pagina iniziale di registrazione e login.

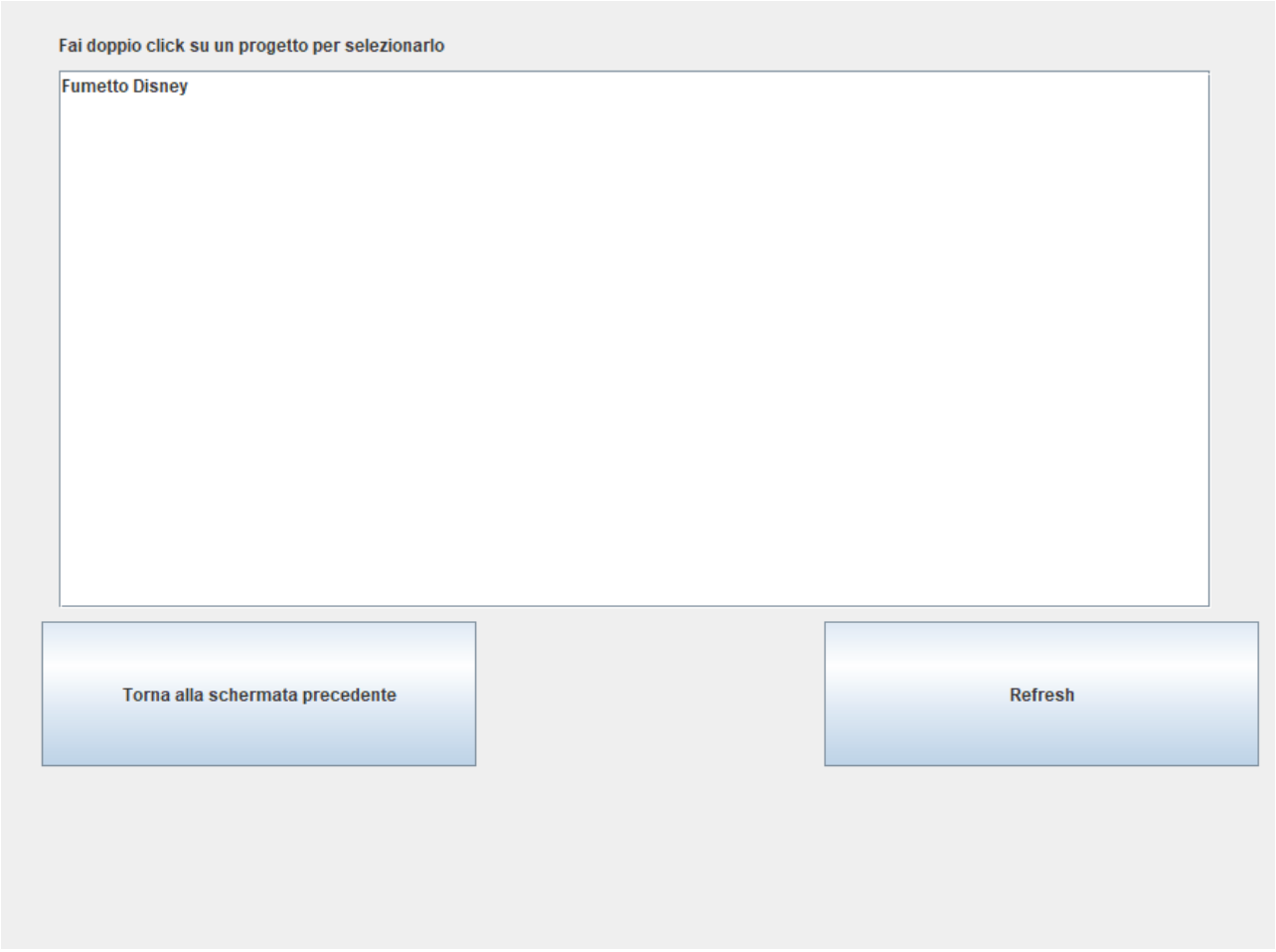


Dopo esserti registrato e aver inserito i tuoi dati, clicca sul pulsante login per accedere alla pagina iniziale di Worth.

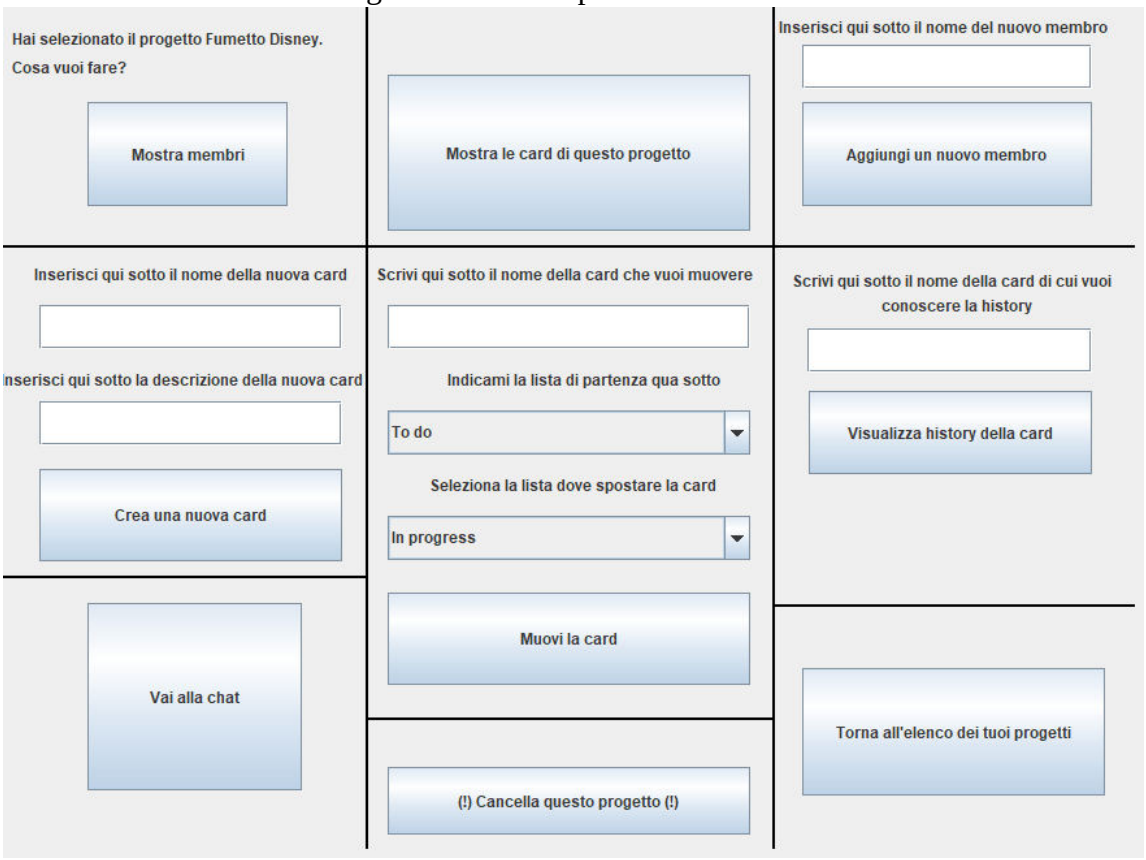


Con i pulsanti a destra sarai in grado di accedere a dei menu di visualizzazione utenti, così da sapere quali utenti registrati a Worth sono online al momento. Con il pulsante Logout puoi disconnetterti da Worth. Con i pulsanti a sinistra sarai invece in grado di creare e visualizzare i tuoi progetti.





Facendo doppio click sul nome di un progetto, accederai alla pagina di visualizzazione di quel progetto. Da lì avrai modo di svolgere numerose operazioni:



Mostra membri: per visualizzare su una lista scorrevole i partecipanti di questo progetto.
Mostra le card di questo progetto: per visualizzare le card del progetto su una lista scorrevole. Fai doppio click sul nome di una card per visualizzarne nome, descrizione e stato attuale.
Aggiungi un nuovo membro: dopo aver scritto nell'area di testo il nome del nuovo membro, clicca sul pulsante per aggiungerlo.
Crea una nuova card: dopo aver specificato il nome della nuova card e la sua descrizione, clicca sul pulsante per aggiungere una nuova card al progetto.
Muovi la card: dopo aver inserito nell'area di testo il nome della card che vuoi muovere, e aver specificato nelle dropdown lists lo stato in cui si trova la card e lo stato in cui vuoi che si trovi, clicca sul pulsante per effettuare lo spostamento.
Visualizza history della card: dopo aver inserito il nome della card nell'area di testo, clicca sul pulsante per vedere i vari stati che la card ha attraversato, compreso quello attuale.
Vai alla chat: per accedere alla chat del progetto.
Cancella questo progetto: per cancellare questo progetto, se e solo se tutte le card sono nello stato "Done".
Torna all'elenco dei tuoi progetti: per tornare all'elenco dei progetti.

Fai doppio click su una card per vederne il contenuto

Prima pagina

Torna alla schermata precedente

Refresh

Nome della card:

Prima pagina

Descrizione:

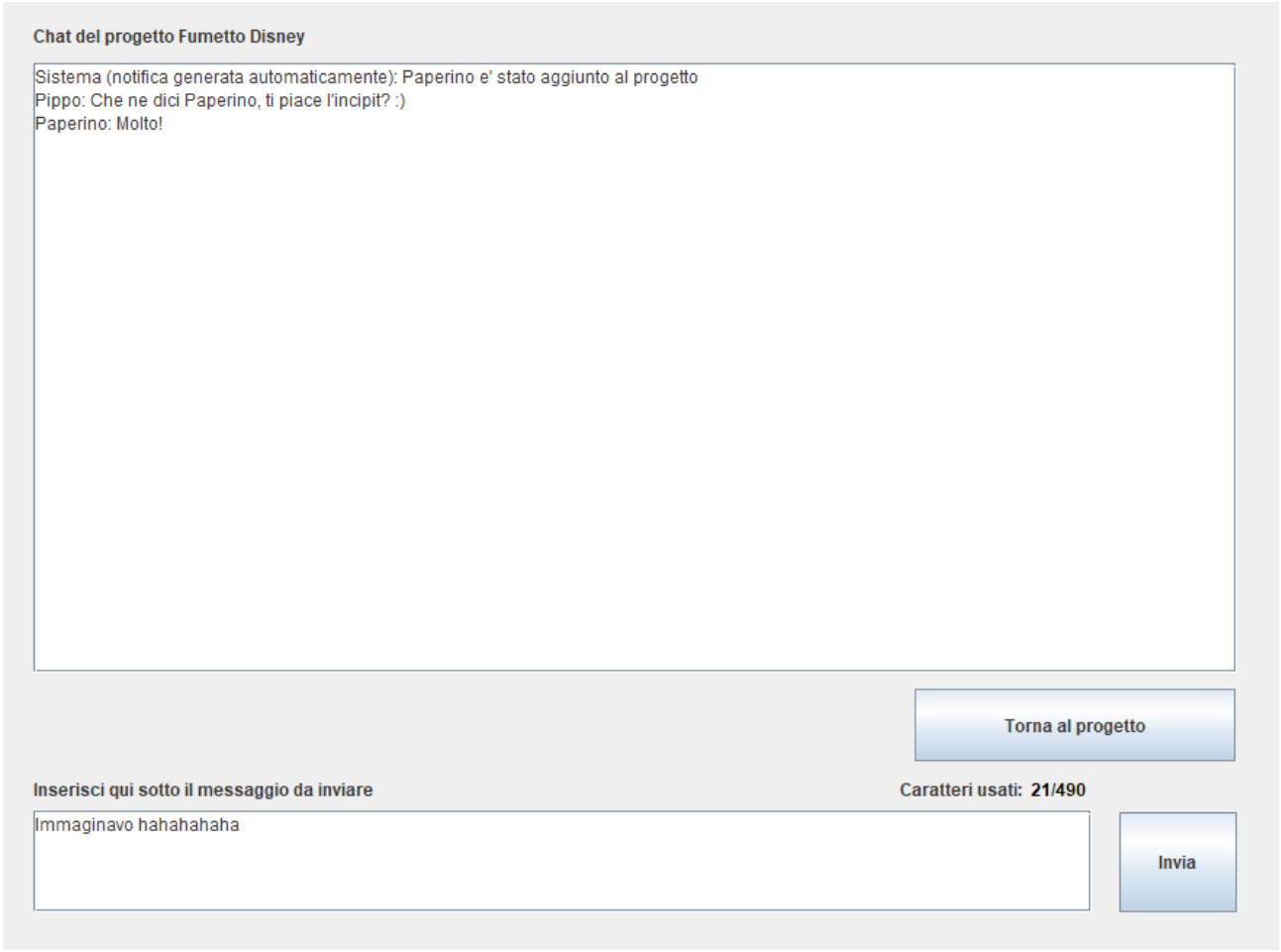
Refresh

Torna all'elenco delle card

Incipit: Paperino per una volta ha fortuna e Gastone sfortuna

Stato attuale (lista in cui si trova adesso questa card):

To do



Spesso troverai un pulsante “refresh” all’interno dell’interfaccia. Premendolo, potrai aggiornare il menu che stai visualizzando con le informazioni più recenti (ad esempio, se stai visualizzando i progetti di cui fai parte, e qualcuno nel frattempo ti ha aggiunto ad un progetto, premendo il pulsante refresh potrai vedere l’elenco aggiornato dei progetti a cui partecipi, compreso l’ultimo a cui sei stato aggiunto).
Le operazioni che richiedono la pressione di un pulsante dopo l’inserimento di informazioni testuali (es. messaggi sulla chat, creazione di una card, etc..) non verranno eseguite se i campi di testo sono vuoti (ad eccezione della descrizione di una nuova card, che può anche essere vuota). Nello spostamento di una card, se lo stato scelto in entrambe le dropdown lists coincide, la pressione del pulsante “Muovi la card” non avrà alcun effetto (non ha senso chiedere di spostare una card dallo stato “In progress” allo stato “In progress”, stesso dicasi per “To be revised”).
Se l’operazione richiesta non può essere soddisfatta (es. creare una nuova card col nome di una card che già esiste nel progetto, aggiungere un utente che già fa parte del progetto/non esiste, spostare la card in uno stato che non rispetta il diagramma degli stati delle card etc.) spunterà a schermo un popup che specifica la ragione per cui non è stato possibile portare a termine la richiesta.

