

Corso di Laurea Triennale in Informatica



UNIVERSITÀ DI PISA

Simone Schiavone

Matricola 582418

Progetto Laboratorio di Reti
“WORTH: WORK TogetHer”

29 Gennaio 2021

1. Descrizione del problema:



WORTH è uno strumento nato per facilitare la gestione di progetti collaborativi che si ispira ad alcuni principi della metodologia *Kanban*. Tale metodologia rientra nelle tecniche di gestione “*agile*”; Si implementa per mezzo di una lavagna che fornisce una vista d’insieme delle attività di un progetto e permette di visualizzarne l’evoluzione, dalla definizione del task fino al suo completamento. Oltre ad offrire quindi un servizio di organizzazione di un progetto/lavoro tramite task, WORTH facilita la comunicazione tra i membri di uno stesso progetto fornendo un servizio di chat .

2. Architettura del sistema:

SERVER

Il sistema WORTH segue un’architettura di tipo client/server. Il server centralizzato mantiene al suo interno le informazioni relative ai progetti, cioè le card ed i membri, e la lista degli utenti registrati. La classe **Server** contiene i seguenti campi:

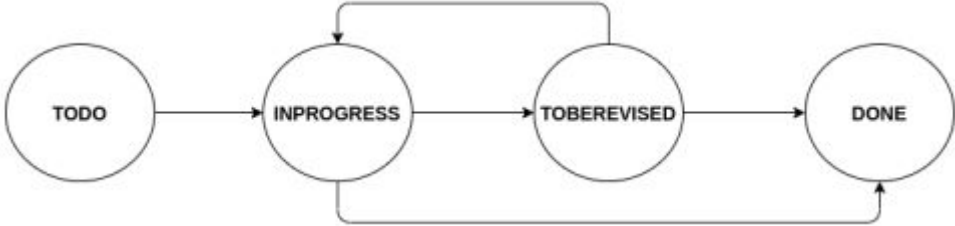
- ❑ un ArrayList “**listaprogetti**” di istanze della classe **Progetto**. Invocazioni corrette di **CreateProject** comporteranno l’inserimento di nuovi progetti in lista.
- ❑ Un ArrayList “**listautentiregistrati**” di istanze della classe **Utente**. Verranno aggiunti nuovi utenti quando il **Client** tramite RMI invocherà il metodo Register del **Server**.
- ❑ Una HashMap “**callbackinfo**” con chiave una istanza di una classe che implementa **ClientInterfacciaRMI** (cioè della classe client) e con valore una **Stringa** contenente il nome dell’utente associato a quel client. In questa HashMap saranno aggiunte entry quando il **Client** invocherà il metodo **Subscribe** per iscriversi al servizio di notifica di aggiornamenti, mentre saranno rimosse entry quando il **Client** invocherà il metodo **Unsubscribe** per disiscriversi dal servizio di notifica.
- ❑ Due oggetti della classe File che denotano la cartella di backup dei progetti e il file .json contenente le informazioni sugli utenti registrati.

SERVER
<ul style="list-style-type: none">- listaprogetti : ArrayList<Progetto>- listautentiregistrati : ArrayList<Utente>- callbackinfo : HashMap<ClientInterfacciaRMI, String>- useraddress : HashMap<SocketAddress, String>- fileutenti : File- filebackup : File- portaRMI : int- portaTCP : int- BufferDim : int
<ul style="list-style-type: none">+ [synch] Register(nickutente : String, password : String) : int- PubblicazioneStubRMI() : void- Login(username : String, password : String, SocketAddress addr) : String- Logout(username : String, SocketAddress addr) : String- Listusers() : String- Listonlineusers() : String- ListProjects(username : String) : String- CreateProject(nomeproj : String, username : String) : String- AddMember(projectname : String, utentedaaggiungere : String, richiedente : String) : String- ShowMembers(projectname : String, richiedente : String) : String- ShowCards(projectname : String, richiedente : String) : String- AddCard(comando : String[]) : String- ShowCard(projectname : String, cardname : String, richiedente : String) : String- MoveCard(projectname : String, richiedente : String, listapartenza : String, listadestinazione : String, richiedente : String) : String- ShowCards(projectname : String, richiedente : String) : String- GetCardHistory(projectname : String, cardname : String, richiedente : String) : String- CancelProject(projectname : String, richiedente : String) : String- InterpretaComando(comando : String[], SocketAddress addr) : String- StartServer() : void+ [synch] Subscribe(client : ClientInterfacciaRMI, username : String) : void+ [synch] Unsubscribe(client : ClientInterfacciaRMI, username : String) : void+ [synch] doCallbacks(client : ClientInterfacciaRMI, username : String) : void.....più altri metodi descritti successivamente....

