

Progetto WORTH

Corso: Laboratorio di Reti
Matricola: 564117
Barberi Federico
A.A. 2020/2021

Indice:

- Descrizione del problema
- Scelte progettuali
- Classi java
- Comunicazione Client/Server
- Schema thread e concorrenza
- Organizzazione Directory
- Compilazione

Descrizione del problema:

WORTH si presenta come uno strumento per la gestione collaborativa di progetti, ispirandosi alla metodologia Kanban. Prevede la registrazione di utenti al servizio tramite username e password, richiedendo univoco lo username all' interno del servizio. WORTH permette la creazione di progetti, dedicandogli sul file system uno spazio fisico mediante la creazione di una directory dedicata (che prende il nome del progetto). All' interno della directory i membri di quel progetto possono creare varie attività (chiamate card e identificate ognuna da un file json dedicato) e gestire e controllare la loro progressione. Gli stati di avanzamento di una card sono 4 : TODO, INPROGRESS, TOBEREVIDED, DONE; e seguono dei rigidi vincoli di precedenza, elencati qua sotto:
TODO: stato alla creazione, si può spostare solo in INPROGRESS;
INPROGRESS: si può spostare solo in TOBEREVIDED oppure DONE;
TOBEREVIDED: si può spostare solo in INPROGRESS oppure DONE;
DONE: non si può spostare, e una volta che tutte le card sono in stato di DONE, si può cancellare il progetto.

Scelte progettuali:

In sede di implementazione è stato scelto di realizzare la versione a riga di comando e di realizzare il Server tramite il multiplexing dei canali mediante NIO.
Sono state prese inoltre le seguenti scelte progettuali cercando di rispettare il più possibile le specifiche assegnate:

- **Backup:** è la prima operazione che esegue il Server all' avvio, i backup dei progetti e degli utenti vengono salvati in due file json distinti: BackupProject.json e BackupUsers.json, entrambi salvati nella directory Backup. Se all' avvio i file non esistono li crea e inizializza le liste dei progetti e degli utenti. Se invece esistono, le liste del Server vengono popolate con il contenuto dei file. Serializzazione e de serializzazione viene gestita tramite ObjectMapper, poiché come versione di json vengono utilizzati i tre jar: jackson-annotation, jackson-core, jackson-databind tutti alla versione 2.9.7.

- **Callback:** oltre alla callback richiesta dalle specifiche, riguardante nuove registrazioni utenti e accessi/disconnessioni, viene implementata anche una callback per l'aggiunta di un membro a un progetto e per la cancellazione di progetti, gli utenti interessati (membri) ricevono una notifica comprendente il nome del progetto e l'indirizzo Multicast dello stesso in modo tale da poter aggiornare la propria HashMap relativa ai progetti-indirizzi. La lista progetti-indirizzi viene ricevuta al login insieme alla lista utenti, le due liste sono separate da una marca di fine lista '*****' che il Client è in grado di riconoscere.
- **Cancellazione progetti:** oltre alla possibilità di eliminare un progetto quando tutte le card sono in stato di "DONE" viene assunto che sia possibile anche eliminare un progetto che sia privo di card.
- **Card:** una card viene persistita sul file system con un file json dedicato, nomecard.json al cui interno sono scritte tutte le informazioni sulla card ed è salvata nella directory di progetto a cui appartiene.
- **Chat:** La chat viene realizzata mediante un ThreadPool, ogni Client, quando è membro di almeno un progetto, avvia il ThreadPool 'ClientChatHandler', il quale fa il submit di un task per ogni progetto di cui siamo membri, il task 'ClientChatListener' riceve il nome del progetto e il suo indirizzo Multicast e sta in ascolto su quella chat fino a quando non viene chiuso il pool (alla logout), quando riceve un messaggio sulla chat lo salva nella struttura dati condivisa (ConcurrentHashMap) che ha come chiave il nome del progetto e come valore il contenuto della chat di quel progetto. Quando un Client la richiede viene re inizializzata in modo tale da far vedere ad ogni Client i messaggi più recenti ovvero tutti i messaggi ricevuti a partire dall' ultima esecuzione del comando di visualizzazione messaggi. Per essere fedeli alle specifiche è stato scelto di implementare un pool per ogni Client, così risulta molto più semplice leggere solo i messaggi ricevuti dall' ultima esecuzione del comando. Viene inoltre assunto che ogni Client è in grado di ricevere i messaggi della chat solo quando è online, tutti i messaggi inviati sulla chat mentre un Client è offline vanno persi.
- **Exit:** per uscire dal programma è obbligatorio effettuare prima la logout e successivamente digitare exit, altrimenti verrà stampato un messaggio di errore.
- **Illegal Statement:** 'Backup' come nome progetto, dato che è il nome della cartella privata del Server per effettuare backup e '*****' come nome utente, dato che è la marca di separazione tra liste quando un Client esegue il login.
- **Liste private Client:** oltre alla lista locale degli utenti (realizzata tramite HashMap con chiave il nome utente e valore lo stato online/offline) al login l'utente riceve anche una lista dei progetti di cui è al momento membro dal Server, localmente anche questa lista viene realizzata tramite HashMap con chiave il nome del progetto e valore l'indirizzo Multicast associato al progetto.
- **Nuove registrazioni:** viene assunto che è possibile registrare nuovi utenti finché non siamo loggati, una volta effettuato il login non è più possibile effettuare nuove registrazioni finché non viene chiamata la logout. Inoltre, la registrazione non