Capitolo 2: Le classi lato server che mantengono informazioni e gli indirizzi di multicast

Premessa valida per quasi ogni classe del progetto: per poter debuggare in fase di sviluppo, ho deciso di dichiarare, nelle classi ove fosse per me necessario, una variabile booleana “DEBUG”. Inoltre, nella classe Constants, è presente un’altra varbiabile booleana GLOBALDEBUG. Se queste due variabili vengono inizializzate a true, le righe di codice if(Constants.GLOBALDEBUG && DEBUG) System.out.println(“testo generico”) verranno eseguite, così da permettere allo sviluppatore di osservare da terminale come procede l’esecuzione del programma. Non hanno altri effetti se non quello di fornire un feedback su terminale di ciò che sta accadendo e di quali righe di codice stanno venendo eseguire. Ho deciso di lasciare queste righe qualora vi fosse necessario debuggare o controllare come certe cose vengono eseguite, ma ho inizializzato la variabile Constants.GLOBALDEBUG a false, lasciando invece quelli locali DEBUG a true così da permettere a voi di decidere, qualora vi fosse necessario, a quali di queste cambiare valore e a quali no. Queste variabili, se presenti in una classe, sono le prime ad essere dichiarate, così da essere subito riconoscibili.

2.1 – La classe ServerInfo

Prima di parlare della struttura generale del server o di come avviene la comunicazione tra esso e il client, ritengo necessario descrivere, in modo riassuntivo, le classi lato server, così da inquadrare gli oggetti che andremo ad utilizzare, le variabili che contengono e i metodi che offrono. Solo dopo aver fatto ciò andremo a vedere come questi strumenti vengono utilizzati. Inoltre, dato che naturalmente “è tutto collegato”, quando vorrò riferirmi ad una particolare variabile di una certa classe lo farò scrivendo NomeClasse.NomeVariabile, come se si trattasse di una variabile statica, anche se naturalmente non sempre lo sarà. Confido che sia chiaro dal contesto e dai paragrafi già letti se si sta parlando di una variabile statica o di una variabile d’istanza. Detto ciò, cominciamo.

La classe ServerInfo è una classe che contiene solo variabili statiche e metodi statici. Le variabili qui presenti si dividono in strutture dati presenti a tempo di esecuzione sul server, costanti e monitor.
UtentiRegistratiWorth: hashmap che ha come chiave il nome di un utente (univoco in tutto Worth), e come valore un oggetto UtenteRegistratoConPassword (vedi 2.2 per maggiori dettagli). Ho scelto una hashmap per conservare questi valori in quanto i nomi degli utenti sono univoci in tutta Worth, e partendo da uno di essi voglio poter ottenere velocemente le informazioni su quell’utente.
ProgettiWorth: hashmap che ha come chiave il nome di un progetto (anch’esso univoco in tutto Worth) e come valore un oggetto WorthVolatileProjectInformations (vedi 2.3 per maggiori dettagli). Ho scelto una hashmap per lo stesso motivo enunciato poco sopra.
ALL_USERS_MONITOR: monitor usato per effettuare in mutua esclusione operazioni che vanno a modificare l’insieme di utenti presenti su Worth (nel capitolo 4 viene descritto in dettaglio come vengono usate queste lock).
ALL_PROJECTS_MONITOR: monitor usato per effettuare in mutua esclusione operazioni che vanno a modificare l’intero insieme di progetti su Worth.
ALL_MULTICAST_ADDRESSES_MONITOR: monitor usato per assicurare che non ci siano collisioni quando si vanno ad associare indirizzi di multicast ai progetti.
UsersDir: variabile che contiene il percorso relativo della directory contenente i file che rappresentano i vari utenti registrati a Worth.

ProjectsDir: variabile che contiene il percorso relativo della directory contenente i progetti di Worth con le loro informazioni (una directory per progetto, dove in ciascuna di esse abbiamo un WorthProjectManager serializzato e un file per ogni card).
RecycledMulticastAddressesList: arraylist contenente gli indirizzi di multicast che erano stati assegnati ad un progetto ma che ora non vi appartengono più, e possono essere riassegnati ad altri progetti. Ho scelto un arraylist in quanto non ho interesse a manipolare questa struttura dati, ma solo ad aggiungere e rimuovere elementi.
InitialMulticastAddress: costante che rappresenta il primo indirizzo di multicast.
finalMulticastAddress: costante che rappresenta l'ultimo indirizzo di multicast.
CurrentMulticastAddress: variabile che rappresenta l'indirizzo di multicast da assegnare al prossimo progetto, qualora non ci fossero indirizzi di multicast da riciclare.

Vediamo ora i metodi:
initializeServerInfo: metodo chiamato dal main del Server per inizializzare alcune delle variabili precedentemente descritte. In particolare, va ad esplorare, nella cartella dove si trova il codice del programma, la cartella WorthServerDirectory (creata dal main del server se è la prima volta che lo si esegue), che conterrà altre due cartelle: UsersDirectory e ProjectsDirecory (anch'esse create dal main la prima volta che lo si esegue). Leggendo il contenuto di queste due cartelle vengono inizializzate le strutture dati utentiRegistratiWorth e progettiWorth con gli utenti che si sono registrati e con i progetti che sono stati creati.
GetMulticastAddressToAssociateToAProject: metodo usato per ottenere un indirizzo di multicast da associare ad un progetto. Tale metodo chiama innanzitutto il metodo **pollRecycledMulticastAddress**, che va a prelevare, se c'è, un indirizzo di multicast dalla lista recycledMulticastAddressList. Se tale lista fosse vuota, viene restituito il valore di currentMulticastAddress, che viene poi incrementato di uno (ovvero, prende il valore del prossimo indirizzo di multicast).
PutRecycledMulticastAddress: metodo usato per inserire nella lista recycledMulticastAddressList un indirizzo di multicast che non è più usato da un progetto.
GetAllUtentiRegistrati: metodo che restituisce una hashmap <nome utente, UtenteRegistrato> contenente l'elenco di tutti i membri di Worth con le relative informazioni pubbliche (il loro nome e il loro stato, online o offline).
DoesUserExist: metodo per verificare se un utente esiste o no
addUser: metodo usato per registrare un utente a Worth, andando a creare un file che lo rappresenti qualora il nome utente scelto dal nuovo utente non sia già stato preso.
SetUserState e **getUserState**: metodi con cui impostare/ottenere lo stato di un utente registrato (online/offline).
GetUtenteRegistrato: metodo per ottenere le informazioni pubbliche di un singolo utente (nome e stato).
UserAuthentication: metodo usato per verificare se la password inserita da un utente registrato è corretta oppure no.
GetUtenteRegistratoConPassword: metodo per ottenere tutte le informazioni, anche quelle private, di un utente (restituisce l'oggetto UtenteRegistratoConPassword associato ad un certo utente).
UpdateExistingUserInformations: metodo usato per aggiornare le informazioni private di un utente (sostituisce il file che lo rappresenta con una versione più recente, e quindi più aggiornata).

2.2 - Le classi UtenteRegistrato e UtenteRegistratoConPassword

Per poter persistere le informazioni degli utenti registrati su Worth, e al contempo trasmettere agli utenti loggati solo le informazioni pubbliche degli utenti (nome e stato, online oppure offline), ho optato per due classi dalle quali vengono istanziati due oggetti per ogni utente.
La classe **UtenteRegistrato** contiene due variabili: una stringa **name** che conserva il nome dell'utente registrato, e lo stato **state**, una variable booleana che associa al valore true il fatto che l'utente è online, e al valore false il suo essere offline. Nel costruttore viene richiesto solo il nome dell'utente, in quanto lo stato è sempre inizializzato come offline. Questo è dovuto al fatto che gli utenti devono prima registrarsi e poi loggarsi.
Ci sono poi metodi setter e getter per queste variabili:
getUtenteRegistratoName: per ottenere il nome (il valore di name) di questo utente.
GetUtenteRegistratoState, getUtenteRegistratoStateAsString e **seUtenteRegistratoState**, per leggere lo stato dell'utente (come valore booleano o come stringa "ONLINE"/"OFFLINE") e impostarlo.

Un oggetto della classe appena descritta viene incapsulato in ogni oggetto della classe **UtenteRegistratoConPassword**, che può essere vista come un contenitore di tutte le informazioni di un utente. Ci sono infatti quattro variabili in questa classe:
user, un'istanza di UtenteRegistrato che contiene le informazioni pubbliche di un utente.
password_sha, che contiene la password cifrata dell'utente (SHA-256).
Seed: un valore numerico generato casualmente nel momento in cui viene istanziato un oggetto di questa classe. Infatti, se questo valore non esistesse e due utenti diversi scegliessero la medesima password, questa verrebbe cifrata nello stesso modo, e ciò potrebbe essere pericoloso. Col seed invece ciò non accade: quando un utente si registra viene generato il seed, concatenato con la sua password e il risultato di questa concatenazione cifrato usando SHA-256. Così anche password uguali difficilmente saranno cifrate in modo uguale.
ListaProgettiDiCuiSonoMembro: hashset di stringhe contenente i nomi dei progetti di cui l'utente fa parte. Ho scelto un hashset per conservare queste informazioni in quanto vorrò controllare velocemente se l'utente fa parte di un progetto o no, oltre al fatto che il nome di un progetto è univoco in tutto Worth (e quindi lo è anche in un sottoinsieme dei progetti esistenti).
Nel costruttore, invocato quando un utente si registra e che chiede come argomenti solo username e password, la password viene concatenata al seed (seed + password) e digerita dal metodo **digestPasswordSHA_256**, che restituisce come risultato il digest della password, salvato nella variabile password_sha. Anche in questo caso abbiamo metodi getters e setters per le variabili:
confrontUsername: per vedere se un certo nome utente inserito coincide con quello della variabile user.name di questo oggetto.
ConfrontPassword: concatena la password inserita al seed e la digerisce, restituendo true se il risultato coincide col valore di password_sha, false altrimenti.
GetUtenteRegistrato: per ottenere il valore di user.
AddProjectToListOfUsersProjects, removeProjectsFromListOfUsersProject e **getProjectsOfUser** per aggiungere, rimuovere e leggere un progetto/i progetti alla/dalla lista dei progetti di cui l'utente fa parte.

2.3 - Gestione degli indirizzi di multicast e le classi WorthProjectManager, WorthVolatileProjectInformations e WorthCard

Vogliamo avere una directory per progetto che contenga i file rappresentativi delle card di quel progetto. Una domanda che mi sono subito posto è: come faccio a ricordarmi le liste dei quattro stati di una card (To do, In Progress, To be revised e Done) e i partecipanti di un progetto? Ho deciso, per risolvere questi problemi, di creare una classe che mantenesse queste informazioni: la

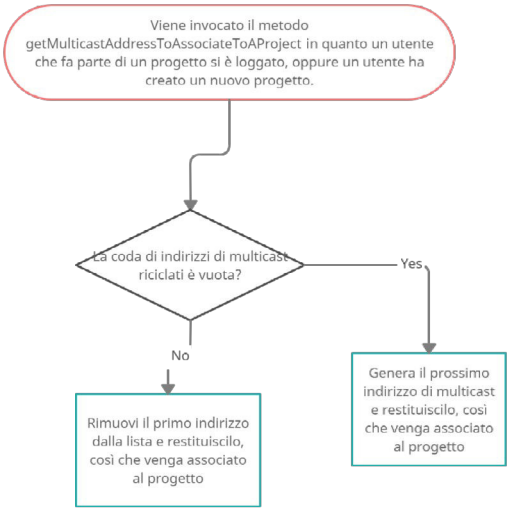
classe **WorthProjectManager**. Nel momento in cui viene creato un nuovo progetto, non solo viene creata una directory ad esso dedicata, ma anche un oggetto di questa classe, che ha le seguenti variabili:
projectName: il nome del progetto.
ListMembersName: hashset dei membri di questo progetto. Ho scelto un hashset in quanto non possono esserci due utenti con lo stesso nome all'interno di Worth.
ToDo, inProgress, toBeRevised e **done**: quattro linkedlists contenenti i nomi delle card. Le card che si trovano nello stato To Do avranno il loro nome memorizzato nella prima lista, quelle nello stato In Progress nella seconda, e così via. Ho scelto delle linkedlist in quanto volevo poter muovere con facilità le card da una lista all'altra (rimozione e aggiunga di un elemento da/ad una lista), senza però rinunciare all'ordine con cui queste sono state aggiunte ad una lista.

