

**Gabriele Ferrigno 603246**

**Relazione progetto Reti 2020/2021**

**WORTH: WORkTogetHer**

1. Cosa e’ WORTH
2. Architettura del Progetto

1.1) Il Server

1.2) Il Client

2. Strutture dati utilizzate

2.1) Lato Server

2.2) Lato Client

1. Meccanismo di Callback
2. Implementazione della chat
3. Implementazione del Backup
4. Gestione della concorrenza
5. Librerie esterne utilizzate
6. Struttura del menu

**Cosa e’ WORTH**

WORTH è uno strumento utilizzato per la collaborazione tra persone su dei progetti, le persone tramite username e password potranno accedere alla piattaforma WORTH e creare o partecipare a nuovi progetti, creare o modificare card appartenenti ai progetti di cui fanno parte il tutto collaborando ed interagendo tra di loro anche tramite chat.

**Architettura del Progetto**

1. Il server:

Il server è di tipo *Multiplexing*, all’avvio del server il costruttore inizializzerà tutte le strutture dati e successivamente cercherà un backup se lo troverà ripristinerà lo stato precedente.

Successivamente tramite la classe *ServerSocketChannel* e *ServerSocket* creerà i socket necessari per instaurare nuove connessioni con i client, gli indirizzi usati sono quelli locali e la porta utilizzata è una porta di default, in questo caso è la porta: 5000.

Tramite un selettore il server si mette in attesa di nuove connessioni da parte dei client utilizzando la funzione *selector.select(),* all’arrivo di una nuova connessione o di un comando verrà prelevata la *SelectionKey* legata ad un determinato client e verrà , a seconda del tipo di key2, effettuata un’operazione di connessione, di lettura di un comando o di scrittura del risultato di un’operazione.

1. Il Client:

Il client avrà il compito di inviare i comandi al server in base alle scelte dell’utente, come prima cosa chiederà all’utente di registrarsi o di effettuare il login, alla registrazione il client instaurerà una connessione con il server tramite TCP, in seguito il client mostrerà nuovamente i comandi per registrarsi ed effettuare il login

(in questo modo evitiamo un controllo a livello di codice sullo stato del client, ad esempio se l’utente che si sta connettendo con un determinato client ha effettuato la login oppure no, perché’ i comandi relativi alla modifica dei progetti o delle card non verranno mostrati se un utente non esegue con successo un’operazione di login), a seguito di un’operazione di login eseguita con successo il client mostrerà tutti i comandi che permetteranno di interagire con i progetti.  
Tutti i comandi, eccetto quelli che operano sulla chat, riceveranno una risposta dal server, la risposta sarà un oggetto di tipo *Result,* in questo oggetto ci saranno tutte le informazioni relative all’esito della risposta ed eventuali strutture dati che il server vuole trasmettere al client.

**Strutture dati utilizzate**

1. Lato Server

Le strutture dati principali, utilizzate lato server, sono: 3 ConcurrentHashmap, una HashMap e una lista:

Immagine che contiene testo

Descrizione generata automaticamente

*projectList:* contiene la lista di tutti i progetti creati e ancora esistenti sul server.

*projectMemberBinding:* è una ConcurrentHashmap che usa come chiavi i nomi degli utenti e come valori degli arrayList con all’interno i nomi dei progetti di cui fa parte un determinato utente.

*registeredUsersData:* è una ConcurrentHashmap che usa come chiavi i nomi degli utenti e come valori delle stringhe che indicano la password dell’utente.

*usersStatus:* è una ConcurrentHashmap che associa un nome utente ad uno stato (online/offline)

*channelBinding:* è una Hashmap utilizzata per associate un socketChannel ad un determinato utente, sarà utilizzata per identificare un determinato client così da non dover passare tutte le volte anche il nome dell’utente.

1. Lato Client

Le strutture dati principali, utilizzate lato client, sono: una ConcurrentHashmap e 4 Hashmap.