coincide con l'accesso, una volta registrato un utente per essere loggato al servizio deve effettuare il login, altrimenti risulterà offline.

- **Request Line (Client)** : con requestLine intendiamo il comando che inviamo dal Client al Server, che sarà della forma :

cmd [args] nickUtente

Oltre al comando e i vari argomenti, se presenti, viene sempre aggiunto il nickUtente di chi fa la richiesta, in alcuni casi verrà ignorata, ma in altri casi risulta utile per far capire al Server chi è l'utente che fa la richiesta, e.g., nella createproject viene passato come argomento solo il nome del progetto da creare, però al Server è utile sapere chi fa la richiesta poiché deve aggiungerlo come membro di progetto.

- **Sintassi:** WORTH è case sensitive, i comandi lato Client devono essere digitati in minuscolo altrimenti non viene riconosciuto (es: register OK, Register ERR), i comandi seguono il nome e i parametri dati nella sezione 2 (Specifica delle operazioni), ad eccezione fatta per il nome del comando 'sendChatMsg' che per semplicità è stato trasformato in 'send'. In ogni caso è previsto un comando 'help' con la descrizione e sintassi delle varie operazioni.
Per quanto riguarda la sintassi dei parametri non sono ammessi spazi nei nomi utente, nei nomi dei progetti, nei nomi delle card, nelle descrizioni delle card, sono però ammessi caratteri speciali come '_', mentre gli spazi sono permessi quando si manda un messaggio sulla chat.
- **Univocità indirizzi Multicast:** viene assunto che la porta Multicast sia sempre la solita e conosciuta a priori dal Client, come le porte TCP e rmi, ed è la porta 30000. L'univocità viene garantita dall'indirizzo scelto casualmente nell'intervallo di indirizzi Multicast che va da 224.0.0.0 a 239.255.255.255. (Il Server controllerà che sia effettivamente un indirizzo Multicast tramite la funzione isMulticastAddress(), e che non sia già stato assegnato ad un altro progetto).

Classi java:

Il progetto si compone di 2 moduli interfaccia e 9 classi java.

Interfacce:

- **NotifyEventInterface:** Interfaccia che modella le funzioni della callback, comprende due metodi, notifyUsers per notificare nuove registrazioni o cambi di stato e notifyProject per notificare nuove aggiunte di membri ad un progetto oppure la cancellazione di un progetto. L'interfaccia viene implementata nella classe Client.
- **RMIInterface:** Interfaccia che modella le funzioni per registrarsi e deregistrarsi alla callback e la funzione di registrazione utenti. L'interfaccia viene implementata nella classe Server.

Classi:

- **Card:** classe che modella una card, ogni card è dotata di un nome, una descrizione, lo stato in cui risiede attualmente (todo, inprogress, toberevised, done) e una history degli stati che ha attraversati nel suo 'ciclo di vita'. Classe Serializable, poiché

interagisce con json per questo motivo è dotata anche del costruttore di default e dei metodi setters & getters.