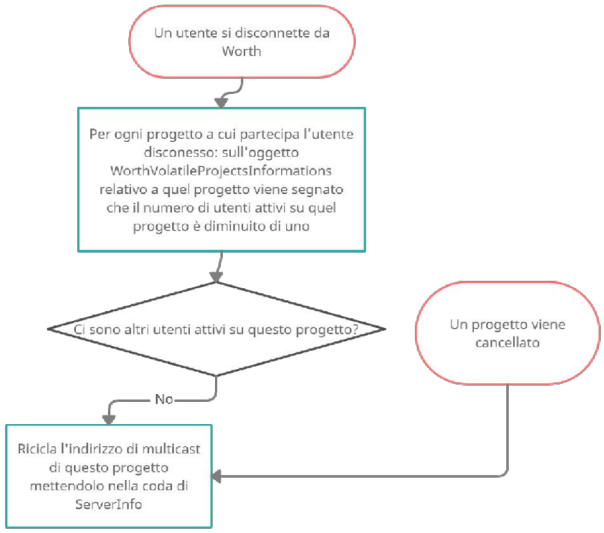
Dopo di che mi sono interrogato su come gestire l'associazione degli indirizzi di multicast ai progetti. Mi sono venute in mentre tre idee che potessero risolvere questo problema, ciascuna l'evoluzione della precedente:
Idea numero 1) Associare un indirizzo di multicast ad ogni progetto.
Pro: meccanismo facile da implementare. Si parte dal valore iniziale di ServerInfo.initialMulticastAddress e lo si incrementa di uno (seguendo il formato corretto, in quanto si tratta di un indirizzo e non di un intero) ogni volta che si è associato l'indirizzo corrente ad un progetto.
Contro: gli indirizzi finiranno velocemente, e non verranno riutilizzati quando un progetto verrà cancellato.
Idea numero 2) Come sopra, ma ogni volta che un progetto viene cancellato si ricicla l'indirizzo di multicast di quel progetto, che verrà messo in una coda (ServerInfo.recycledMulticastAddressesList) a cui verrà data la priorità di lettura quando si va ad associare un nuovo indirizzo di multicast ad un progetto.
Pro: meccanismo relativamente facile da implementare. Basta seguire il procedimento dell'idea precedente estendendolo con un meccanismo di riciclaggio: quando un progetto viene eliminato, salva nella coda il suo indirizzo di multicast. Quando bisogna associare un nuovo indirizzo ad un progetto, si controlla la coda. Se non è vuota, si usa il suo primo elemento come indirizzo, altrimenti, si genera un nuovo indirizzo di multicast.
Contro: se vengono creati più progetti di quanti ne vengano cancellati gli indirizzi di multicast rischiano di non bastare.
Idea numero 3) Manteniamo l'idea del riciclo degli indirizzi, ma estendiamola cambiando la nostra concezione di "indirizzi associati ad un progetto". Gli indirizzi di multicast servono agli utenti per usare la chat di un progetto, ma se nessun utente sta usando un determinato progetto in un certo istante, avergli associato un indirizzo di multicast è uno spreco. Pertanto, associamo indirizzi solo a quei progetti attivi, cioè usati da almeno un utente (ovvero, se l'utente Pippo partecipa ai progetti P1, P2 e P3 e Pippo si logga su Worth, P1, P2 e P3 sono considerati progetti attivi).
Pro: gli stessi dell'idea 2, con l'aggiunta che stavolta un indirizzo di multicast viene separato da un progetto non solo quando quest'ultimo viene cancellato, ma anche quando non viene usato da alcun utente.
Contro: gli indirizzi di multicast potrebbero comunque finire, ma questo è un limite degli indirizzi stessi.

Questa mia terza idea si è concretizzata nella classe **WorthVolatileProjectInformations**, che comprende due variabili:
multicast_address_of_this_project, che contiene l'indirizzo di multicast associato a questo progetto.
number_of_users_using_this_project, intero che indica quanti utenti che partecipano a questo progetto sono online.

Il costruttore inzializza la variabile multicast_address_of_this_project con null, mentre number_of_users_using_this_project è inizializzato a 0.
I metodi, tutti sincronizzati sull'oggetto, vanno a modificare o a leggere le sue variabili:
incrementNumberOfUsersUsingThisProject: incrementa di uno il valore della variabile number_of_users_using_this_project. Quando viene chiamato questo metodo, almeno un utente starà usando questo progetto, pertanto, se multicast_address_of_this_project è a null, viene chiamato ServerInfo.getMulticastAddressToAssociateToAProject per associare un indirizzo di multicast al progetto.
DecrementNumberOfUsersUsingThisProject: decrementa di uno il valore della variabile number_of_users_using_this_project. Se questo valore scende a 0, l'indirizzo di multicast viene separato da questo progetto chiamando il metodo ServerInfo.putRecycledMulticastAddress, e la variabile multicast_address_of_this_project settata a null.
GetNumberOfUsersUsingThisProject e **getMulticastAddressOfThisProject**: per leggere il valore delle variabili in mutua esclusione.
FreeMulticastAddressBecauseProjectIsBeingCancelled: chiamato quando il progetto viene cancellato così da separare lo stesso dal suo indirizzo di multicast e metterlo nella coda di indirizzi riciclati.

In questa descrizione sembra però mancare un elemento: come fa un oggetto WorthVolatileProjectInformations a sapere a quale progetto è associato?
Per rispondere, dobbiamo andare a vedere la classe ServerInfo che contiene la hashmap progettiWorth, che associa il nome di ogni progetto all'oggetto WorthVolatileProjectInformations. Quando il server parte e esistono già delle directory relative a dei progetti, queste vengono lette, al loro nome viene associato un nuovo oggetto WorthVolatileProjectInformations, e la coppia inserita nella hashmap. Quando un utente che fa parte di alcuni progetti si logga, per ogni suo progetto viene chiamato il metodo WorthVolatileProjectsInformations.incrementNumberOfUsersUsingThisProject. Se al progetto non era ancora stato assegnato un indirizzo di multicast, gli viene assegnato adesso chiamando ServerInfo.getMulticastAddressToAssociateToAProject. Quando viene creato un nuovo progetto (vedremo come quando discuteremo della classe ServerTcpOperations) viene creata la directory per quel progetto, un WorthProjectManager e una sua entry nella struttura dati ServerInfo.progettiWorth. Dato che un progetto appena creato è sicuramente usato da un utente, viene subito chiamato il metodo incrementNumberOfUserUsingThisProject sul WorthVolatileProjectsInformations relativo al nuovo progetto. Quando un progetto viene cancellato, la directory relativa al progetto e tutti i suoi contenuti vengono cancellati, l'indirizzo di multicast messo nella coda di indirizzi da riciclare e la entry del progetto cancellato rimossa da ServerInfo.progettiWorth.
Di seguito dei diagrammi UML che spiegano il flusso di esecuzione relativo alla gestione degli indirizzi multicast.





Veniamo ora alle card. Una card in Worth è rappresentata da un oggetto WorthCard, che contiene quattro variabili:
name: il nome della card.
CurrentState: stringa che rappresenta lo stato corrente della card (To do, In Progress, To be revised e Done).
Description: la descrizione della card.
History: un arraylist contenente gli spostamenti della card da uno stato all'altro. Ho scelto un arraylist in quanto voglio mantenere l'ordine degli spostamenti, inserendo sempre il nuovo elemento in coda.

Capitolo 3: Persistenza delle informazioni mediante serializzazione

3.1 – Oggetti serializzati

Worth deve persistere le informazioni degli utenti registrati, così come i progetti creati e le informazioni ad essi relative. Per fare ciò, ho deciso di utilizzare la serializzazione degli oggetti mediante Gson. In particolare gli oggetti che vengono serializzati e deserializzati per rendere consistenti (e persistenti) le informazioni sopra citate sono:
UtenteRegistratoConPassword, che a sua volta conterrà un oggetto **UtenteRegistrato**;
WorthProjectManager;
WorthCard.
Queste informazioni vengono persistite sul file system nella cartella *WorthServerDirectory*, creata dal programma quando il server viene eseguito per la prima volta. All'interno di essa troviamo due cartelle: *UsersDirectory* e *ProjectsDirectory.* La prima contiene i file serializzati json degli oggetti UtenteRegistratoConPassword, la seconda è un po' più complicata. Per ogni progetto esistente su Worth, questa cartella contiene un'altra cartella che porta il nome del progetto. Dentro di essa abbiamo il WorthProjectManager serializzato come file json che ha lo stesso nome della directory, e tutte le card WorthCard serializzate come file json, che hanno come nome quello della card. Per questo motivo, un utente non può, all'interno di un progetto, chiamare una card con lo stesso nome del progetto stesso (andrebbe a sovrascrivere il WorthProjectManager del progetto altrimenti).
Inoltre, ho deciso di stabilire il seguente formato di serializzazione: quando un oggetto deve essere serializzato, prima si scrive nel file la lunghezza dell'oggetto serializzato in byte, e poi l'oggetto serializzato vero e proprio. Così facendo, quando devo andare a deserializzare, posso prima leggere un intero, grazie al quale vado ad allocare un array di byte della lunghezza giusta, e poi l'oggetto serializzato vero e proprio, che inserisco nell'array, converto in stringa e trasformo in oggetto da json. Per le scritture e le letture verso/da disco ho usato il meccanismo degli stream, usando in particolare i FileStream, i BufferedStream e i DataStream

3.2 – La prima esecuzione del server

Quando il main del server viene eseguito per la prima volta, la prima cosa che fa è accertarsi che esistano le cartelle sopracitate che contengono o conterranno le informazioni da persistere. Siccome stiamo parlando della sua primissima esecuzione, dove cioè non esistono ancora queste cartelle, il main andrà a creare la cartella WorthServerDirectory. Al suo interno creerà le directory UsersDirectory e ProjectsDirectory

3.3 - Le successive esecuzioni del server

Quando il server viene eseguito dopo la prima volta, le cartelle esistono già, e potrebbero contenere informazioni serializzate sugli utenti e sui progetti. Pertanto le cartelle non vengono create, ma esplorate. Il main del server chiama quindi ServerInfo.InitializeServerInfo che va ad esplorare le due cartelle. Prima controlla quella degli utenti. Per ogni file contenuto in essa (se ce ne sono), il file viene deserializzato e viene creato un oggetto UtenteRegistratoConPassword che ha nel campo "name" del sotto-oggetto UtenteRegistrato il nome del file deserializzato, escludendo l'estensione ".json". Viene creata quindi una coppia <nome, UtenteRegistratoConPassword> che viene inserita in ServerInfo.utentiRegistratiWorth.
Poi viene esplorata la cartella dei progetti, dove però non avviene nessuna deserializzazione. Si leggono semplicemente i nome delle directory in essa contenute, e per ogni nome di progetto letto si procede a creare un nuovo oggetto WorthVolatileProjectInformations che non ha un indirizzo di multicast associato e ha 0 utenti che lo stanno utilizzando. Si aggiunge infine la coppia <nome progetto, WorthVolatileProjectInformations> alla struttura dati ServerInfo.progettiWorth. Il motivo per cui non vado a deserializzare i file contenuti in queste cartelle è che non ritengo di averne bisogno. Un gran numero di progetti e card comporterebbe un grosso impiego di memoria da parte del server, se dovesse deserializzare tutti i file contenuti nella directory dei progetti di Worth. Preferisco piuttosto andare a deserializzare e serializzare questi file quando sarà strettamente necessario, ovvero quando un utente farà delle operazioni sui progetti o sulle card.

3.4 - Quando i file vengono serializzati

Anche se parleremo nel dettaglio delle operazioni lato server in uno dei prossimi capitoli, ci tengo a dare un'idea di come avviene il meccanismo di serializzazione e deserializzazione dei file. Una delle mie più grandi preoccupazioni fin dall'inizio dello sviluppo è stata: come faccio ad assicurarmi che le informazioni presenti nella memoria del processo server a tempo di esecuzione (e quindi volatili) siano consistenti con quelle presenti su disco?
Ho risposto a questa domanda nel seguente modo: su Worth, un'operazione di modifica applicata a un utente, un progetto o una card non si può considerare finita finché il contenuto della memoria del processo non coincide con quello dei file serializzati. Prendiamo come esempio il funzionamento del metodo moveCard, andando ad anticiparne il comportamento.
Il server riceve da un client la richiesta di spostamento di una card da una lista ad un'altra. Se la richiesta è soddisfacibile, il server, prima di rispondere al client, esegue le seguenti operazioni:
-deserializza il WorthProjectManager di questo progetto per controllare se la richiesta è soddisfacibile. Se non lo è si blocca subito e restituisce un codice di errore al client (non serve riscrivere il WorthProjectManager in quanto lo si è letto senza modificarlo), altrimenti si continua;
-il server rimuove la card dalla lista in cui si trova e la aggiunge alla lista di destinazione richiesta dal client;
-il server serializza il WorthProjectManager per rendere persistenti le modifiche effettuate;
-il server deserializza la card in questione;
-aggiorna il suo stato e la sua history;
-la card viene serializzata, andando a sovrascrivere il file precedente che la rappresentava.
Pertanto, solo una volta che i cambiamenti richiesti da un client sono stati resi persistenti sul file system del server è possibile confermare il successo dell'operazione.

Capitolo 4: Monitor e metodi lato server

4.1 – Introduzione ai monitor usati

Per la sincronizzazione ho deciso di utilizzare i monitor (i blocchi synchronized) anziché le lock esplicite in quanto queste ultime le ho utilizzate pesantemente nel progetto di Sistemi Operativi e le

trovo più complesse da gestire rispetto ai monitor. Inoltre, essendo i monitor un concetto per me nuovo, avendolo incontrato proprio durante questo corso, ho deciso di mettermi alla prova e provare a usarlo al meglio… e devo dire che ne sono felice, in quanto ho scoperto degli utilizzi stravaganti di queste lock implicite che saranno trattati approfonditamente nel paragrafo 4.3 e 4.4.
Parliamo adesso dei monitor usati: ho deciso di utilizzare in totale quattro tipi di monitor per assicurare la mutua esclusività delle operazioni effettuate sulle risorse condivise del server. I primi tre sono a grana grossa, mentre l'ultima tipologia è a grana fine. Tutti i monitor sono situati nella classe ServerInfo, e parliamo di:
ALL_USERS_MONITOR: monitor usato per accedere in mutua esclusione alla struttura dati utentiRegistratiWorth e in generale per avere accesso esclusivo alle informazioni degli utenti (tra cui quindi anche la cartella contente gli oggetti UtenteRegistratoConPassword serializzati).
ALL_PROJECTS_MONITOR: monitor usato per accedere in mutua esclusione ai dati relativi a tutti i progetti di worth, come la struttura dati progettiWorth o la cartella contenente le directory dei progetti.
ALL_MULTICAST_ADDRESSES_MONITOR: monitor usato per assicurare che non ci siano collisioni nell'assegnamento degli indirizzi di multicast
I valori della hashmap progettiWorth: se le operazioni di modifica di un progetto esistente fossero state effettuate in mutua esclusione usando la lock ALL_PROJECTS_MONITOR, si sarebbe perso un sacco del parallelismo possibile, al quale non volevo rinunciare. Pertanto ho deciso di distinguere il concetto di "Operazioni **sui** progetti" dal concetto di "Operazione **su un** progetto". Nel primo tipo di operazioni sarà necessario detenere la lock a grana grossa ALL_PROJECT_MONITOR, nel secondo basterà quella a grana fine, ovvero sarà necessario sincronizzarsi sul WorthVolatileProjectInformations relativo al progetto che si vuole andare a modificare.