- ❑ Delle costanti che denotano le porte usate per la connessione TCP, per RMI e la dimensione del buffer.
- ❑ Una HashMap **useraddress** con chiave un **SocketAddress** e con valore una stringa che denota un username. In questa struttura dati viene aggiunta dopo una operazione di login un'associazione tra la SocketAddress del client e username dell'utente autenticato su esso; in fase di logout e di chiusura forzata del client viene rimossa tale associazione. Ho ritenuto necessario aggiungere queste informazioni nel server per poter gestire in modo corretto le chiusure forzate dei client (es ctrl+c da terminale su ubuntu).

I metodi della classe **Server** riportati nella tabella descrittiva sono quelli strettamente legati alle operazioni offerte da WORTH; Tali comandi, dopo aver effettuato gli opportuni controlli sui parametri in input, agiscono quindi sulle collezioni di dati sopra descritte.

Login	Comando per effettuare l'operazione di login al servizio WORTH. Il nickname inserito deve essere già registrato al servizio e la password deve essere corretta. L'utente non deve essere già online da altri client. Se i controlli vanno a buon fine aggiorniamo l'oggetto della classe Utente associato all'username e notificiamo tutti i client che si sono iscritti al servizio di notifica (Callback). Aggiungiamo anche l'associazione tra il socketaddress da cui è arrivata la richiesta e username all'hashmap useraddress .
Logout	Comando per effettuare l'operazione di logout al servizio WORTH. Il nickname inserito deve essere già registrato al servizio. L'utente deve trovarsi nello stato "online" per poter effettuare il logout. Se i controlli vanno a buon fine aggiorniamo l'oggetto della classe Utente associato all'username e notificiamo tutti i client che si sono iscritti al servizio di notifica (Callback). Rimuoviamo l'associazione tra il socketaddress da cui è arrivata la richiesta e username dall'hashmap useraddress .
Listusers	Comando utilizzato da client per visualizzare la lista degli utenti registrati al servizio WORTH con il relativo stato. Al client viene restituita la rappresentazione JSON di una lista di stringhe composte dalla concatenazione dell'username e dello stato. In questo modo non si inviano al client informazioni sensibili sull'utente come la sua password (cosa che potrebbe accadere se si serializzasse direttamente l'oggetto della classe Utente associato a quell'username). Se sul client vi è un utente autenticato la richiesta "listusers" viene risolta localmente senza contattare il server ma consultando l'hashmap utenti che viene aggiornata dal server tramite callbacks. In caso contrario il client invia una richiesta al server che attiverà questo metodo.
Listonlineusers	Comando utilizzato dal client per visualizzare la lista degli utenti registrati attualmente online. Al client viene restituita la rappresentazione JSON di una lista di stringhe cioè la lista degli username degli utenti online. In questo modo non si inviano al client informazioni sensibili sull'utente come la password (cosa che potrebbe accadere se si serializzasse direttamente l'oggetto della classe Utente associato a quell'username). Se sul client vi è un utente autenticato la richiesta "listonlineusers" viene risolta localmente senza contattare il server ma consultando l'hash map utenti che viene aggiornata dal server tramite callbacks. In caso contrario il client invia una richiesta al server che attiverà questo metodo.
Listprojects	Comando utilizzato per recuperare la lista dei progetti di cui un utente è membro. Il metodo restituisce la rappresentazione JSON di una lista di stringhe ovvero i nomi dei progetti a cui partecipa l'utente che ha richiesto l'operazione (l'username è un parametro del metodo).
Createproject	Comando utilizzato per la creazione di un nuovo progetto. Il metodo controlla che non ci siano in WORTH progetti con lo stesso nome. Se il controllo ha esito positivo crea un nuovo progetto con il nome specificato (il nome è un parametro del metodo) ed