- **Client:** Classe che implementa tutte le funzioni del servizio WORTH dal lato del Client, inoltre implementa le funzioni della NotifyEventInterface. Alla creazione vengono salvati tutti i numeri di porta della comunicazione e vengono create le liste locali. All'avvio, tramite metodo start(), invece viene stabilita una connessione TCP con il Server, successivamente viene fatto il setup per l' RMI e la callback, allocati buffer per la comunicazione e poi si mette in attesa di comandi. Ogni comando ricevuto viene tokenizzato, controllato all' interno di uno switch case e passato alla funzione corrispondente, se non c'è match viene stampato un messaggio di errore, suggerendo di usare 'help' per ricevere un menu di aiuto.
Ogni funzione eseguirà sempre un controllo per vedere se si è o meno loggati (a seconda delle richieste di una operazione) e successivamente un controllo sui parametri, superati questi due controlli vengono effettuati altri controlli se richiesti e poi:
La funzione register utilizza il metodo RMI per la registrazione di un utente e stampa l'esito dell'operazione.
Le funzioni readChat e sendChatMsg comunicano con il ThreadPool e sul gruppo Multicast del progetto interessato.
Le funzioni listUsers e listOnlineUsers utilizzano le liste private del Client.
Tutte le altre funzioni invece vengono passate al Server sulla connessione TCP (tramite la funzione ausiliaria ServerComunication) e viene letto e stampato a schermo l'esito. Il Server ad ogni operazione può rispondere con < ok, + descrizione in caso di successo oppure < errore, + descrizione in caso di fallimento.
- **ClientChatHandler:** Classe che modella un ThreadPool per la gestione delle chat, alla creazione prende come parametri la lista (HashMap) progettiIndirizzi di ogni Client, crea un CachedThreadPool e inizializza la ConcurrentHashMap chatList che conterrà i messaggi ricevuti sulla chat di ogni progetto. Successivamente avvia un task per ogni progetto presente nella lista, passandogli come parametri nome del progetto, indirizzo Multicast, e un'istanza di chatList. La classe è anche dotata di un metodo per effettuare lo shutdown del pool, di un metodo per sottomettere nuovi task (nel caso veniamo aggiunti a nuovi progetti) e di un metodo per recuperare la chat di un progetto.
- **ClientChatListener:** Classe che modella un task del pool. Alla creazione riceve come parametri il nome di un progetto, il suo indirizzo Multicast, e un'istanza di chatList. Nel metodo run() il task crea una MulticastSocket e si unisce al gruppo Multicast, grazie all' indirizzo passato e alla porta che è nota. Successivamente alloca il buffer di ricezione e il DatagramPacket e si mette in ascolto, quando riceve un pacchetto aggiorna la chatList.
- **MainClassClient:** Classe che crea un Client, passandogli numero di porta TCP, RMI e MULTICAST e successivamente avvia il Client tramite il metodo Start().
- **MainClassServer:** Classe che crea un Server, passandogli i numeri di porta per la comunicazione (TCP, RMI, MULTICAST), esegue il setup per l' RMI creando stub e registry e poi pubblicando lo stub nel registry, e poi avvia il Server tramite il metodo Start().
- **Project:** classe che modella un progetto, ogni progetto è dotato di un nome (univoco), di un indirizzo Multicast per la chat, di una lista di membri del progetto e delle 4 liste per le card : lista TODO, lista INPROGRESS, lista TOBEREVED, lista DONE.

Classe Serializable poiché interagisce con json (tutti i progetti vengono scritti sul file BackupProject.json) per questo motivo è dotata anche del costruttore di default e dei metodi setters & getters. Oltre a questi metodi presenta anche una serie di metodi di utilità quali deleteCard per la eliminazione di una card, moveCard per spostare una card da una lista di partenza a una lista di destinazione, vincoliDiPrecedenza per vedere se lo spostamento di una card è consentito, metodi per vedere se un utente/una card è membro/appartiene al progetto ecc..