4.2 - Associazione monitor-operazioni lato server

Dobbiamo ora per forza di cose introdurre la classe ServerTcpOperations, che contiene quasi tutti i metodi specificati nel testo del progetto Worth. Tuttavia parleremo del loro funzionamento a livello molto astratto, in quanto ci interessa adesso vedere quali monitor sono usati durante l'esecuzione di questi metodi per garantire la mutua esclusione delle operazioni. Parleremo più approfonditamente dei singoli metodi in un capitolo successivo.

Metodo	Monitor su cui si sincronizza e motivazione
serverLogin(username, password)	ALL_USERS_MONITOR. È necessario sincronizzarsi su questo monitor in quanto vogliamo evitare che un utente registrato possa loggarsi da due client differenti su Worth (se infatti un utente è online e tenta di loggarsi da un altro client, l'operazione viene respinta).
serverLogout(username)	Questa operazione non è sincronizzata su nessun monitor, in quanto, avendo garantito che l'utente possa loggarsi solo da un client nel metodo precedente, non è possibile che si crei una race condition legata al logout.
serverCreateProject(project_name, creator)	ALL_PROJECTS_MONITOR. Se non avessimo un monitor su tutti i progetti, potremmo rischiare una race condition in cui due utenti diversi creano, nello stesso istante, due progetti con lo stesso nome. Le conseguenze potrebbero essere imprevedibili. Si tratta di una "Operazione **sui**

	progetti” quindi.
serverAddMember(project_name, new_member)	ServerInfo.progettiWorth.get(project_name). PREMESSA: ricordo che con la scrittura sopra si intende che la sincronizzazione avviene sull’oggetto WorthVolatileProjectInformations associato al progetto project_name, e tale coppia è situata nella struttura dati progettiWorth nella classe ServerInfo. Questo è il primo di molti metodi che sfruttano il monitor a grana fine per la sincronizzazione. Infatti, una volta che un progetto è stato creato (garantendo la mutua esclusione del nome grazie a serverCreateProject) non è necessario acquisire la lock ALL_PROJECTS_MONITOR, in quanto l’operazione di aggiunta di un utente a un progetto esistente è un’operazione che modifica solo i dati del progetto project_name. È quindi una “Operazione **su un** progetto”.
serverShowMembers(project_name)	ServerInfo.progettiWorth.get(project_name). Per evitare che l’utente legga informazioni non aggiornate o parziali sui membri del progetto. Questa operazione potrebbe infatti, se non sincronizzata, andare in conflitto con serverAddMember.
serverShowCards(project_name)	ServerInfo.progettiWorth.get(project_name). Per evitare che l’utente legga informazioni parziali dall’elenco delle card del progetto project_name.
serverShowCard(project_name, card_name)	ServerInfo.progettiWorth.get(project_name). Per evitare di leggere informazioni non consistenti di una card.
serverAddCard(project_name, card_name, description)	ServerInfo.progettiWorth.get(project_name). Per evitare la race condition in cui due utenti del progetto, nello stesso momento, decidono di creare due card con lo stesso nome.
serverMoveCard(project_name, card_name, starting_list, destination_list)	ServerInfo.progettiWorth.get(project_name). Per evitare che due utenti, nello stesso momento, tentino di spostare la stessa card in due liste diverse.
serverGetCardHistory(project_name, card_name)	ServerInfo.progettiWorth.get(project_name). Per evitare di leggere informazioni non consistenti di una card.
serverCancelProject(project_name)	In ordine: ALL_PROJECTS_MONITOR, ServerInfo.progettiWorth.get(project_name), ALL_USERS_MONITOR. Questa a livello di monitor è stata sicuramente l’operazione più difficile da gestire. Il primo monitor è necessario a gestire la race

	condition in cui un utente tenta di cancellare un suo progetto e un altro utente tenta di crearne uno con lo stesso nome. Il secondo serve a garantire che nessun'altra operazione possa essere svolta sul progetto mentre questo sta per essere cancellato, altrimenti si rischiano danni gravi come la lettura da file inesistenti o la scrittura in directory che non esistono più. La terza è dovuta al fatto che ogni UtenteRegistratoConPassword contiene una lista dei progetti di cui l'utente fa parte, da cui devo eliminare il progetto che sta venendo cancellato. Se così non facessi, potrebbero esserci delle inconsistenze.

Non è stata citata l'operazione di registrazione, che non è presente nella classe ServerTcpOperations, bensì si trova nell'oggetto remoto RegistrationImplementation. Questa operazione, senza troppe sorprese, è sincronizzata sul monitor ALL_USERS_MONITOR, così da evitare una race condition in cui due utenti con lo stesso username vogliono registrarsi nello stesso momento.

Se avete letto la relazione fino a qui con attenzione vi sarà balenato un dubbio in testa: la sincronizzazione a grana fine avviene sulle entry di ServerInfo.progettiWorth. Ma cosa succede se un utente si sincronizza su una entry che viene cancellata a seguito della cancellazione del progetto? Risponderemo a questo domanda subito, nei due paragrafi che seguono, osservando un interessante fenomeno legato alla sincronizzazione tra thread.

4.3 – Un caso particolare di uso dei monitor

Nel corso dello sviluppo di Worth mi sono imbattuto nella domanda: se un thread si sincronizza su un monitor, e mentre esegue il blocco di codice sincronizzato il monitor viene cancellato, che cosa succede al thread nel blocco? E se altri thread sono in attesa che il monitor si liberi, cosa fanno quando "si rendono conto" che il monitor non esiste più? Per rispondere a questa domanda, consideriamo il seguente codice, diviso in un main e un thread (riporto il codice come immagine e testo, così da renderlo più leggibile ma allo stesso tempo permettere a chi sta leggendo di provarlo sulla propria macchina):

```java
import java.util.HashMap;

public class ProvaLock {

    public static void main(String[] args) {

        HashMap<String, String> hp = new HashMap<String, String>();
        hp.put("a", "Oggetto a");
        for(int i=0; i<10; i++) {
            Thread t = new Thread(new ThreadDiProva(hp));
            t.start();
        }
    }
}
```

import java.util.HashMap;

public class ProvaLock {

```
	public static void main(String[] args) {

		HashMap<String, String> hp = new HashMap<String, String>();
		hp.put("a", "Oggetto a");
		for(int i=0; i<10; i++) {
			Thread t = new Thread(new ThreadDiProva(hp));
			t.start();
		}
	}
}
```

```
import java.util.HashMap;

public class ThreadDiProva implements Runnable{

    HashMap<String, String> hp = new HashMap<String, String>();

    public ThreadDiProva(HashMap<String, String> hp) {
        this.hp = hp;
    }

    public static int counter = 0;

    @Override
    public void run() {
        synchronized(hp.get("a")) {
            System.out.println("Counter: " + counter++);
            hp.remove("a");
        }
    }
}
```

```
import java.util.HashMap;

public class ThreadDiProva implements Runnable{

	HashMap<String, String> hp = new HashMap<String, String>();

	public ThreadDiProva(HashMap<String, String> hp) {
		this.hp = hp;
	}

	public static int counter = 0;

	@Override
	public void run() {
		synchronized(hp.get("a")) {
			System.out.println("Counter: " + counter++);
			hp.remove("a");
		}
	}
}
```

Riporto adesso alcuni risultati dell'esecuzione di questo codice:

```
Counter: 0
Counter: 1
Counter: 2
Counter: 3
Exception in thread "Thread-9" Counter: 4
Counter: 5
Counter: 6
java.lang.NullPointerException
Counter: 7
Counter: 8
        at ThreadDiProva.run(ThreadDiProva.java:15)
        at java.base/java.lang.Thread.run(Thread.java:830)
```

```
Counter: 0
Counter: 1
Counter: 2
Exception in thread "Thread-8" Counter: 3
Counter: 4
Exception in thread "Thread-9" Exception in thread "Thread-7" Counter: 5
java.lang.NullPointerException
Counter: 6
        at ThreadDiProva.run(ThreadDiProva.java:15)
        at java.base/java.lang.Thread.run(Thread.java:830)
java.lang.NullPointerException
        at ThreadDiProva.run(ThreadDiProva.java:15)
        at java.base/java.lang.Thread.run(Thread.java:830)
java.lang.NullPointerException
        at ThreadDiProva.run(ThreadDiProva.java:15)
        at java.base/java.lang.Thread.run(Thread.java:830)
```

Come era lecito aspettarsi, vengono sollevate delle NullPointerException, ma non sempre: è come se alcuni thread rilevassero il monitor come esistente anche quando non lo è. Infatti, dopo che il primo thread ha stampato “Counter: 0”, il monitor non esiste più, eppure il counter a volte viene comunque incrementato dagli altri thread, che non lanciano una NullPointerException. Ciò risulta ancora più chiaro modificando leggermente il codice del thread come segue:

```
import java.util.HashMap;

public class ThreadDiProva2 implements Runnable{

    HashMap<String, String> hp = new HashMap<String, String>();

    public ThreadDiProva2(HashMap<String, String> hp) {
        this.hp = hp;
    }

    public static int counter = 0;

    @Override
    public void run() {
        try {
            synchronized(hp.get("a")) {

                System.out.println("Counter: " + counter++);
                hp.remove("a");

            }
        }catch(NullPointerException e) {
            System.out.println("Ouch");
        }
    }
}
```

import java.util.HashMap;

public class ThreadDiProva2 implements Runnable{

	HashMap<String, String> hp = new HashMap<String, String>();

	public ThreadDiProva2(HashMap<String, String> hp) {
		this.hp = hp;
	}

	public static int counter = 0;

	@Override
	public void run() {
		try {
			synchronized(hp.get("a")) {
				System.out.println("Counter: " + counter++);
				hp.remove("a");
			}
		}catch(NullPointerException e) {
			System.out.println("Ouch");
		}
	}
}

Di nuovo, ecco il risultato di alcune esecuzioni:

```
Counter: 0
Counter: 1
Counter: 2
Ouch
Counter: 3
Counter: 4
Counter: 5
Counter: 6
Counter: 7
Counter: 8
```

```
Counter: 0
Counter: 1
Counter: 2
Ouch
Ouch
Counter: 3
Ouch
Counter: 4
Counter: 5
Counter: 6
```

```
Counter: 0
Counter: 1
Counter: 2
Counter: 3
Counter: 4
Counter: 5
Counter: 6
Counter: 7
Counter: 8
Counter: 9
```

Il monitor viene comunque cancellato dal primo thread, ma **alcuni thread, prima di acquisire il monitor e eseguire il blocco sincronizzato, riescono a leggere il valore del monitor. Questi thread quindi prima leggono il monitor, poi si sospendono, e poi eseguono il blocco sincronizzato, come se il monitor esistesse ancora, ma di fatto non esiste più.** Il risultato è estremamente imprevedibile, come mostrano le immagini. Questo però non è ciò che vogliamo: vorremmo che solo il primo thread riuscisse a stampare il valore di counter, mentre tutti gli altri no. Dopo un po' di riflessioni sono dunque arrivato al seguente codice:

```
import java.util.HashMap;

public class ThreadDiProva3 implements Runnable{

    HashMap<String, String> hp = new HashMap<String, String>();

    public ThreadDiProva3(HashMap<String, String> hp) {
        this.hp = hp;
    }

    public static int counter = 0;

    @Override
    public void run() {
        try {
            synchronized(hp.get("a")) {
                if(hp.containsKey("a")) {
                    System.out.println("Counter: " + counter++);
                    hp.remove("a");
                }else {
                    throw new NullPointerException();
                }
            }
        }catch(NullPointerException e) {
            System.out.println("Ouch");
        }
    }
}
```

```
import java.util.HashMap;

public class ThreadDiProva3 implements Runnable{

	HashMap<String, String> hp = new HashMap<String, String>();

	public ThreadDiProva3(HashMap<String, String> hp) {
		this.hp = hp;
	}

	public static int counter = 0;

	@Override
	public void run() {
		try {
			synchronized(hp.get("a")) {
				if(hp.containsKey("a")) {
					System.out.println("Counter: " + counter++);
					hp.remove("a");
				}else {
					throw new NullPointerException();
				}
			}
		}catch(NullPointerException e) {
			System.out.println("Ouch");
		}
	}
}
```

Con questa modifica, l'output è sempre e soltanto uno:

Così facendo, anche i thread che sono riusciti a leggere il valore del monitor prima che questo venisse cancellato non compiono azioni improprie. È come se, dopo essere entrati nel blocco sincronizzato, i thread si chiedessero: “aspetta, ma il monitor che sto tenendo esiste ancora?” e se la risposta è negativa, lanciano una NullPointerException, come se avessero letto il valore NULL del monitor.

4.4 - Implicazioni del caso precedente nello sviluppo di Worth

In Worth, come già descritto, per garantire una sincronizzazione a grana fine ho deciso che le operazioni su un progetto esistente si sarebbero svolte in mutua esclusione solo su quel progetto, sincronizzandosi sul WorthVolatileProjectManager di quel progetto. Ma in fase di cancellazione del progetto, quest’oggetto viene cancellato. Come garantire allora che i thread che hanno già letto il valore di quel monitor non compiano azioni improprie? Semplice: dopo aver acquisito il monitor, il thread effettua un controllo circa la sua esistenza o meno, con la riga di codice if(ServerInfo.progettiWorth.containsKey(project_name)). Se il controllo va a buon fine, il metodo continua la sua esecuzione normale. Altrimenti viene lanciata una NullPointerException che viene subito catchata, e che ha come esito quello di restituire al client un messaggio di errore, segnalandogli che non è stato possibile soddisfare la sua richiesta in quanto il progetto non esiste più. Questo controllo viene effettuato in tutti i metodi sincronizzati su ServerInfo.progettiWorth.get(project_name), elencati nel paragrafo 2 di questo capitolo.