	<p>inserisce nei membri di tale progetto l'utente che ha invocato il metodo. Poichè ogni progetto ha associata una chat di gruppo, al momento della creazione del progetto il server provvede ad assegnare un indirizzo IP di multicast che sarà utilizzato dai client per lo scambio di messaggi. Sono invocati altri metodi di supporto per salvare sul disco le informazioni sul progetto (Cartella di backup del progetto, lista membri, ipmulticast).</p>
Addmember	<p>Comando utilizzato per aggiungere un membro ad un progetto. Il metodo verifica che l'utente richiedente sia un partecipante al progetto, che esista un progetto con il nome passato come parametro, che l'utente da aggiungere sia registrato a WORTH e che l'utente da aggiungere non partecipi già al progetto. Se i controlli vanno a buon fine il metodo aggiunge l'utente alla lista dei partecipanti al progetto e salva sul disco la nuova lista.</p>
ShowMembers	<p>Comando che restituisce la lista degli utenti membri di un progetto. Il metodo controlla che l'utente richiedente sia partecipante al progetto e che il progetto esista. Se i controlli hanno esito positivo, il metodo restituisce la rappresentazione JSON di una lista di nomi di progetto (stringhe).</p>
ShowCards	<p>Comando che restituisce la lista delle card del progetto specificato come parametro. Il metodo verifica che il progetto specificato esista in WORTH e che il richiedente dell'operazione ne faccia parte. In caso affermativo, il metodo restituisce la rappresentazione JSON di una lista di nomi di cards (stringhe).</p>
AddCard	<p>Comando per l'inserimento di una nuova card nel progetto specificato dall'utente. Il metodo prende in input l'intera stringa digitata dall'utente in quanto in tale stringa sono contenute le informazioni per creare una nuova carta cioè il nome e la descrizione. La descrizione può essere composta da una singola parola o da una sequenza di parole separate da uno spazio. Il metodo quindi tokenizza la stringa ricevuta dal client, con il metodo <i>split</i> della classe String, ed estrae il nome del richiedente (per verificare che il richiedente sia un partecipante al progetto), il nome del progetto (per verificare l'esistenza del progetto in WORTH) e il nome della card da creare (per verificare che non esista nel progetto una card con lo stesso nome). Se i controlli precedenti sono andati a buon fine il metodo provvede a salvare sul disco nella cartella corretta la nuova carta aggiunta.</p>
ShowCard	<p>Comando che mostra lo stato di una carta specifica di un progetto (nome carta e nome progetto sono parametri del metodo). Il metodo verifica che il progetto specificato esista in WORTH, che la carta esista nel progetto e che l'utente richiedente sia membro del progetto). La stringa restituita dal metodo segue questo pattern: "NomeCarta: [nome] Descrizione: [descr] Status: [lista_in_cui_si_trova_la_carta]"</p>
MoveCard	<p>Comando che permette lo spostamento di una certa carta di un progetto specificato da uno stato all'altro. Gli stati disponibili sono "todo", "inprogress", "toberevised", "done". Le transizioni di stato ammesse sono le seguenti:</p>  <pre> graph LR TODO((TODO)) --> INPROGRESS((INPROGRESS)) INPROGRESS --> TOBEREWISED((TOBEREWISED)) TOBEREWISED --> DONE((DONE)) INPROGRESS --> TOBEREWISED TOBEREWISED --> INPROGRESS </pre> <p>Il metodo controlla che il progetto specificato esista in WORTH, che l'utente richiedente sia un partecipante al progetto e che la carta da spostare sia presente nel progetto; successivamente richiama sulla carta target il metodo MoveCard (classe Card) che si occupa di verificare che la transizione richiesta rispetti i vincoli mostrati nella figura sopra riportata oltre che a spostare la carta nella lista di flusso corretta. Se tutti i</p>

CARD

La classe **Card** rappresenta un'astrazione del singolo task di un progetto. Le variabili contenute al suo interno sono autoesplicative; l'ArrayList di stringhe "**movimenticarta**" contiene la lista delle transizioni di stato che ha subito la carta descritte testualmente da una stringa. Ogni card che viene creata tramite il metodo costruttore viene inizialmente impostata nello stato "*todo*". Il nome della card deve essere composto da una sola parola (non sono ammessi spazi).

CARD
- nome : String - descrizione : String - stato : String - movimenticarta : ArrayList<String>
+ Card(n : String, descr : String) : Card + SetStato(s : String) + NuovoMovimento(descrizione : String) : String + GetName(descrizionemovimento : String) : void + GetDescrizione() : String + GetStatus() : String + GetMovimenti() : ArrayList<String>

PROGETTO

La classe **Progetto** contiene al suo interno più collezioni di dati, in particolare l'ArrayList **cards** contiene tutte le cards di un progetto mentre le liste **todo**, **inprogress**, **toberevised**, **done** contengono le stesse carte della lista globale ma suddivise per stato della carta. Ho scelto di mantenere una lista globale di cards per facilitare le operazioni di ricerca di una carta, che altrimenti avrebbero comportato lo scorrimento di tutte le liste di "flusso". In uno stesso progetto non possono esserci cards con lo stesso campo "nome" e non possono esserci utenti con lo stesso campo "username". Il nome del progetto deve essere composto da una sola parola (non sono ammessi spazi).