- **Server:** Classe che implementa le funzioni del servizio WORTH che appartengono alla connessione TCP, la funzione di register dato che implementa RMIInterface e le funzioni per effettuare le notifiche tramite callback. Alla creazione il Server si memorizza i numeri di porta della comunicazione, crea le proprie liste private, inizializza l'ObjectMapper per l'interazione con file json e avvia il backup per ricostruire lo stato del sistema. All'avvio il Server esegue il set up della connessione TCP e successivamente, dato che è stato scelto di implementarlo tramite selector, si mette in ascolto iterando sulle chiavi. Ad ogni richiesta di connessione alloca i buffer per la comunicazione e registra un interesse in lettura, dato che si aspetta di ricevere un comando. Una volta letto il comando richiama la funzione in grado di soddisfare la richiesta del Client e registra l'interesse di scrittura al Client, dato che gli comunicherà l'esito dell'operazione. Inoltre, ogni qualvolta ci siano delle modifiche alle liste utenti, liste progetti oppure cards aggiornerà il file json relativo tramite la funzione jsonUpdate.
- **Utente:** classe che modella un utente, ogni utente è dotato di un nickUtente(univoco), di una password e di uno stato(online/offline). Classe Serializable, poiché interagisce con json (tutti gli utenti vengono scritti sul file BackupUsers.json) per questo motivo è dotata anche del costruttore di default e dei metodi setters&getters.

Comunicazione Client/Server

Premessa: Sono stati scelti i seguenti numeri di porta per la comunicazione

- **TCP :** 5678
- **RMI :** 9000
- **MULTICAST :** 30000

Come richiesto dalle specifiche, non tutte le operazioni sono effettuate sulla connessione TCP instaurata tra Client e Server anzi, potremo dividere le operazioni in questo modo:

- **RMI method :** register.
- **TCP method :** login, logout, listprojects, createproject, addmember, showmember, showcards, showcard, addcard, movecard, getcardhistory, cancelproject, help, exit.
- **Local method :** listusers, listonlineusers.
- **UDP Multicast method :** readchat, send.

Il Client prima di eseguire qualsiasi operazione fa sempre un controllo sullo stato di logged, e sui parametri passati in modo tale da mandare al Server la requestLine nella forma sintatticamente corretta, starà poi al Server effettuare i vari controlli semantici che possono essere : utente già registrato, progetto inesistente, card già presente ... Il Client in alcuni casi specifici può effettuare ulteriori controlli (i.e. controllare se si è membri o meno di un progetto per le operazioni sulla chat ..).

RMI method

L' unica operazione che richiede l'uso di RMI è la register, lato Client prende come Parametri la stringa relativa alla requestLine tokenizzata mediante il metodo split (come regola di tokenizzazione assumiamo gli spazi), un'istanza dell'interfaccia RMI. Per richiamare la funzione register implementata sul Server.

Lato Server viene effettuato un controllo sui parametri che siano diversi da null e che l'utente non sia già registrato, se tutto va a buon fine il Server aggiorna la propria lista Utenti, il file BackupUsers.json utile per il ripristino del sistema, invia una notifica tramite callback e restituisce SUCCESS, altrimenti viene restituito ERROR o lanciata un'eccezione se i parametri sono null.

Oltre al metodo register Client e Server implementano anche i vari metodi utili per il meccanismo di RMI Callback quali :

Client : notifyUsers e notifyProjects della NotifyEventInterface.

Server: registerForCallback e unregisterForCallback della RMInterface, doCallbackUsers, updateUsers, doCallbackProjects e updateProjects che sono i metodi che "eseguono" le callback.

TCP method

I metodi TCP vengono effettuati sulla connessione TCP instaurata tra Client e Server.

Lato Client prendono tutti come parametri la requestLine tokenizzata, e un'istanza Della SocketChannel del Client. Una volta controllato che la sintassi sia corretta viene richiamata una funzione ausiliaria privata del Client,

ServerComunication(SocketChannel Client), che si occupa di mandare effettivamente la requestLine al Server, di ricevere l'esito dell'operazione e restituirlo alla funzione chiamante in modo da stamparlo a schermo.

Lato Server una volta ricevuta una richiesta si elabora, si effettuano i vari controlli semantici e se tutto corretto si prepara una stringa di esito positivo, e si registra l'interesse in scrittura dell'esito positivo (altrimenti viene restituito un esito negativo). Inoltre, se necessario, si aggiornano le liste private del Server, i file json per il Backup, si inviano notifiche tramite callback e, nel caso di una nuova card aggiunta a un progetto o di uno spostamento di una card, il Server manda un messaggio sulla chat di progetto.

Gli esiti sono della forma:

< ok, + descr (che di solito coincide con quello che si richiede al Server);

< errore, + descr (di quello che può essere andato storto).