Capitolo 5: Funzionamento del server

5.1 – Multiplexing dei canali mediante NIO: la classe ServerWorthMultiplexerNio

Lato server, per ricevere le richieste dai client e per rispondergli, ho deciso di implementare il multiplexing dei canali mediante NIO. In particolare, nel ServerWorthMain, viene lanciato un thread della classe **ServerWorthMultiplexerNio**. Senza troppe sorprese, la classe crea un ServerSocketChannel, chiama la open su di esso, prende il ServerSocket associato e ci lega (bind) l’indirizzo ip del localhost e la porta su cui è in ascolto (presente nella classe Constants, classe che racchiude una serie di costanti all’interno di Worth). Il ServerSocketChannel viene quindi impostato in modalità non blocking, viene creato un selector e su questo viene registrato il ServerSocketChannel, interessato ad operazioni di accept. Inizia quindi un ciclo while(true) dove il selector invoca la select() bloccante. Quando si sblocca, va a prendere l’insieme delle ready keys e itera su di esse. Per ogni key si va a vedere quale operazione è pronta e si procede nel modo appropriato:
key.isAcceptable(): in questo caso è possibile instaurare la connessione con il client. Si prende quindi il SocketChannel relativo alla comunicazione con questo nuovo client e lo si setta a non blocking, così da poterlo registrare sul selettore. Dopo averlo registrato, si crea un oggetto ServerWorthObjectAttached che viene associato alla key appena creata. I dettagli di questo oggetto

verranno esaminati nel quarto paragrafo di questo capitolo. Si tratta comunque, in breve, di un oggetto che permette al server di ricordarsi “a che punto era arrivato” nella gestione della richiesta di un client.
key.isReadable(): in questo caso si ha che il client sta inviando dei dati al server, richiedendogli di svolgere un’operazione. Viene dunque chiamato il metodo readDataFromClient, che identifica l’operazione da svolgere, legge gli argomenti dell’operazione, li salva nell’attachment della key e, dopo aver letto tutti i dati dal client, lancia un thread che si occupi di gestire la sua richiesta.
key.isWritable(): quando il thread lato server che si occupa di soddisfare la richiesta del client ha terminato la sua esecuzione, l’interest set della key viene impostato come interessato ad operazioni di scrittura. Quando insomma il server è pronto per rispondere al client, viene chiamato il metodo sendSomethingToClient che, in base al contenuto dell’attachment, determina cosa mandare al client: un intero oppure una stringa, che spesso è un oggetto serializzato o una lista di oggetti serializzati.
Vediamo ora più nel dettaglio come si svolge la comunicazione client-server, come quest’ultimo gestisce le richieste del client e come funziona l’attachment.

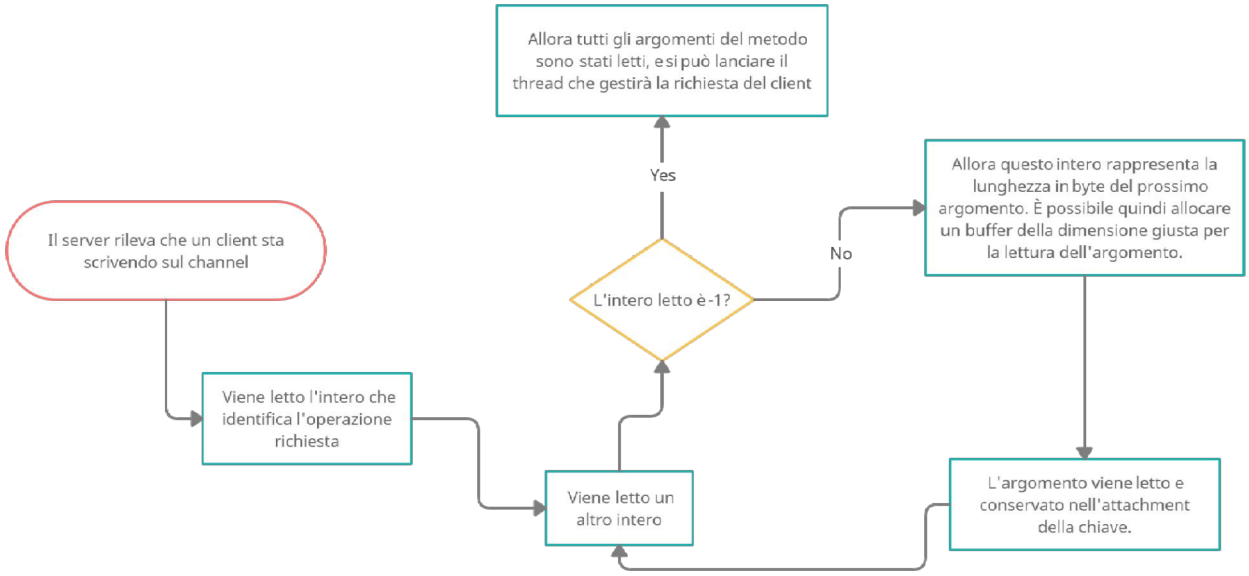
5.2 – Automa a stati finiti per la comunicazione client-server

La comunicazione tra client e server è stata forse la prima cosa a cui ho dovuto pensare. Il server mette a disposizione del client una varia gamma di operazioni, ciascuna con i suoi argomenti. Ma come può il client indicare al server quale operazione sta richiedendo? E come fa il client a mandare al server il numero giusto di argomenti? Ho risposto a queste domande immaginando la comunicazione tra i due come una macchina a stati finiti.

```
public static final int OP_LOGIN = 0;
public static final int OP_LOGOUT = 1;
public static final int OP_LISTPROJECTS = 2;
public static final int OP_CREATEPROJECT = 3;
public static final int OP_ADDMEMBER = 4;
public static final int OP_SHOWMEMBERS = 5;
public static final int OP_SHOWCARDS = 6;
public static final int OP_SHOWCARD = 7;
public static final int OP_ADDCARD = 8;
public static final int OP_MOVECARD = 9;
public static final int OP_GETCARDHISTORY = 10;
public static final int OP_CANCELPROJECT = 11;
```

Innanzitutto, ho deciso di associare un intero specifico ad ogni operazione che il client poteva eseguire sulla connessione TCP. 0 = login, 1 = logout, 2 = list projects, etc.. Per maggiore leggibilità, ho comunque definito nella classe Constants delle costanti rappresentative dell’operazione desiderata dal client.

Poi ho sfruttato una caratteristica comune a tutti i metodi: il fatto che siano composti da un numero prefissato e finito di argomenti, che sono tutte stringhe. Sono pertanto arrivato a costruire il seguente protocollo di comunicazione, modellato come una macchina a stati finiti:
il client invia per prima cosa l’intero identificativo dell’operazione, così che il server sappia come interpretare gli argomenti che riceverà;
successivamente, per ogni argomento dell’operazione richiesta, il client invia prima la lunghezza in byte dell’argomento, poi l’argomento vero e proprio (che è sempre una stringa);
infine, per segnalare al server che tutti i dati sono stati inviati, invia l’intero speciale -1. Il server, quando lo riceve, capisce che ha finito di leggere i dati del client, e può lanciare il thread che si occuperà di gestire la sua richiesta.
Qui di seguito una rappresentazione grafica di questo automa lato server (ovvero di come si comporta il server quando il client scrive qualcosa sul channel).



5.3 - Gestione delle singole richieste mediante ThreadPool e la classe ServerRequestHandler

Una volta che il server ha identificato l'operazione richiesta dal client e ha ricevuto da questo gli argomenti, va a lanciare un thread che gestisca la sua richiesta. Per fare ciò non lancia necessariamente un nuovo thread ogni volta, ma utilizza un ThreadPool per evitare che ci sia un sovra-utilizzo di risorse computazionali.
Ci sono vari motivi per cui ho deciso di gestire le richieste dei client usando un thread istanziato da una ThreadPool:
1) Se stessimo parlando di un'applicazione client-server realmente usata, sarebbe più che plausibile immaginare che il server abbia delle capacità computazionali tali da permettergli di lanciare più thread in contemporanea. Comunque sia, per evitare rallentamenti e utilizzi troppo dispendiosi delle risorse del server, ho optato per una ThreadPool di massimo 10 thread per la gestione delle richieste. Questo ovviamente introduce maggiore concorrenza sul server, ed è anche per questo motivo infatti che mi è stato necessario uno studio dei monitor, illustrati nel capitolo precedente.
2) Spesso le operazioni dei client richiedono letture o scritture da/verso disco, che se bloccanti possono richiedere tempo. Avrei potuto optare anche per una gestione single-threaded delle richieste ammettendo che le letture e scritture su disco non fossero bloccanti. Ho deciso tuttavia di scartare questo approccio in quanto ciò avrebbe comportato un abbassamento del throughput dal punto di vista dei client. Magari un client ha solo bisogno di leggere una card, mentre un altro deve cancellare un progetto molto grande. Con gestione a thread singolo di queste richieste, e scrittura/lettura su disco non bloccante, la prima richiesta, che in teoria sarebbe veloce da eseguire, potrebbe rallentare non poco.
Anche l'idea di gestione a thread singolo con letture e scritture bloccanti non mi allettava, perché di nuovo, se il server riceve la richiesta di lettura di una card da un client e di cancellazione di un progetto enorme da un altro nello stesso istante, se quest'ultimo ottiene la priorità di esecuzione il primo client avrà un tempo di risposta molto dilatato rispetto a quello che si sarebbe potuto aspettare.
Con il ThreadPool, invece, il server è in grado di gestire più richieste in contemporanea, facendo in modo che ogni client "subisca" solo le letture/scritture su disco relative alla sua operazione richiesta. Ciò aumenta il parallelismo soprattutto quando vengono effettuate operazioni su progetti differenti.

I thread istanziati dal ThreadPool sono oggetti runnable della classe ServerRequestHandler, che fanno una cosa molto semplice: prendono l'attachment della key relativa al client in attesa, controllano che operazione il client ha richiesto, la eseguono usando il metodo appropriato della classe ServerTcpOperations, prendono il risultato dell'operazione, lo salvano sull'attachment e impostano la chiave come interessata ad operazioni di scrittura. Nella fase di testing mi sono reso conto che non sempre (anzi, quasi mai), quando questo thread ha terminato la sua esecuzione e impostato l'interest ops della key all'operazione di scrittura, la select si sblocca. Per forzarla, pertanto, viene chiamato dal thread il metodo wakeup() sulla select del ServerWorthMultiplexerNio.

5.4 - La classe ServerWorthObjectAttached

Un oggetto di questa classe viene istanziato tutte le volte che viene registrata una nuova key sul selector. Questa classe contiene tutte le variabili necessarie a ricordare gli argomenti che il client ha inviato al server sul channel, come l'username, la password, il nome del progetto, il nome del nuovo membro, etc..
Contiene inoltre delle variabili "di stato" che gli permettono di leggere correttamente i dati inviati:
id_operation: variabile usata per ricordarsi quale operazione il client ha richiesto al server di eseguire, e dare anche quindi un significato agli argomenti che il client sta per inviare.
bytes_to_read: variabile per ricordarsi quanto sarà lungo, in byte, il prossimo argomento che il client sta per inviare.
reading_state: variabile ove si conserva il valore della macchina a stati finiti usata per leggere i dati del client nella comunicazione client server (vedere immagine del paragrafo 5.2). In particolare gli stati sono:
Constants.ATT_STATE_OPERATION_IDENTIFICATION: quando il client sta comunicando al server l'intero rappresentativo dell'operazione a cui è interessato;
Constants.ATT_STATE_DATA_LENGTH: quando il client sta comunicando al server la lunghezza del prossimo argomento;
Constants.ATT_STATE_DATA_READING: quando il client sta comunicando al server il prossimo argomento;
Constants.ATT_STATE_ALL_DATA_READ: quando il client manda l'intero speciale -1 al server, per indicare che non ha altri dati da inviare.
counter_infos_read: variabile che indica, una volta specificata l'operazione, quale argomento si sta andando a leggere.
failed_to_read_counter: variabile per gestire il crash del client.
ResInt: variabile in cui viene conservato il valore di ritorno intero dell'operazione eseguita sul server.
ResString: come quella sopra, con la differenza che conserva un valore di ritorno String piuttosto che Int.
StateResString: variabile di supporto per resString. La stringa restituita sarà infatti spesso una lista di oggetti serializzati, del tipo [intero, oggetto serializzato, intero, oggetto serializzato, …], dove l'intero rappresenta la lunghezza in byte del prossimo oggetto serializzato.

L'oggetto viene dunque modificato (le sue variabili) nel corso della comunicazione dal client al server. In particolare:
il server, nella classe ServerWorthMultiplexerNio, identifica l'operazione richiesta dal client, e la salva in id_operation.
Dopo di che si cominciano a leggere i dati inviati dal client, secondo il formato [lunghezza, argomento].
Ogni volta che si riceve un intero, si salva il suo valore in bytes_to_read, così da sapere di che dimensione deve essere il prossimo buffer.
Ogni volta che invece si riceve un argomento e lo si legge, ServerWorthMultiplexerNio invoca il metodo setNewDataRead dell'oggetto ServerWorthObjectAttached, che in base all'operazione

specificata e al valore di counter_infos_ready, va ad interpretare in un modo o nell'altro il valore appena letto, salvandolo nella variabile appropriata.
Finiti di leggere tutti i dati dal client (si è ricevuto -1), si lancia un ServerRequestHandler, che userà i valori dell'attachment (cioè gli argomenti delle operazioni) e i metodi della classe ServerTcpOperations per soddisfare la richiesta del client.