All'interno della classe **Progetto** è presente anche un campo **ipchat** che rappresenta l'indirizzo di multicast assegnato dal server in fase di creazione del progetto. La classe Progetto ha due metodi costruttori; quello con due parametri è utilizzato dal **Server** per creare un nuovo

Progetto e prevede che venga passato come parametro anche l'utente che sta creando il progetto in modo da inserirlo subito tra i partecipanti al progetto. La versione con un solo parametro è utilizzata dal **Server** al suo avvio, nella fase in cui egli va a ricostruire il suo stato consultando le cartelle di backup create in precedenza. Nel progetto non è possibile inserire card dal nome "ipchat" o "membri" per non sovrascrivere i file di backup relativi ai membri di un progetto o dell'ip della chat di progetto.

PROGETTO
- nome : String - cards : ArrayList<Card> - membri : ArrayList<Utente> - todo : ArrayList<Card> - inprogress : ArrayList<Card> - toberevised : ArrayList<Card> - done : ArrayList<Card> - ipchat : InetAddress
+ Progetto(n : String) : Progetto + Progetto(n : String, creatore : Utente) : Progetto + MoveCard(card : Card, start : String, end : String) : String + AssegnaIpChat(ip : InetAddress) : boolean + GetName() : String + GetMembri() : String + GetCards() : ArrayList<Card> + isMember(username : String) : boolean + existsCard(nome : String) : Card + GetTODOCards() : ArrayList<Card> <i>in modo analogo sono definiti gli altri metodi osservatori per le varie liste di cards...</i>

UTENTE

Le variabili ed i metodi della classe Utente sono di immediata comprensione. È opportuno segnalare che la classe Utente prevede anche l'hashing, con l'algoritmo

UTENTE
- nome : String - password : String - online : boolean
+ Utente(n : String, psw : String) : Utente + GetUsername() : String + VerificaPassword(pswimmessa : String) : boolean + isOnline() : boolean + hash(input : String) : String + SetOnline() : int + SetOffline() : int

SHA-256, delle password inserite dall'utente in modo da non salvare le password in chiaro nel file di ripristino. Il metodo **VerificaPassword** effettuerà quindi la cifratura della password immessa dall'utente nella fase di login e verificherà che questa corrisponda a quella associata all'utente in fase di registrazione.

CLIENT

La classe **Client** estende la classe **RemoteObject** ed implementa l'interfaccia **ClientInterfacciaRMI**.

```
public interface ClientInterfacciaRMI extends Remote{
    //Metodo per aggiornare l'hashmap del client che contiene le associazioni username-status (online/offline).
    //Se nell'hashmap non esiste ancora una entry con chiave "username" allora viene aggiunta con valore "status"
    //altrimenti si sostituisce il vecchio valore associato con il nuovo status;
    public void AggiornaUtente(String username,String status) throws RemoteException;

    //Metodo per far partire un thread della classe ChatReaderWriter che resta in ascolto su un certo
    //indirizzo ip di multicast. Questo thread sarà memorizzato in una struttura dati del client in modo da
    //poterlo reperire durante le operazioni di lettura chat, invio messaggio o interruzione.
    public void AvviaRicezione(InetAddress address,String nomeproj) throws RemoteException;

    //Metodo per interrompere un thread memorizzato in una struttura dati del client che e' in ascolto
    //sull'indirizzo di multicast "address".
    public boolean StopRicezione(InetAddress address) throws RemoteException;
}
```