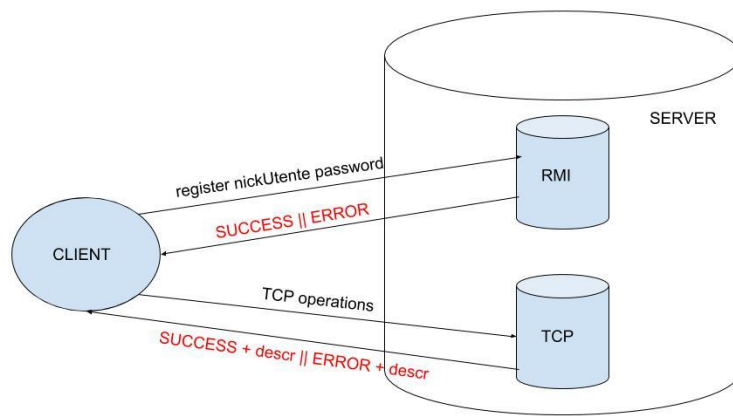
Vi sono alcuni casi particolari ad esempio:

login : lato Client oltre alla requestLine e all' istanza di SocketChannel viene passata anche un'istanza di RMInterface e NotifyEventInterface poiché, in caso di esito positivo ci registriamo alla callback.

Lato Server invece oltre al messaggio di esito positivo restituiamo anche la lista utenti registrati al servizio con il loro stato e la lista dei progetti di cui è membro l'utente con il loro indirizzo Multicast. Sarà poi al Client ricostruirsi le due liste, e se abbiamo anche almeno un progetto nella lista e il ThreadPool per la gestione delle chat non è ancora stato attivato viene attivato.

logout: lato Client oltre alla requestLine e all' istanza di SocketChannel viene passata anche un'istanza di RMInterface e NotifyEventInterface poiché in

caso di esito positivo, viene effettuata la de registrazione alla callback e chiuso il ThreadPool delle chat.



TCP operations:
login, logout, listprojects, createproject, addmember,
showmembers, showcards, showcard, addcard,
movecard, getcardhistory, cancelproject, help, exit

Local method

I metodi locali non “lasciano” il Client ma, sfruttano la lista locale degli utenti ricevuta al login, mostrando la lista completa degli utenti registrati al servizio oppure la lista degli utenti online in quel momento. Le liste locali del Client vengono aggiornate tramite callback quindi hanno sempre un valore consistente. Vengono comunque fatti i controlli sullo stato del login, poiché bisogna avere eseguito l’accesso per richiamare certe operazioni, e i controlli dei parametri passati.

UDP Multicast method

I metodi Multicast interagiscono, direttamente o indirettamente, con il ThreadPool. Per la gestione della chat tramite Multicast ogni Client avvia un proprio ThreadPool appena è membro di almeno un progetto (Un Client si rende conto di essere diventato membro di un progetto quando riceve dalla callback una notifica e la coppia projectName – MulticastAddress, oppure quando è lui stesso a creare un progetto e in quel caso oltre all’ esito positivo riceverà il nome del progetto e l’indirizzo Multicast associato, da salvarsi nella propria lista locale progettiIndirizzi). Il Client avvia il ThreadPool sottomettendo un task per ogni progetto di cui è membro, ogniqualvolta creerà o diventerà membro di un nuovo progetto utilizzerà la funzione ‘nuovaChatProgetto(projectName, MulticastAdress)’ per sottomettere un nuovo task al pool.

Il metodo readChat recupera la lista dei messaggi che l’utente non ha ancora letto di un determinato progetto, ad ogni readChat viene re inizializzato il campo value (che contiene il contenuto della chat) di chatList, in modo tale da ricevere solo i messaggi ricevuti a partire dall’ ultima esecuzione del comando.

Il metodo sendChatMsg invece recupera l’indirizzo Multicast dalla lista progettiIndirizzi e invia un DatagramPacket contenente il messaggio sul gruppo Multicast relativo al progetto.