Immagine che contiene testo

Descrizione generata automaticamente



*threadBinding:* questa HashMap associa il nome di un progetto ad un Future, sarà utilizzata per terminare il thread che si occupa della ricezione dei messaggi di un determinato progetto quando questo verrà cancellato.  
  
Future: È un oggetto ottenuto come risultato dalla creazione di un task, quando parleremo della chat verrà spiegato più in dettaglio.

*IPBinding:* HashMap che associa gli IP delle chat dei progetti ai progetti

*messageHistoryProjects:* HashMap che associa un Progetto ad un arraylist contenente i messaggi ricevuti sulla chat.

*lastReadMessageCounters:* una Hashmap che associa un progetto ad un contatore, questo contatore indica la posizione dell’ultimo messaggio letto di quel determinato progetto.

*listOnlineUsers:* questa ConcurrentHashMap e’ ottenuta dopo l’operazione di login, o a seguito di un’operazione di registrazione da parte di un utente (tramite Callback) e associa il nome di un utente al suo stato.

**Meccanismo di Callback**

L’interfaccia RMICallbackClient definisce 3 metodi utilizzati dal server per indicare che un determinato evento è accaduto e notificare tutti i client che si sono registrati alla Callback.  
Immagine che contiene testo

Descrizione generata automaticamente

*notifyFromEventServer:* Serve a notificare i client riguardo al cambiamento di stato di un utente oppure all’avvenuta registrazione di un nuovo utente, così da aggiornare la lista degli utenti registrati e il loro stato lato client.  
*addedtoNewProjectevent:* Utilizzato quando un utente aggiunge un altro utente ad un progetto, in questo caso l’utente che viene aggiunto riceve le informazioni riguardanti il progetto e l’indirizzo IP della chat di quel progetto

*projectRemoved:* Utilizzato quando un progetto viene rimosso, vengono avvisati tutti gli utenti facenti parte di quel progetto così che possano terminare il thread che si occupa di controllare i messaggi sulla chat.

**Implementazione della Chat**

La chat è stata implementata tramite *Multicast UDP*, in questo caso gli unici partecipanti a questo meccanismo sono i client, non c’è alcuna comunicazione con il server quando si invia/riceve un messaggio.  
Il server si occuperà solo di assegnare un indirizzo IP di Multicast unico e casuale ad ogni progetto tramite la classe *MIPManager,* tale classenon farà altro che generare un indirizzo IP randomico, controllare se non è già stato assegnato e salvarlo in un arrayList, il server tramite un’istanza della classe *MIPManager* potrà gestire gli indirizzi IP dei progetti.  
Un client alla creazione di un progetto o quando viene aggiunto al progetto riceverà una *Callback*, questa *Callback* non farà altro che aggiungere l’IP del progetto nella *HashMap* *IPBinding* e successivamente avvierà un *Thread* passando ad un *ThreadPool* una task di tipo *ProjectChatSniffer*.  
Subito dopo la creazione di un’istanza della classe *ProjectChatSniffer* nell’HashMap threadBinding verrà inserito un oggetto di tipo Future, ottenuto dall’operazione *executorChat.submit(task)* questo oggetto sarà utilizzato per interrompere il thread relativo alla chat di un progetto che è stato cancellato.

La classe *ProjectChatSniffer* (come suggerito dal nome della classe) fungerà da “sniffer” cioè creerà un *MulticastSocket* sull’indirizzo IP di un progetto e farà la join, dopodiché’ entrerà in un ciclo *while(true)* dove invocherà una *receive* e si metterà in ascolto di eventuali messaggi, all’arrivo di un messaggio invocherà il metodo *addMessage* del client così da aggiungere il messaggio all’arrayList di messaggi relativi al progetto.

Per determinare se un progetto è stato eliminato o meno (e quindi se una task deve terminare o meno) utilizzando il metodo *setSoTimeout*(4000) sul *multicastSocket*, ogni 4 secondi invocheremo una *SocketTimeoutException* e grazie ad un blocco try and catch potremo gestire questa eccezione in modo tale da controllare se quella task è stata terminata oppure no (in seguito alla cancellazione di un progetto).