La classe **Client** contiene al suo interno una HashMap **utenti**, con chiave il nome dell'utente e valore il suo stato ("Online/Offline") e una ArrayList **listascoltatori** di ChatReaderWriter. È presente anche una variabile **utenteassociato** che memorizza il nome dell'utente che è attualmente loggato sul client. Il **main** di questa classe effettua un ciclo infinito richiedendo ripetutamente all'utente di digitare sul terminale una stringa contenente un particolare comando (rispettando la sintassi delle operazioni offerte da WORTH, che possono essere visionate digitando il comando "help"). Se il comando inserito dall'utente è *register*, *readchat*, *sendchatmsg* o *help*, il client risolve autonomamente la richiesta, sfruttando i propri metodi, senza dover inviare il comando sulla connessione TCP stabilita con il **Server**. Se sul client vi è un utente autenticato i comandi *listusers* e *listonlineusers* sono risolti localmente consultando l'hashmap **utenti**. Alcuni comandi (*listprojects*, *createproject*, *addmember*, *showmembers*, *showcards*, *showcard*, *addcard*, *movecard*, *getcardhistory*, *cancelproject*, *sendchatmsg*, *readchat*) richiedono che l'utente sia autenticato; Il metodo *VerificaLogin* controllerà che l'utente che ha richiesto uno dei comandi con login abbia effettuato la procedura di autenticazione. In caso affermativo concatenerà alla stringa ricevuta in input il nome dell'utente associato al client in modo che quando la stringa complessiva sarà inviata al server, questo possa capire qual è l'utente che ha richiesto

CLIENT
<ul style="list-style-type: none"> - portaRMI : int - portUDP : int - portaTCP : int - BufferDIM : int - Terminatore : String - utenteassociato : String - utenti : HashMap<String, String> - listascoltatori : ArrayList<ChatReaderWriter>
<ul style="list-style-type: none"> + Client() : Client + sendmessage(comandoletto : String) : void + readchat() : boolean + register(input : String) : void + AggiornaUtente(username : String, status : String) : void + Subscribe() : void + Unsubscribe() : void + AvviaRicezione(address : InetAddress, nomeprogetto : String) : void + StopRicezione(address : InetAddress) : boolean + StopRicezione() : void + Help() : void + VerificaLogin(comandoletto : String, c : Client) : String

l'operazione. La parte terminale del ciclo di lettura delle richieste si occupa di interpretare la risposta ricevuta dal server; tale risposta può essere una stringa che contiene un feedback in linguaggio naturale (Es: "Login EFFETTUATO; Benvenuto/a") oppure può contenere la rappresentazione JSON di una collezione di oggetti (progetti, card, utenti...).

CHATREADERWRITER

La classe **ChatReaderWriter** estende la classe **Thread**. All'interno si trova una **ArrayList** di stringhe **codamessaggi** che contiene i vari messaggi ricevuti sull'indirizzo di multicast **indmulticast**. Il metodo **run()** prevede che fin quando il thread non viene interrotto (mediante l'invocazione del metodo **Interrupt()** della classe **Thread**), questo resta perennemente in ascolto in attesa di ricevere un datagramma sulla **socketmulticast**. Al momento della ricezione di un datagramma, il thread estrae il messaggio (stringa) in esso contenuto e lo appende alla **codamessaggi**. Il metodo **LeggiChat()** stampa sul terminale la lista dei messaggi ricevuti dal momento in cui il thread si è messo in ascolto su quell'indirizzo di multicast; dopo la stampa la lista viene svuotata. Il metodo **InviaMessaggio** permette di inviare un datagramma contenente una stringa all'indirizzo di multicast **indmulticast**.

CHATREADERWRITER
<ul style="list-style-type: none">- indmulticast : InetAddress- socketmulticast : MulticastSocket- nomeproj : String- port : int- codamessaggi : ArrayList<String>
<ul style="list-style-type: none">+ ChatReaderWriter(ia : InetAddress, p : int, name : String) : ChatReaderWriter+ GetAddress() : InetAddress+ GetProject() : String+ LeggiChat() : void+ run() : void+ InviaMessaggio(text : String) : void