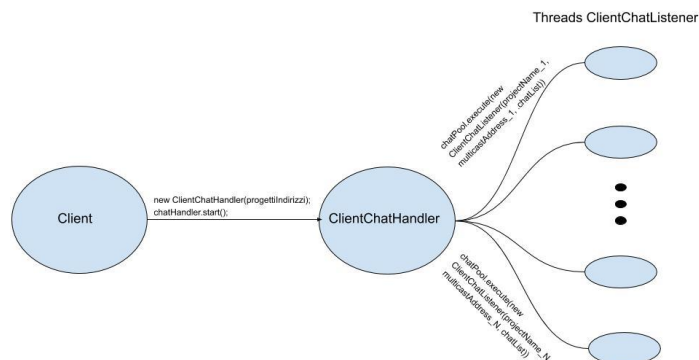
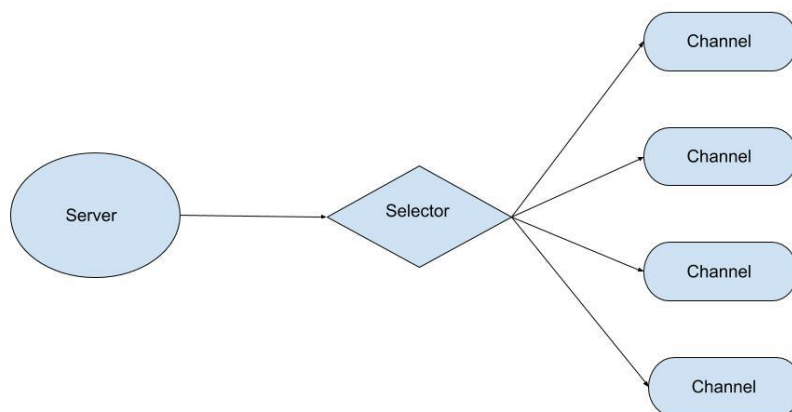
Il Server invece non interagisce direttamente con il ThreadPool, ma può mandare dei messaggi sulla chat, i messaggi sono relativi all’ aggiunta di nuove card o allo spostamento delle stesse, non vengono inviati messaggi per nuove aggiunte membri

o per cancellazioni di progetto dato che gli utenti vengono già informati tramite callback. Per inviare messaggi sulla chat il Server utilizza le informazioni sui progetti e invia un DatagramPacket contenente il messaggio.

Schema thread e concorrenza

Qua sotto viene riportato uno schema, molto semplificato, dei thread attivati dal Server (solo uno, il thread Server che gestisce i channels tramite selector) e dal Client per la gestione della chat attraverso un ThreadPool. I thread ClientChatListener interagiscono con una struttura dati condivisa che è la chatList, realizzata appositamente tramite ConcurrentHashMap. Per quanto riguarda il Server viene realizzato tramite multiplexing dei canali come detto in precedenza, per cui situazioni di criticità si hanno solo per la funzione register, registerForCallback, unregisterForCallback, doCallbackUsers, doCallbackProject, opportunamente definite synchronized.

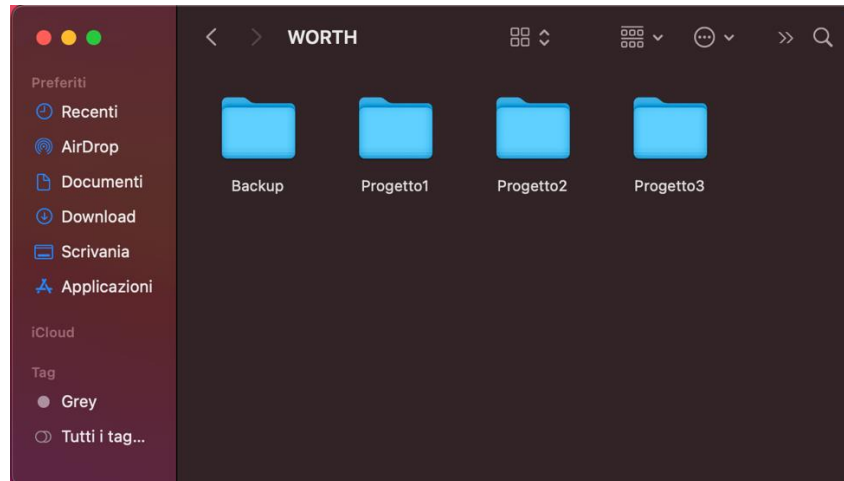
Per quanto riguarda la chiusura delle Socket aperte (ServerSocketChannel nel Server, SocketChannel nel Client e MulticastSocket nei thread di ClientChatListener) non vengono chiuse esplicitamente, ma vengono aperte sfruttando il meccanismo di try-with-resource che si occupa lui stesso di chiuderle.