5.5 - Gestione dei crash client: chiusura "d'emergenza" della connessione TCP

Normalmente un client dovrebbe disconnettersi da Worth facendo il logout. Tuttavia ciò non sempre potrebbe accade, in quanto potrebbe crashare (nel nostro caso è molto facile che un utente "crashi": basta che prema la X rossa in alto a destra nell'interfaccia grafica). Se questo accade, viene lanciata, nella classe ServerWorthMultiplexerNio, una IOException, che simula il logout del client. Infatti viene chiamato il metodo updateCrash dell'oggetto CallbackImplementation (di cui parleremo in un prossimo capitolo), che va ad aggiornare tutti gli altri utenti online del fatto che ora l'utente crashato è offline (settando inoltre lo stato di questo utente come offline), e indicando, per ogni progetto di cui l'utente fa parte, che ora c'è un utente in meno che sta usando tale progetto, chiamando il metodo decrementNumberOfUsersUsingThisProject del WorthVolatileProjectInformations associato a quel progetto (Vedi capitolo 2.3). Il server procede quindi a cancellare la chiave associata a quel client e a chiudere il channel.
Eseguendo su linux mi sono reso conto che questa eccezione non sempre veniva lanciata, in quanto il server continuava a tentare di leggere dal client dati che non arrivavano mai, non cambiando la position del buffer, che rimaneva sempre a 0, e mantenendo il limit sulla capacity. Per gestire questo strano fenomeno ho aggiunto un controllo extra lato server: se per dieci volte che si è tentato di leggere dal client si ha avuto come esito quello di non aver letto neanche un byte (position == 0), viene lanciata una IOException, che viene subito catturata e gestita come sopra (è possibile ricordarsi il numero di letture fallite mediante la variabile failed_to_read_counter della classe ServerWorthObjectAttached).

Capitolo 6: RMI e callback

Prima di giungere a questo argomento avrei preferito parlare della classe ServerTcpOperations, così da trattare prima interamente il server e poi spostarmi lentamente verso il client. Tuttavia, se così facessi, sarebbe molto complicato trattare come si deve i metodi di login e logout, la cui "ciccia" sta proprio nel meccanismo delle callback. Ciò che farò ora pertanto sarà introdurre i meccanismi di RMI che ho utilizzato nel progetto, partendo dalla registrazione, e concludendo il capitolo con un'anticipazione dei metodi TCP parlando del login e del logout. Il resto dei metodi verrà approfondito ampiamente nel prossimo capitolo.
Sarà spesso citato inoltre il "thread del client che si occupa della chat di un progetto". Ne parleremo meglio in un prossimo capitolo dedicato ai client, ma per ora ci basti sapere, ad alto livello, che si tratta di un thread (lato client) il cui unico scopo è quello di ricevere e salvare i messaggi inviati sulla chat di un progetto.

6.1 – La registrazione

Il meccanismo di registrazione avviene, come da specifica, usando RMI. In particolare, lato server troviamo l'interfaccia **RegistrationInterface** e la sua implementazione **RegistrationImplementation**. C'è solo un metodo: **registerToWorth**, che prende come parametri l'username e la password. Il metodo, sincronizzato sul monitor ALL_USERS_MONITOR, fa una

serie di controlli prima di registrare l'utente. In particolare restituisce un codice di errore diverso quando si verifica una delle seguenti situazioni:
-La password è vuota;
-L'username è vuoto;
-L'username è lungo più di 20 caratteri;
-Esiste già un utente con quel nome;
I primi tre controlli sono immediati da fare, per il quarto invece bisogna eseguire qualche passaggio in più. Viene chiamato infatti il metodo ServerInfo.addUser, che tenta di aggiungere l'utente a Worth. Se in utentiRegistratiWorth esiste già una chiave con valore username la registrazione fallisce, altrimenti viene creato un nuovo oggetto UtenteRegistratoConPassword, che viene poi serializzato e inserito nella cartella degli utenti. Viene quindi aggiunta una nuova entry alla struttura dati utentiRegistratiWorth, e al client viene confermata l'avvenuta registrazione. Allo stesso tempo, viene fatta la callback a tutti gli utenti già online su Worth per avvisarli che c'è un nuovo utente registrato a Worth (offline, diventerà online non appena si loggerà).
Questo oggetto viene creato ed esportato sul registry al momento dell'avvio del main del server.

6.2 - Callback: register e unregister

Per quanto riguarda invece il meccanismo di RMI callback, ho creato un oggetto remoto sul server e uno sul client. Per il server abbiamo **CallbackInterface** e **CallbackImplementation**, mentre sul client abbiamo **UsersInformationsInterface** e **UsersInformationsImplementation**.
CallbackInterface: questa interfaccia comprende i due metodi remoti che il client può invocare per registrarsi alle callback (e contemporaneamente ottenere la lista degli utenti registrati a Worth) e cancellarsi dalle callback. In CallbackImplementation abbiamo non solo l'implementazione di questi metodi, ma anche altri metodi che andiamo ora a descrivere che permettono di fare le callback.

6.3 – Oggetto remoto del client

Per utilizzare il meccanismo di RMI callback, il client deve passare un riferimento al suo oggetto remoto al server in fase di registrazione alle callback. Parliamo in particolare dell'interfaccia **UsersInformationsInterface**, implementata dalla classe **UsersInformationsImplementation**.
Quando un client si registra al meccanismo di callback RMI passa un riferimento a questo oggetto remoto al server, che può utilizzarlo per invocare due metodi sul client:
updateUserState: metodo che prende come argomento un oggetto serializzato UtenteRegistrato e lo aggiunge alla struttura dati locale al client che contiene gli oggetti di questo tipo, uno per ogni utente registrato a Worth. Accedendo a questa struttura il client potrà sapere quali utenti sono online e quali offline.
UpdateUserChat: metodo con cui il server può comunicare al client un indirizzo di multicast associato ad un progetto di cui l'utente fa parte. In particolare il server può dire al client di attivare un thread che si occupi della chat di quel progetto, ma può anche dirgli di interrompere quel thread quando il progetto viene cancellato.
In fase di login, il client, registrandosi alle callback, va a creare un oggetto UsersInformationsImplementation e ad esportarne lo stub sul server, che quest'ultimo utilizzerà per comunicare al client aggiornamenti futuri.

6.4 – Vari metodi per effettuare le callback

Le callback devono servire ad avvisare gli utenti loggati su Worth quando un nuovo utente si registra, va online o va offline. Non solo, servono anche per comunicare agli utenti gli indirizzi di

multicast su cui aspettare di ricevere messaggi (le chat dei progetti di cui gli utenti fanno parte). Ci sono quindi due motivi per cui fare le callback agli utenti loggati:
1) avvisarli del fatto che un utente è passato dall'essere offline all'essere online o viceversa.
2) comunicargli informazioni sulla natura dei progetti di cui fanno parte.
Parleremo del primo scenario nel prossimo paragrafo in merito al login e al logout, mentre parleremo del secondo quando tratteremo delle operazioni TCP.
La struttura dati che mantiene i riferimenti agli oggetti remoti dei client è una hashmap <nome utente, UsersInformationsInterface> chiamata **registeredClientsForCallback**, che si espande quando un utente si logga e si contrae quando qualcuno effettua un logout.

Vediamo quindi adesso i metodi che permettono di effettuare le callback presenti nell'oggetto CallbackImplementation.
RegisterForCallback: metodo esportato chiamato dal client quando si logga a Worth. In questo metodo, come prima cosa, si va ad aggiornare lo stato dell'utente loggato (da offline a online), per poi fare le callback a tutti gli altri client che si erano registrati al meccanismo di callback, chiamando il metodo updateAllUsers. L'utente viene quindi registrato anch'esso al meccanismo di callback (si inserisce la sua stub in registeredClientsForCallback), e per ogni progetto di cui fa parte viene chiamato il metodo updateSomeUsersChat, per avvisarlo che deve creare un thread per ogni progetto di cui fa parte, così da riceve i messaggi delle relative chat. Infine, il metodo restituisce all'utente la lista degli utenti registrati a Worth, con relativo stato (online/offline).
UnregisterForCallback: metodo chiamato da un client in fase di logout per cancellarsi dal meccanismo di callback. Come prima cosa si vanno a controllare i progetti di cui l'utente fa parte, e per ciascuno di essi si chiama il metodo updateSomeUsersChat per dire al client di interrompere tutti i thread che sono in attesa di messaggi sulle chat. Questa comunicazione è necessaria in quanto un utente potrebbe disconnettersi da un computer, ma poi sullo stesso client un altro utente potrebbe effettuare il login, aggiungendo nuovi thread a quelli precedenti. Dopo di che la stub del client viene rimossa da registeredClientsForCallback, si imposta lo stato dell'utente a offline e si chiama il metodo updateAllUsers per avvisare gli altri utenti online del fatto che un utente si è disconnesso.
UpdateAllUsers: usato per avvisare tutti i client registrati alle callback del cambiamento di stato di un utente.
UpdateSomeUsersChat: metodo che prende quattro argomenti: un arraylist di stringhe, contenente i nomi degli utenti da avvisare, il nome di un progetto, un indirizzi di multicast e un booleano che rappresenta l'operazione da compiere. In base al valore di quest'ultimo argomento il messaggio comunicato al client cambia molto:
Constants.CHAT_ADD (true): si va a dire ad ogni utente il cui nome è presente nell'arraylist di creare un thread che si occupi di ricevere sull'indirizzo di multicast specificato i messaggi del progetto project_name. In questi casi l'arraylist contiene un solo elemento.
Constants.CHAT_REMOVE (false): si va a dire ad ogni utente il cui nome è presente nell'arraylist di interrompere il thread che si sta occupando di ricevere messaggi relativi al progetto project_name.

Abbiamo già visto che questo metodo viene invocato quanto un client chiama registerForCallback e unregisterForCallback, ma ci sono altri tre metodi che il server può invocare e che contengono al loro interno una chiamata a updateSomeUsersChat:
updateChatCreateProject: quando un progetto viene creato bisogna dire subito al suo creatore di lanciare un thread che si occupi di ricevere messaggi sulla chat del progetto.
UpdateChatAddMember: se Topolino aggiunge Paperino ad un progetto, e Paperino è online, bisogna dirgli di lanciare un thread che si occupi di ricevere i messaggi relativi alla chat di quel progetto.
UpdateChatCancelProject: se un progetto viene cancellato, bisogna avvisare tutti i suoi membri di interrompere l'esecuzione del thread relativo alla chat.

UpdateCrash: è come il metodo unregisterForCallback, con la differenza che, dato che il client è crashato, non c'è bisogno di dirgli di fermare i thread relativi alle chat.

6.5 – Login e logout

Avendo visto quali sono e dove vengono usati gli oggetti e i metodi remoti, andiamo adesso a vedere due metodi che ne fanno un pesante uso: il login e il logout. Per poter vedere però dove di preciso entra in gioco l'RMI callback, dobbiamo parlare sia del client che del server.

Operazione di login:

CLIENT

L'utente, dopo aver inserito nell'interfaccia grafica il nome utente e la password, preme il pulsante di login, che chiama il metodo **ClientTcpOperations.clientLogin(username, password)**. Il client apre quindi la connessione con il server, inviandogli il nome utente e la password.

SERVER

Il server accetta la connessione col client e chiama il metodo **ServerTcpOperations.serverLogin(username, password)**. Se le credenziali non vengono riconosciute, al client viene inviato un codice di errore e la connessione interrotta. Altrimenti la connessione rimane e viene inviato al client un codice di conferma.

CLIENT

Avendo effettuato correttamente il login, l'esecuzione del metodo clientLogin continua, e viene invocato il metodo registerForCallback dell'oggetto remoto del server.

SERVER

l'invocazione di registerForCallback fa sì che lo stato dell'utente venga settato a online, e a tutti gli altri utenti online venga notificato il cambiamento di stato dell'utente, mediante il metodo updateAllUsers. Lo stub del client viene quindi aggiunto alla struttura dati registeredClientsForCallback, e al client viene comunicato, mediante il metodo updateSomeUsersChat, di lanciare un thread per ogni progetto di cui l'utente fa parte. Qui chiaramente il server comunica al client gli indirizzi di multicast relativi ai progetti. Infine viene restituita la lista di utenti registrati a Worth, con il relativo stato.

CLIENT

la lista viene letta (deserializzata) e conservata nella struttura dati ClientInfo.worthUsersList, da cui l'utente attingerà quando vorrà conoscere lo stato degli utenti di Worth. L'utente è finalmente loggato.

Operazione di logout:

CLIENT

L'utente, sulla pagina iniziale di Worth, preme il pulsante di logout, che invoca il metodo **ClientTcpOperations.clientLogout(username)**, il quale invia al server l'username.

SERVER

Il server riconosce la richiesta di logout del client e invoca **ServerTcpOperations.serverLogout(username)**, mandando un codice di conferma al client. Chiude quindi la connessione col client e cancella la sua chiave.

CLIENT

viene invocato il metodo unregisterForCallback sull'oggetto remoto del server.

SERVER

L'invocazione di questo metodo produce i seguenti effetti:

-per ogni progetto di cui l'utente fa parte viene fatta una callback sulla stub UsersInformationsInterface del client, invocando il metodo updateSomeUsersChat per dire al client di fermare l'esecuzione dei thread che si stanno occupando delle chat.

-La stub del client viene rimossa dalla struttura dati registeredClientsForCallback.

-Lo stato dell'utente viene impostato a offline.

-Vengono fatte le callback a tutti gli altri utenti connessi a Worth mediante il metodo updateAllUsers per avvisarli del cambiamento di stato di questo utente.

CLIENT
Il client chiude la connessione col server.

Capitolo 7: Le operazioni client-server – ClientTcpOperations e ServerTcpOperations

Parliamo ora delle operazioni che il client può effettuare sulla connessione Tcp sul server. Questo capitolo va a trattare la maggior parte dei metodi richiesti nel testo del progetto, con alcune eccezioni:
-il login e il logout sono stati trattati nel capitolo precedente, nell'ultimo paragrafo.
-listUsers e listOnlineUsers si basano sul meccanismo di callback e sono operazioni effettuate in locale dal client. Verranno trattate nel prossimo capitolo, incentrato esclusivamente sul client.
-listProjects(). Diversamente da quanto specificato nel testo del progetto, ho deciso di implementare questo metodo senza richiedere il coinvolgimento della connessione Tcp. Verrà descritto il suo funzionamento nel capitolo 9 dopo aver spiegato il funzionamento delle callback lato client.