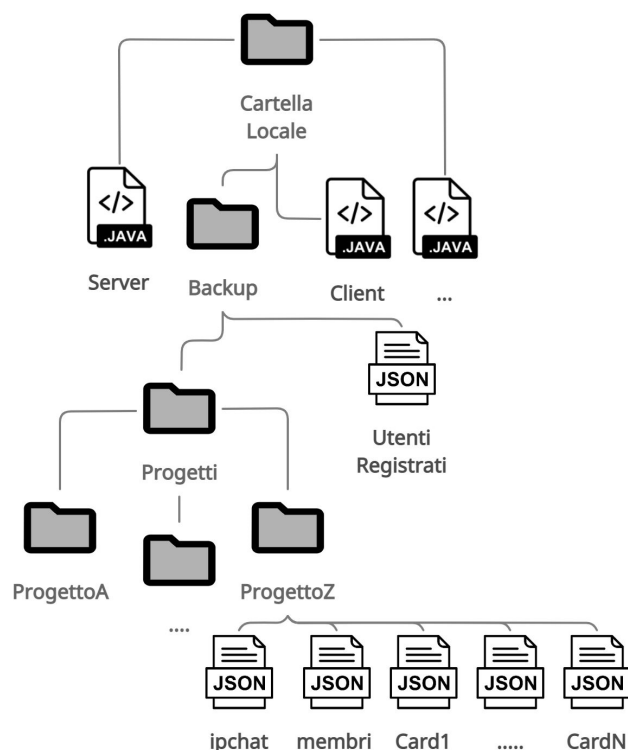
```
public void run() {
    try {
        while(!Thread.currentThread().isInterrupted()) {
            byte[] buffer=new byte[1024];
            DatagramPacket pacchetto= new DatagramPacket(buffer,buffer.length);
            try {
                socketmulticast.receive(pacchetto);
                if(Thread.currentThread().isInterrupted())
                    break;
            } catch (IOException e) {
                e.printStackTrace();
            }
            String messaggio=new String(pacchetto.getData(),0,pacchetto.getLength());
            codamessaggi.add(messaggio);
        }
        socketmulticast.leaveGroup(indmulticast);
    }catch(IOException e) {
        e.printStackTrace();
    }finally {
        socketmulticast.close();
    }
}
```

3. Persistenza

Per la persistenza ho utilizzato la libreria GSON versione 2.8.6. Il metodo costruttore della classe **Server** crea nella cartella del progetto WORTH una cartella denominata "Backup". Al

suo interno sarà creata una cartella “Progetti” dove saranno memorizzate le informazioni relative ai progetti creati dagli utenti. Il metodo

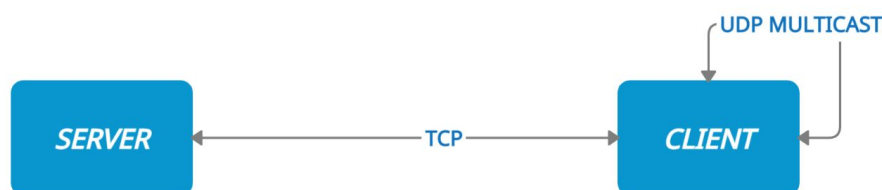
SalvataggioUtenti della classe **Server** crea un file nominato *utentiregistrati.json* contenente la rappresentazione JSON lista degli utenti registrati a WORTH; Ogni volta che viene registrato un nuovo utente alla piattaforma WORTH, il metodo sovrascrive il file precedente con la rappresentazione della nuova lista. Della classe **Utente** non persistiamo il valore del campo **online** (di tipo transient) per non avere inconsistenze in caso di arresto del server; infatti al momento della ricostruzione dello stato tutti gli utenti sono impostati offline tramite il metodo **Offline** della classe **Utente**. Alla creazione di un nuovo progetto il metodo **CreaCartellaBackupProgetto** crea dentro Backup/Progetti una nuova cartella che ha per nome il nome del progetto. All'interno della nuova cartella saranno creati due file: *ipchat.json* e *membri.json*. Il primo file memorizzerà l'indirizzo di multicast associato alla chat di quel progetto mentre il secondo conterrà la rappresentazione JSON della lista dei membri di quel progetto; queste informazioni saranno importanti per ricostruire lo stato del server in caso di riavvio. Nel momento in cui viene aggiunto un nuovo partecipante al progetto il metodo **SalvataggioMembriProgetto** aggiorna il file .json della lista di membri. All'interno di ogni cartella di progetto sono contenuti anche i file .json relativi alle varie cards. Alla creazione di ogni card ed in caso di qualsiasi modifica ad essa, viene invocato il metodo **SalvataggioCartaProgetto** che si occuperà di creare/aggiornare il file .json relativo a quella card. In caso di cancellazione del progetto il metodo **CancellaCartella** cancellerà la cartella di backup del progetto e tutti i file contenuti al suo interno (ipchat, membri, cards...).