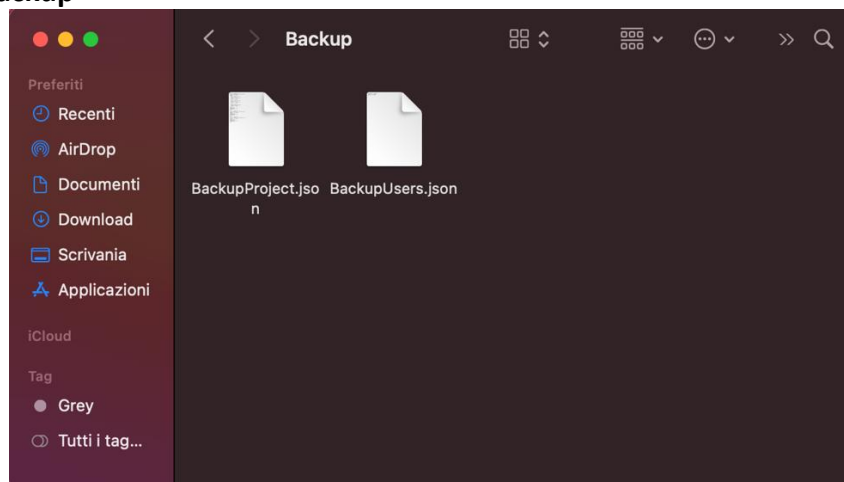
Organizzazione Directory:

Qua sotto viene riportato un esempio di come è organizzata la directory WORTH, creata e gestita dal Server, mostrando e riassumendo un po' tutto ciò che è stato elencato in precedenza. Per questo esempio sono stati creati 3 progetti: Progetto1, Progetto2, Progetto3 e, all' interno di Progetto1, sono state create 3 card: Card1, Card2, Card3.

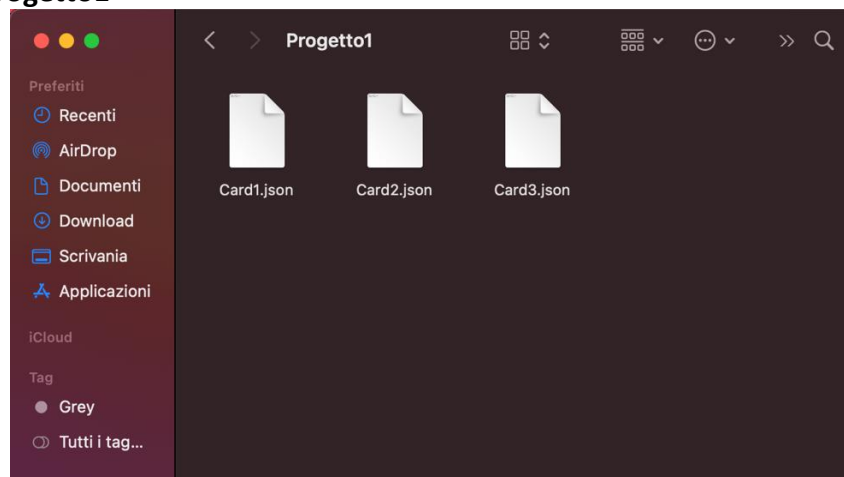
./WORTH



./WORTH/Backup



./WORTH/Progetto1



Compilazione

Il progetto è stato interamente svolto e testato su ambiente MacOS, utilizzando come JDK la versione 11.

È stato sviluppato interamente usando Eclipse come IDE e, non utilizzando librerie esterne eccetto il package 2.9.7 di Jackson che comprende (elencati per dipendenze):

- Jackson-annotations-2.9.7.jar
- Jackson-core-2.9.7.jar
- Jackson-databind-2.9.9.jar

I 3 jar sono forniti nella directory "jars" del progetto.

Step compilazione e avvio:

Per prima cosa è necessario compilare tutti i file e avviare il Server.

Spostandosi nella directory di progetto è necessario compilare tutti i file, specificando nel classpath la posizione dei jar per Jackson, che saranno tutti contenuti nella directory "jars":

```
javac -cp ".../jars/jackson-annotations-2.9.7.jar:./jars/jackson-core-2.9.7.jar:./jars/jackson-databind-2.9.7.jar:" *.java
```