Per quanto riguarda invece i contenuti di questo capitolo:

7.1 – createProject

CLIENT
L'utente, quando si trova sulla pagina iniziale di Worth, può creare un nuovo progetto specificando il nome nell'apposita area di testo e premendo poi il pulsante. Invoca così facendo il metodo **ClientTcpOperations.clientCreateProject(project_name)**, il quale invia al server mediante la connessione TCP il nome del nuovo progetto.
SERVER
Il server gestisce la richiesta del client invocando il metodo **ServerTcpOperations.serverCreateProject(project_name, creator)**. Il secondo argomento (l'username dell'utente che sta creando il progetto) viene reperito dall'attachment associato alla key di quel client, che ha inizializzato il campo username quando l'utente ha effettuato il login (il server insomma, quando riceve una richiesta da un utente loggato, è sempre in grado di sapere chi ha effettuato tale richiesta).
Il metodo serverCreateProject, sincronizzato su ALL_PROJECTS_MONITOR, controlla innanzitutto se esiste già un progetto con quel nome. Se così fosse, restituisce un codice di errore al client. Altrimenti crea una nuova directory dedicata a quel progetto, poi un WorthProjectManager che viene serializzato e inserito nella cartella. Viene inoltre creata una nuova entry <nome progetto, WorthVolatileProjectInformations> inserita in ServerInfo.progettiWorth. Il server inoltre si segna subito che il numero di utenti che stanno usando quel progetto è aumentato di uno (il progetto è appena stato creato da un utente), assegnandogli un indirizzo di multicast e comunicandolo al client creatore mediante RMI callback. Viene poi aggiornata la lista di progetti di cui il client fa parte, andando a modificare la entry <nome utente, UtenteRegistratoConPassword> presente in ServerInfo.utentiRegistratiWorth. Per rendere consistenti le modifiche, l'oggetto UtenteRegistratoConPassword viene subito scritto nella cartella degli utenti, andando a sovrascrivere il file precedente con quel nome. Viene infine restituito un codice di conferma al client.
CLIENT
In base all'intero ricevuto, il client mostra un popup di feedback all'utente.

7.2 – addMember

CLIENT
Dopo aver acceduto ad un progetto, l'utente può aggiungere un nuovo partecipante a tale progetto usando il campo di testo e il pulsante in alto a destra nell'interfaccia grafica. Inserendo il nome dell'utente da aggiungere e premendo sul pulsante, viene effettuata una chiamata a **ClientTcpOperations.clientAddMember(project_name, new_member)**, che comunica al server il nome del progetto e il nome del membro da aggiungere.
SERVER
per gestire la richiesta, il server invoca il metodo **ServerTcpOperation.serverAddMember(project_name, new_member)**, sincronizzato sul WorthVolatileProjectInformations relativo al progetto project_name (sincronizzato quindi su ServerInfo.progettiWorth.get(project_name)).
Come già descritto, ogni metodo sincronizzato su questo monitor fa un primo controllo circa l'esistenza del progetto in questione. Se il controllo risulta negativo, l'operazione è interrotta e al client viene restituito un codice di errore, che lo riporterà alla pagina iniziale di Worth. Non menzioneremo più questo controllo nella descrizione dei prossimi metodi dando per scontato che l'utilizzo di una lock a grana fine implichi il controllo dell'esistenza del progetto.
Il server controlla quindi se l'utente da aggiungere esiste all'interno di Worth, e se esiste, controlla anche se fa già parte di questo progetto. Se necessario, manda quindi un codice di errore al client.
Se invece è possibile aggiungere l'utente al progetto, viene aggiornato l'oggetto UtenteRegistratoConPassword di quell'utente, indicando nella lista di progetti di cui fa parte il progetto project_name. L'oggetto viene quindi serializzato. Viene poi deserializzato, modificato, e riserializzato il WorthProjectManager del progetto in questione, aggiungendo alla sua struttura dati listMembersName il nome del nuovo partecipante.
Siccome l'utente aggiunto potrebbe essere online, si fa un controllo: se tale utente è davvero online, gli si fa una callback tramite updateChatAddMember per dirgli di lanciare un thread relativo alla chat del progetto a cui è stato appena aggiunto. Altrimenti non si fa nulla.
Viene infine inviato un messaggio automatico sulla chat del progetto per avvertire i partecipanti che un nuovo utente è stato aggiunto al progetto. Come ultima cosa, viene restituito un codice di conferma al client.
CLIENT
Il client riceve un codice dal server relativo all'esito dell'operazione, e in base a questo crea un popup per dare un feedback al client.

7.3 – showMembers

CLIENT
Il client, dopo che l'utente ha premuto sul pulsante "Mostra membri", invoca il metodo **ClientTcpOperations.clientShowMembers(project_name)**, che comunica al server il nome del progetto.
SERVER
Viene invocato il metodo **ServerTcpOperations.serverShowMembers(project_name)**, sincronizzato sul WorthVolatileProjectInformations relativo al progetto project_name. Viene dunque deserializzato il WorthProjectManager del progetto selezionato, recuperato l'hashset dei membri, e restituito al client sottoforma di stringa json.
CLIENT
Il valore di ritorno di del metodo showMembers lato client è proprio l'hashset dei membri, deserializzato a partire dalla stringa json restituita dal server. Questo hashset viene mostrato come lista all'utente, sull'interfaccia grafica.

7.4 – showCards

CLIENT
l'utente, cliccando sul pulsante "Mostra le card di questo progetto", invoca il metodo **ClientTcpOperations.clientShowCards(project_name)**, che invia al server il nome del progetto.
SERVER
Viene chiamato il metodo **ServerTcpOperations.serverShowCards(project_name)**, sincronizzato sul WorthServerVolatileInformations del progetto. In questo metodo il server va a deserializzare il WorthProjectManager del progetto selezionato, legge le quattro list di card e le concatena in un nuovo arraylist, che poi restituisce serializzato in formato json al client. Se non ci sono card da mostrare viene restituito un codice di errore.
CLIENT
il client riceve dal metodo ClientTcpOperations.clientShowCards(project_name) l'arraylist di card del progetto, che inserisce in una lista scorrevole e la mostra all'utente, aggiornando l'interfaccia grafica. Se non c'erano card nel progetto, viene notificato questo fatto all'utente con un popup.

7.5 – showCard

CLIENT
Quando l'utente sta visualizzando la lista di card di un progetto, può fare doppio click su una di esse per ottenerne le informazioni relative. Quando lo fa, viene invocato dal client il metodo **ClientTcpOperations.clientShowCard(project_name, card_name)**, che trasmette questi due argomenti al server.
SERVER
Viene invocato il metodo **ServerTcpOperations.serverShowCard(project_name, card_name)**, sincronizzato sul WorthVolatileProjectManager del progetto project_name, che va a deserializzare la WorthCard in formato json relativa alla card richiesta dal client. Legge quindi il nome, la descrizione e lo stato, che sono tutte stringhe, le inserisce in un arraylist, lo serializza e lo invia al client come stringa json.
CLIENT
Il client riceve questa stringa che viene deserializzata, e va a modificare l'interfaccia grafica in modo da mostrare all'utente le informazioni della card.

7.6 - addCard

CLIENT
L'utente, per aggiungere una nuova card al progetto, inserisce il nome e la descrizione degli appositi campi di testo sulla sinistra, e poi preme il pulsante "Crea una nuova card". Viene allora invocato il metodo **ClientTcpOperations.clientAddCard(project_name, card_name, description)**, che invia al server queste tre informazioni.
SERVER
Per creare la nuova card, il server invoca il metodo **ServerTcpOperations.serverAddCard(project_name, card_name, description)**, sincronizzato sul progetto project_name. Come prima cosa effettua una serie di controlli:
-Controlla se il nome della card coincide con quello del progetto;
-Dopo aver deserializzato il WorthProjectManager del progetto, controlla se esiste già, in una delle quattro liste di card, una card col nome specificato dal client;
-Controlla se il nome della card supera i 50 caratteri;
Se si presenta uno di questi casi, il server interrompe l'operazione e restituisce un codice di errore al client. Altrimenti aggiunge la nuova card al progetto, inserendola nella lista "To do", salva le modifiche effettuate serializzando l'oggetto WorthProjectManager, manda un messaggio sulla chat del progetto per avvertire gli utenti della creazione di questa card e restituisce un codice di successo al client.

CLIENT
Il client, a seconda del codice di risposta ricevuto dal server, fornisce all'utente un feedback mediante un popup.

7.7 - moveCard

CLIENT
Per muovere una card, l'utente inserisce il nome della card nel box di testo al centro dello schermo, e specifica la lista di partenza della card e quella di destinazione nelle due dropdown lists. Preme quindi il pulsante "muovi la card", che fa sì che il client invochi il metodo **ClientTcpOperations.clientMoveCard(project_name, card_name, from_list, to_list)**, che comunica al server questi quattro argomenti.
SERVER
Il metodo **ServerTcpOperations.serverMoveCard(project_name, card_name, from_list, to_list)**, sincronizzato sul progetto in questione, deserializza il WorthProjectManager del progetto ed effettua una serie di controlli:
-Verifica che in una delle quattro liste ci sia una card con quel nome
-Se c'è, controlla che la lista di partenza specificata dal client per quella card coincida con la lista in cui si trova al momento tale card.
-Se così è, verifica che lo spostamento sia "legittimo", ovvero che sia possibile spostare la card nella lista specificata dal client, rispettando il diagramma degli stati delle card.
Se uno di questi controlli fallisce viene restituito un codice di errore al client. Altrimenti la card viene rimossa dalla lista in cui si trova, aggiunta alla lista di destinazione e il WorthProjectManager serializzato per applicare le modifiche. In più viene deserializzato anche l'oggetto WorthCard della card spostata, così che se ne possa cambiare lo stato e aggiornare la history. Si salvano anche questi cambiamenti su disco mediante serializzazione e viene mandato un messaggio sulla chat del progetto per avvertire i membri dello spostamento. Si restituisce infine un codice di conferma al client.
CLIENT
In base al codice ricevuto dal server, il client mostra un popup di feedback all'utente.

7.8 – getCardHistory

CLIENT
L'utente, per ottenere la history di una card, inserisce il nome della card nel box di testo sulla destra, e clicca il pulsante "Visualizza history della card". Questo fa sì che venga invocato il metodo **ClientTcpOperations.clientGetCardHistory(project_name, card_name)**, che invia al server il nome del progetto e quello della card.
SERVER
Il server gestisce la richiesta di visualizzazione della history di una card chiamando il metodo **ServerTcpOperations.serverGetCardHistory(project_name, card_name)**, sincronizzato sul progetto project_name. Viene subito deserializzato il WorthProjectManager relativo, e si controlla se in almeno una delle liste è presente una card col nome card_name. Se non c'è, viene restituito un codice di errore al client. Altrimenti si va a prendere l'oggetto serializzato json di quella card, lo si deserializza, e si restituisce al client un arraylist serializzato che contiene la history della card con l'aggiunta dello stato attuale.
CLIENT
Il client legge la lista serializzata, e se non contiene come primo elemento un codice di errore, mostra la history della card all'utente sull'interfaccia grafica come lista scorrevole. Altrimenti comunica all'utente l'errore mediante un popup.

7.9 – cancelProject

CLIENT
L'utente che vuole cancellare il progetto che sta visualizzando sul client Worth clicca sul pulsante "(!) cancella questo progetto (!)", invocando così il metodo **ClientTcpOperations.clientCancelProject(project_name)**, inviando al server il nome del progetto che vuole cancellare.
SERVER
Il server gestisce questa delicata operazione mediante il metodo **ServerTcpOperations.serverCancelProject(project_name)**, che si sincronizza sulle lock ALL_PROJECTS_MONITOR, il WorthVolatileProjectInformations del progetto e ALL_USERS_MONITOR, per i motivi trattati nel capitolo 4. Questa operazione attraversa una serie di controlli e fasi, subito dopo aver deserializzato il WorthProjectManager del progetto:
1) Se c'è almeno una card che non si trova nello stato "Done", viene restituito un codice di errore al client.
2) Se invece tutte le card sono nello stato "Done", il server prende l'elenco dei membri del progetto, e:
2.1) Per ogni utente di questo elenco, viene rimossa dalla lista di progetti di cui l'utente fa parte il nome del progetto corrente, e le modifiche vengono applicate sul file system.
2.2) Da questa lista vengono filtrati gli utenti online
2.3) Dal risultato di questo filtraggio vengono fatte le callback per avvisare questi utenti che un progetto sta per essere cancellato, pertanto è necessario cancellare il thread che si occupa della sua chat. Ciò viene fatto invocando il metodo updateChatCancelProject dell'oggetto remoto CallbackImplementation.
3) Si recupera l'indirizzo di multicast associato al progetto (se è diverso da null), chiamando il metodo freeMulticastAddressBecauseProjectIsBeingCancelled sul WorthVolatileProjectInformations del progetto.
4) Viene cancellata la directory del progetto con tutti i suoi contenuti.
Infine, si restituisce un codice di conferma al client.
CLIENT
In base al codice ricevuto, il client crea un popup per avvisare l'utente che il progetto è stato o meno cancellato.