4. Connessioni TCP e UDP

Il **Client** ed il **Server** instaurano tra di loro una connessione TCP sulla

porta 9876 per l'invio dei comandi. Il server non è multithreaded ma sfrutta il multiplexing dei canali, che permette di gestire più connessioni mediante un unico thread. Sulla connessione TCP saranno quindi inviati i comandi richiesti dal client e le relative risposte del server. L'operazione di registrazione di un nuovo utente utilizza il meccanismo *RMI (Remote Method Invocation)*, che permette ad un client di invocare un metodo di un server remoto. Il metodo **Register()** della classe server restituirà al client un codice che indicherà l'esito dell'operazione richiesta. Successivamente all'operazione di login, il client si registra ad un servizio di notifica del server per ricevere gli aggiornamenti sullo stato degli utenti registrati. Tali notifiche asincrone sfrutteranno il meccanismo delle *Callbacks* via *RMI*; in questo caso sarà quindi il



Server che invocherà dei metodi del Client.

Il servizio di chat del progetto prevede l'utilizzo del protocollo *UDP*. L'utilizzo di indirizzi di multicast permette ad ogni utente di inviare direttamente messaggi ai partecipanti di uno stesso progetto senza doverli "dirottare" verso il server. Il server avrà il compito di determinare un indirizzo di multicast da assegnare al progetto al momento della sua creazione; il metodo **GeneraMulticastAddress()** permette di generare un indirizzo di multicast in modo randomico, controllando inoltre che non ci siano altri progetti in WORTH con lo stesso indirizzo.

```
private InetAddress GeneraMulticastAddress(){
    if(listaprogetti.size()==(256^3*16)) {
        System.out.println("ServerWORTH: non ho piu' indirizzi multicast da assegnare");
        return null;
    }

    int a,b,c,d;
    a=(int)(Math.random()*16);
    a+=224;
    b=(int)(Math.random()*256);
    c=(int)(Math.random()*256);
    d=(int)(Math.random()*256);
    String address=a+"."+b+"."+c+"."+d;
    InetAddress ia=null;
    try {
        ia = InetAddress.getByAddress(address);
    } catch (UnknownHostException e) {
        e.printStackTrace();
    }

    for(Progetto p : listaprogetti) {
        if(p.GetIpChat().equals(ia)) {
            return GeneraMulticastAddress();
        }
    }
    System.out.println("ServerWORTH: L'indirizzo generato e' -> "+address);
    return ia;
}
```

Nel momento in cui in un client connesso a WORTH viene effettuato il login di un certo utente, il server provvede a far partire l'esecuzione di alcuni Thread della classe *ChatReaderWriter* che rimangono in attesa di ricevere nuovi messaggi su un indirizzo di multicast. Per far ciò il server invocherà il metodo **AvviaRicezione** del client (*Callback RMI*) passando come parametro un indirizzo di multicast. Per ogni utente autenticato vengono quindi attivati un numero di Thread pari al numero di progetti a cui esso partecipa. Nel momento in cui l'utente effettuerà l'operazione di logout, il client provvederà ad interrompere tutti i thread ascoltatori avviati in precedenza. In caso di cancellazione di un progetto, il server invocherà il metodo **StopRicezione(address)** su quei client dove sono attualmente autenticati membri del progetto da cancellare; il metodo invierà un messaggio fittizio sul gruppo in modo che i thread ascoltatori che sono bloccati sull'operazione *socketmulticast.receive(pacchetto)* possano svegliarsi e verificare di essere stati interrotti tramite una *interrupt()*; Una tecnica simile è adottata anche quando un utente si disconnette da WORTH, in tal caso il messaggio inviato sarà significativo ("WorthChatServe: *Utente si è disconnesso*"). Quando un utente viene aggiunto ai membri di un progetto, se questo risulta essere online (quindi iscritto al servizio di notifica), il server attiva sul suo client un thread ascoltatore di chat tramite *Callback*.

Le operazioni di aggiunta carta al progetto e spostamento carta comportano l'invio