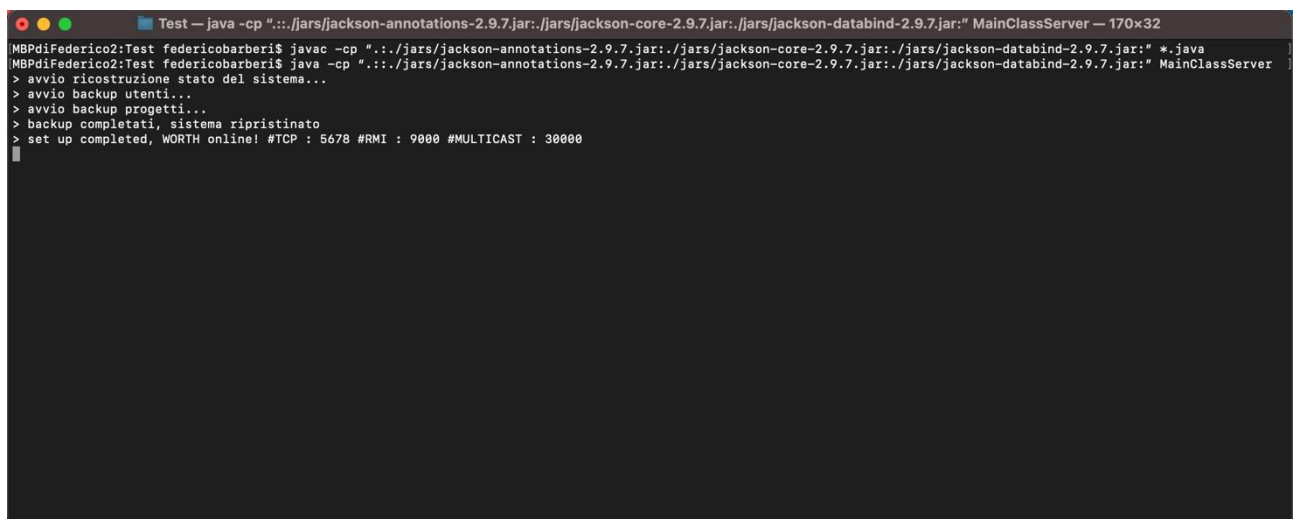
A questo punto è necessario avviare il Server attraverso il suo metodo main, anche questa volta è necessario passare il Classpath:

```
java -cp ".../jars/jackson-annotations-2.9.7.jar:./jars/jackson-core-2.9.7.jar:./jars/jackson-databind-2.9.7.jar:" MainClassServer
```

Ora non resta che eseguire il Client dal suo metodo main :

```
java MainClassClient
```

Qua sotto vengono riportati due screen, il primo di compilazione file java e avvio Server, il secondo di avvio Client e richiamando il comando help per visualizzare il menu di aiuto.



```
Test - java -cp ".../jars/jackson-annotations-2.9.7.jar:./jars/jackson-core-2.9.7.jar:./jars/jackson-databind-2.9.7.jar:" MainClassServer - 170x32
MBPdiFederico2:Test federicobarberi$ javac -cp ".../jars/jackson-annotations-2.9.7.jar:./jars/jackson-core-2.9.7.jar:./jars/jackson-databind-2.9.7.jar:" *.java
MBPdiFederico2:Test federicobarberi$ java -cp ".../jars/jackson-annotations-2.9.7.jar:./jars/jackson-core-2.9.7.jar:./jars/jackson-databind-2.9.7.jar:" MainClassServer
> avvio ricostruzione stato del sistema...
> avvio backup utenti...
> avvio backup progetti...
> backup completati, sistema ripristinato
> set up completed, WORTH online! #TCP : 5678 #RMI : 9000 #MULTICAST : 30000
```

```
Test — java MainClassClient — 89x26
[MBPdiFederico2:Test federicobarberi$ java MainClassClient
----- WELCOME TO WORTH! -----

help
----- WORTH MENU -----
Sintassi comandi(in minuscolo) e parametri da passare :
1) register nickUtente password
2) login nickUtente password
3) logout nickUtente
4) listusers
5) listonlineusers
6) listprojects
7) createproject projectName
8) addmember projectName nickUtente (Project Member only)
9) showmembers projectName (Project Member only)
10) showcards projectName (Project Member only)
11) showcard projectName cardName (Project Member only)
12) addcard projectName cardName descrizione (Project Member only)
13) movecard projectName cardName listaPartenza listaDestinazione (Project Member only)
14) getcardhistory projectName cardName (Project Member only)
15) readchat projectName (Project Member only)
16) send projectName messaggio (Project Member only)
17) cancelproject projectName (Project Member only)
18) exit
```