Capitolo 8 – Le classi lato client che mantengono informazioni e i metodi listUsers e listOnlineUsers

8.1 – La classe ClientInfo

Sebbene il client debba accedere al server per ottenere informazioni circa i suoi progetti e le sue card, ha bisogno di ricordarsi un paio di cose per far funzionare correttamente l'interfaccia grafica. Inoltre, visto che i metodi listUsers e listOnlineUsers si basano non sulla connessione TCP, bensì sul meccanismo di RMI callback, il client ha bisogno anche di una struttura dati che tenga traccia degli utenti registrati a Worth, aggiornata dal server al cambiamento di stato di uno di questi utenti (registrazione/login/logout), come specificato nel testo del progetto. Vediamo quindi le variabili della classe ClientInfo.
nome_utente_loggato: variabile che viene inizializzata al momento del login con il nome dell'utente loggato, e che viene settata a null nel momento in cui l'utente effettua il logout. Questa variabile serve al client per ricordarsi il nome dell'utente che lo sta usando, così da spedire tale nome al server in fase di logout.

nome_progetto_selezionato: variabile che viene inizializzata quando un utente seleziona un progetto dalla lista di progetti di cui fa parte, e che viene settata a null quando l'utente smette di agire su quel progetto. Questa variabile viene molto utilizzata quando si invocano metodi della classe ClientTcpOperations. Molte operazioni TCP richiedono infatti di comunicare al server il nome del progetto su cui si vuole agire, e grazie a questa variabile il client sa sempre su quale progetto l'utente vuole apportare delle modifiche, e può comunicarlo correttamente al server.
WorthUsersList: hashmap di coppie <nome utente, UtenteRegistrato> che permette al client di avere la lista degli utenti registrati a Worth con il loro stato (online/offline). Ho scelto una hashmap in modo da semplificare le operazioni di ricerca e modifica dello stato di un utente, sfruttando il fatto che su Worth non possono esistere due utenti diversi con lo stesso nome. Si vede qui inoltre perché, a suo tempo, abbiamo differenziato l'oggetto **utenteRegistrato** dall'oggetto **utenteRegistratoConPassword**. Un oggetto del primo tipo può essere comunicato senza remore al client dal server, in quanto contiene solo il nome di un utente e il suo stato (quindi contiene le informazioni PUBBLICHE). Il secondo invece è bene che possa accedervi solo il server, in quanto sì contiene un'istanza di utenteRegistrato per conoscere il nome e lo stato di ogni utente, ma contiene anche il seed e la password hashata per quell'utente (oltre alla lista di progetti di cui fa parte), tutte informazioni private che non devono essere diffuse (quindi informazioni PRIVATE).
Col metodo **updateUsersList** inoltre è possibile aggiungere una nuova coppia <nome, UtenteRegistrato> a questa struttura dati, oltre che aggiornarne una già esistente, ad esempio quando cambia lo stato di un utente registrato a Worth.

8.2 – La classe ClientTcpOperations

Abbiamo già parlato dei metodi di questa classe alla fine del capitolo 6 e nell'intero capitolo 7, esponendone la semplicità. Ciò che però non abbiamo citato è come questi metodi inviino i dati dal client al server, cosa che faremo in questo paragrafo.
Ogni metodo di questa classe atto ad inviare informazioni al server circa l'operazione da compiere e gli argomenti su cui agire si basa sulla chiamata di cinque fondamentali metodi, dal funzionamento prevedibile:
sendId(int identificatore_della_operazione): con questo metodo, che richiede come argomento l'intero che va ad identificare l'operazione richiesta (vedere la prima immagine del paragrafo 5.2), il client alloca un bytebuffer di 4 byte, vi inserisce l'intero e lo manda al server sul channel.
sendData(string argomento): con questo metodo il client invia al server la stringa "argomento", che rappresenta appunto uno degli argomenti dell'operazione richiesta. Viene presa la lunghezza in byte della stringa da mandare, si alloca un bytebuffer di 4 byte, ci si scrive dentro la lunghezza della stringa come un intero e si manda tale intero al server mediante il channel. Subito dopo si ri-alloca il bytebuffer con la dimensione della stringa in byte, ci si scrive dentro la stringa e la si manda al server. L'operazione viene ripetuta per ogni argomento che il client ha bisogno di mandare al server.
SendDataEnd(): metodo chiamato dopo aver mandato al server, con i due metodi precedenti, tutte le informazioni necessarie al corretto svolgimento di una operazione. SendDataEnd consente infatti di inviare l'intero -1 sul channel in maniera del tutto analoga al metodo sendId, così da segnalare al server che sono state mandate tutte le informazioni necessarie allo svolgimento di una operazione. La combinazione di questi tre metodi è l'implementazione lato client del diagramma FSM mostrato nel paragrafo 5.2.
receiveInt(): metodo usato dal client per ricevere sul channel l'intero di risposta di una operazione. Si alloca un buffer di 4 byte, si invoca la read, si legge il contenuto del buffer e si restituisce il valore letto, che rappresenta l'esito dell'operazione.
ReceiveString(): non tutte le operazioni effettuate sulla connessione TCP prevedono come risposta un intero, alcune restituiscono una stringa, solitamente una lista di elementi serializzata secondo il formato json (come nei metodi showCards o getCardHistory). In questi casi si alloca prima un buffer di 4 byte per ricevere la lunghezza della stringa che sta per essere inviata, poi lo si ri-alloca della dimensione giusta così da leggere correttamente la stringa inviata dal server.

8.3 - Come l’oggetto remoto del client modifica le sue informazioni locali: listUsers e listOnlineUsers

Abbiamo visto, quando abbiamo parlato di RMI callback, che il client, una volta effettuato il login o il logout chiama rispettivamente i metodi registerForCallback e unregisterForCallback dell’oggetto remoto del server, così da venire aggiornato in continuazione da quest’ultimo circa i cambiamenti di stato degli utenti. Il server infatti può chiamare i due metodi presenti nell’oggetto remoto usersInformationsImplementation: **updateUserState** e **updateUserChat.** Parleremo ora del primo, trattando il secondo nel capitolo successivo, quando spiegheremo come il client gestisce le chat dei progetti.
Il server, quando un utente si registra a Worth, va online o va offline, notifica tutti gli utenti online di questo cambiamento facendo le callback a tutti gli oggetti usersInformationsImplementation registratisi al meccanismo di callback RMI durante la fase di login. In particolare, l’invocazione di questo metodo prevede che la stringa inviata dal server (si parla di stringa in quanto il meccanismo di callback prevede tipi di dato primitivi come valori di ritorno dei metodi remoti) venga deserializzata usando Gson, ottenendo così un oggetto UtenteRegistrato relativo all’utente che si è registrato/ha fatto il login/logout. Viene quindi recuperato il nome dell’utente dall’oggetto deserializzato e aggiunta/aggiornata la coppia <nome, UtenteRegistrato> alla struttura dati ClientInfo.worthUsersList, mediante il metodo ClientInfo.updateUsersList.
A questo punto, un utente loggato su Worth che sta visualizzando la pagina iniziale dell’interfaccia grafica (quella con cinque pulsanti) può premere su “Mostra utenti” o “Mostra utenti online” per visualizzare una lista scorrevole degli utenti registrati o loggati a Worth. Tale lista viene ottenuta scorrendo la struttura dati ClientInfo.worthUsersList, eventualmente filtrando i soli utenti online (Il codice si trova nella classe WorthPanelShowAllUsers e WorthPanelShowOnlineUsers). Se l’utente vuole avere una versione più aggiornata degli utenti registrati o online, può premere il pulsante refresh oppure uscire e rientrare da quella schermata dell’interfaccia.

Capitolo 9

9.1 – La classe ClientChatManager

Per la gestione del meccanismo di chat multicast lato client ho deciso di seguire un approccio basato sulle RMI callback. Ogni volta che un utente si logga, crea un progetto o viene aggiunto ad un progetto esistente riceve una callback dal server che gli impone di avviare uno o più thread in attesa di messaggi su uno specifico indirizzo multicast, associato ad un progetto. Per comprendere al meglio questo meccanismo, vediamo prima le classi coinvolte, partendo da **ClientChatManager**.
Gli oggetti di questa classe, che implementa l’interfaccia Runnable, si occuperanno di ricevere i messaggi inviati sulla chat di multicast di un progetto. Al suo interno troviamo le seguenti variabili:
list_of_messages: arraylist dove memorizzare i vari messaggi inviati sulla chat. Ho scelto di usare un arraylist in quanto sono interessato a mantenere un ordine per i messaggi, ovvero quello temporale.
project_name: variabile che permette di capire a quale progetto è associato questo thread.
multicast_address: l’indirizzo di multicast su cui il thread deve ricevere i messaggi inviati, che sarà quindi relativo al progetto project_name.
Il metodo run di questo oggetto è così strutturato:
Se l’indirizzo di multicast che gli è stato passato è diverso da null (potrebbe avere questo valore quando il server esaurisce gli indirizzi di multicast) il thread termina subito. Questo comporta l’assenza di chat per il progetto project_name. Altrimenti il thread si unisce al gruppo di multicast associato al progetto, ed entra in un ciclo while che dura finché il thread non viene interrotto.

In questo ciclo, il thread si mette in attesa di messaggi sull'indirizzo di multicast indicato. Per evitare il troncamento dei messaggi ho imposto che la lunghezza massima di questi fosse di 512 byte, quindi 512 caratteri. Tuttavia gli utenti, mediante l'interfaccia grafica, saranno in grado di inviare messaggi di lunghezza massima di 490 caratteri. Questo perché all'inizio di ogni messaggio viene specificato il nome del mittente, che ha lunghezza massima 20 caratteri, seguito da due punti e uno spazio, per un totale massimo di 22 caratteri. Il resto del messaggio è scelto dall'utente.
Quando viene ricevuto un messaggio, si controlla innanzitutto se è vuoto (""). Un messaggio di questo tipo viene ricevuto solo quando il thread deve essere interrotto a seguito della cancellazione del progetto, o quando l'utente esegue il logout. Infatti, quando si verifica una di queste due operazioni, il CallbackImplementation del server invia una notifica mediante callback al client usando il metodo updateSomeUsersChat, che invoca il metodo UpdateUserChat dell'oggetto remoto del client (UsersInformationsImplementation) il cui effetto finale è quello di inviare un messaggio vuoto sulla chat. Ho scelto la stringa vuota come messaggio speciale in quanto né il server né alcun client può inviare questo messaggio sulla chat. Il server invia sempre messaggi con lunghezza ben maggiore di uno, e se un utente clicca sul pulsante "Invia" (nell'interfaccia della chat) quando il messaggio inserito è la stringa vuota, questo non viene inviato.
Quindi, se il messaggio ricevuto è la stringa vuota, il thread si interrompe invocando Thread.currentThread.interrupt(), e termina la sua esecuzione. Altrimenti aggiunge il messaggio all'arraylist list_of_messages. Per permettere la visualizzazione in tempo reale dei messaggi ricevuti, ho aggiunto un ulteriore controllo:
Se l'utente sta visualizzando un progetto e quel progetto corrisponde al valore della variabile project_name interna a questo thread, e la chat è aperta sull'interfaccia grafica (ClientChatInfo.chat_selected==true), allora aggiorna subito l'area di testo dell'interfaccia dove compaiono i messaggi.
Il thread, prima di terminare, lascia il gruppo e chiude la multicastSocket.

È inoltre presente un metodo, **getInitialMessages**, che permette di ottenere il contenuto della lista di messaggi ricevuti, usato dall'interfaccia grafica quando l'utente entra nella chat.

9.2 – La classe ClientChatInfo e il metodo listProjects

Il client, per far funzionare correttamente il meccanismo di callback, ha bisogno di avere sempre presente quali sono i thread-chat in esecuzione e le loro variabili. Per questo motivo esiste la classe **ClientChatInfo**, che contiene una struttura dati e una variabile.
MapChatManagers: hashmap <nome progetto, ClientChatManager> con cui il client associa il nome dei progetti di cui l'utente fa parte ai ClientChatManager relativi. Questa struttura è necessaria per mostrare all'utente la chat giusta quando, visualizzato un progetto, ne chiede l'accesso. Ho scelto una hashmap in modo da velocizzare l'accesso al giusto ClientChatManager, considerando il fatto che su Worth il nome di un progetto è univoco.
chat_selected: variabile booleana settata a true quando l'utente sta visualizzando la chat di un progetto, false altrimenti. Viene usata dal thread relativo alla chat di quel progetto per sapere se deve aggiornare in tempo reale i messaggi visualizzati a schermo o se gli basta aggiungere il messaggio alla lista di messaggi ricevuti.

9.3 - Aggiunta/rimozione di chat thread mediante RMI callback

Concludiamo infine l'argomento RMI callback tornando a parlare dell'oggetto UsersInformationsImplementation, che prevede, oltre al metodo UpdateUserState, anche il metodo **UpdateUserChat**. Esso viene invocato dal server in più occasioni:
-al login dell'utente;
-al logout;
-quando viene creato un progetto;

-quando l’utente viene aggiunto ad un progetto;
-quando un progetto viene cancellato;
Con questo metodo è possibile dire al client di eseguire un nuovo thread-chat o di interrompere l’esecuzione di uno già esistente. Sono previsti tre argomenti:
project_name: il nome del progetto a cui va associata/rimossa una chat;
ind_di_multicast: l’indirizzo di multicast da associare ad un progetto quando si va a creare il thread-chat relativo;
action: variabile booleana che indica al client come interpretare gli argomenti precedenti.
Il metodo è strutturato come segue:
Si analizza il valore di action. Se è uguale a Constants.CHAT_ADD (true), vuol dire che bisogna creare un nuovo thread-chat relativo al progetto project_name associandogli l’indirizzo di multicast ind_di_multicast. Il client procede dunque a creare un nuovo ClientChatManager, che riceverà messaggi sull’indirizzo di multicast ind_di_multicast, relativi al progetto project_name. Viene quindi aggiunta una nuova entry alla struttura dati ClientChatInfo.mapChatManagers, del tipo <project_name, ClientChatManager> e viene avviato il thread.
Se invece il valore di action è uguale a Constants.CHAT_REMOVE (false), questo metodo provvede a mandare un messaggio vuoto all’indirizzo di multicast associato al progetto project_name, così da interrompere l’esecuzione del thread. Viene infine rimossa la entry <project_name, ClientChatManager> dalla struttura dati ClientChatInfo.mapChatManagers.