automatico di un messaggio di notifica sulla chat del progetto.

```
public void AvviaRicezione(InetAddress address,String nomeproj) throws RemoteException{
    ChatReaderWriter newreader=new ChatReaderWriter(address,portaUDP,nomeproj);
    this.listaascoltatori.add(newreader);
    newreader.start();
}

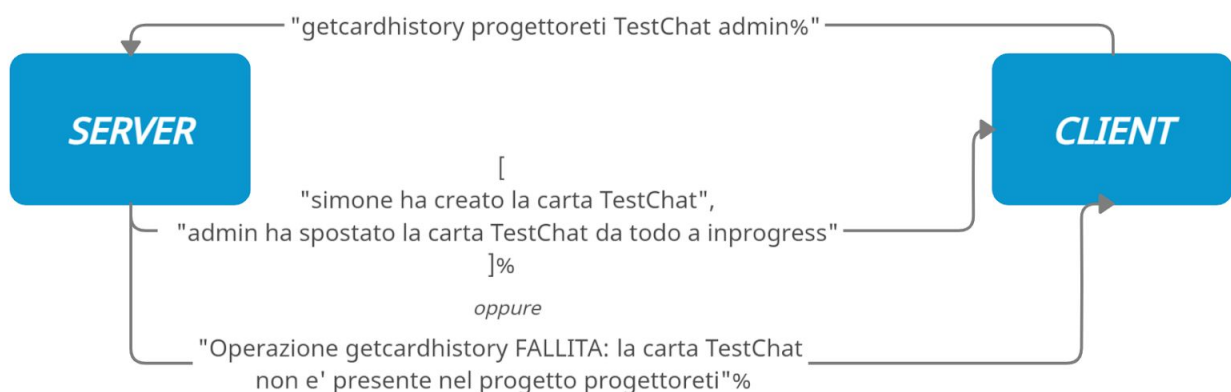
public boolean StopRicezione(InetAddress address) throws RemoteException {
    for(ChatReaderWriter c : listaascoltatori) {
        if(c.GetAddress().equals(address)) {
            c.interrupt();
            c.InviaMessaggio("MESSAGGIO FITTIZIO");
            listaascoltatori.remove(c);
            return true;
        }
    }
    return false;
}

public void StopRicezione() {
    for(ChatReaderWriter c : listaascoltatori) {
        c.interrupt();
        c.InviaMessaggio("WorthChatService: "+this.utenteassociato+" si e' disconnesso");
    }

    listaascoltatori.clear(); //cancello anche la lista degli ascoltatori
}
```

5. Formato dei messaggi client-server

Nel progetto è stato utilizzato il NIO per le comunicazioni client server, quindi si è reso necessario sviluppare un protocollo che definisse il formato dei messaggi scambiati tra client e server. Con NIO non è possibile leggere direttamente su un canale o scrivere su esso; è necessario quindi disporre di buffer che contengono dati letti dal canale o che vanno scritti sul canale. Poichè la dimensione del buffer di ricezione potrebbe essere minore della dimensione dei dati inviati sul canale, vi è il rischio di perdere parte del messaggio inviato dall'altro capo della comunicazione; è quindi previsto che i messaggi inviati debbano contenere un carattere terminatore (%) in modo che il ricevente possa riscontrare di aver ricevuto interamente il messaggio, altrimenti si effettuano più letture ricostruendo la stringa inizialmente inviata.



```

while(!comandocompleto) {
    ByteBuffer input=ByteBuffer.allocate(BufferDIM);
    byteletti=client.read(input);
    //System.out.println("Ho letto "+byteletti+" bytes");

    if(byteletti!=-1){
        String failed=this.useraddress.get(client.getRemoteAddress());
        if(failed==null) {
            //client non autenticato
            System.out.println("ServerWORTH: Chiusura connessione con "+client.socket().getRemoteSocketAddress());
        }else {
            //client autenticato
            Utente u=this.GetUserByName(failed);
            System.out.println("ServerWORTH: Chiusura connessione con "+failed+client.socket().getRemoteSocketAddress());
            u.SetOffline();
            UpdateStatus(failed,"offline");
        }
        keyestratta.cancel();
        keyestratta.channel().close();
        continue;
    }

    String msgricevuto=new String(input.array());
    sb.append(msgricevuto);
    if(msgricevuto.contains(terminatore)) {
        //Ho letto tutto il messaggio
        comandocompleto=true;
    }
}
String a=sb.toString().trim();
if(a.length()>0){
    //rimuovo dalla stringa ricevuta il carattere terminatore che si trova in fondo
    String comandoricevuto=a.substring(0, a.length()-1);
}

```

6. Istruzioni per la compilazione

La cartella di progetto conterrà al suo interno il file .jar della libreria *gson* utilizzata per la serializzazione. Aprire un terminale nella cartella WORTH. Digitare su di esso i seguenti comandi:

```
javac -cp .:gson-2.8.6.jar *.java
```

```
java -cp .:gson-2.8.6.jar Server
```

Su questo terminale sarà attivo quindi il server Worth. Aprirne un altro per attivare il client WORTH digitando:

```
java -cp .:gson-2.8.6.jar Client
```