Riguardo alla lettura di un messaggio in seguito alla ricezione del comando readChat nel client la procedura è molto semplice, quello che il client farà è prelevare l’arrayList dei messaggi relativi ad un determinato progetto *dall’HashMap* “*messageHistoryProjects*”, prelevare il contatore con all’interno la posizione dell’ultimo messaggio letto e verificare se ci sono nuovi messaggi da leggere, a quel punto stamparli sul terminale.

**Implementazione del Backup**

Per il backup è stata usata la libreria esterna *GSon* per serializzare i dati e salvarli in file di tipo json.  
Il server all’avvio controlla tramite la funzione *searchAndRestoreBackup* controlla se la cartella backupDir è presente, se lo e’ allora provvede a ripristinare le seguenti strutture dati:   
 - registeredUsersData

-projectList

-projectMemberBinding

E successivamente richiama *mipManager.restoreMipAddress()* che si occuperà di ripristinare il file contenente gli indirizzi IP già assegnati ai vari progetti.

Per il salvataggio delle varie informazioni che devono persistere anche quando il server viene terminato ho usato la funzione *setBackup* che viene richiamata periodicamente da un thread, questo thread, chiamato semplicemente timer, ogni 15 secondi eseguirà la funzione *setBackup* del server e quest’ultimo salverà tutte le informazioni nei file e nelle directory assegnate sempre tramite la libreria *GSon*.  
N.B. riguardo le card il server dovrà richiamare, per ogni progetto, la funzione backup definita e implementata nella classe *Project.*

**Gestione della concorrenza**

Lato server, avendo usato Multiplexing, le uniche race condition saranno date dalle operazioni di backup, più precisamente quando il server dovrà effettuare il backup dei progetti e delle carte, in questo caso le operazioni saranno svolte all’interno di un blocco *synchronized* che ci permetterà di acquisire la lock sulla struttura *projectList*.  
La stessa cosa avverrà all’interno dei singoli progetti quando dovrò salvare ogni singola carta in un file diverso, attraverso un blocco *synchronized* potrò gestire la concorrenza.

Lato client invece gli unici thread che avremo saranno quelli relativi alle chat, in questo caso nella funzione *addMessage* sarà presente un blocco *synchronized* che si occuperà di bloccare la risorsa *messageHistoryProjects*

In alcuni casi particolari sono state usate delle ConcurrentHashMap e sfruttando l’operazione atomica putIfAbsent potremo garantire una gestione adeguata della concorrenza.

**Librerie esterne utilizzate**

L’unica libreria esterna utilizzata è GSon, la versione utilizzata è la 2.8.8.  
Per aggiungere questa libreria al progetto (usando intelliJIdea) basterà premere su File -> Project Structure -> Libraries ed infine premere sul + e recarsi nella directory dove abbiamo salvato il file .jar che fa riferimento alla nostra libreria, premere su Apply ed infine su OK.  
Un metodo alternativo è quello di aggiungere la dipendenza direttamente da maven.  
<https://github.com/google/gson> qui possiamo trovare una spiegazione completa della libreria.

<https://search.maven.org/artifact/com.google.code.gson/gson/2.8.8/jar> mentre questo è il link da usare per scaricare il file .jar della libreria GSon.

**Struttura del menù**

Il menù del client sarà presentato come una lista di comandi, ogni comando sarà identificato da un numero, per selezionare un determinato comando basterà selezionare scrivere il numero relativo al comando e premere invio.

I menù sono essenzialmente 2, uno sarà mostrato all’avvio del client e conterrà i comandi di registrazione e di login, a seguito di un login avvenuto con successo sarà mostrato il secondo menù, contenente i comandi rimanenti.

Alla fine di ogni operazione sarà mostrato il risultato sul terminale ed un messaggio avviserà l’utente che l’operazione è terminata, se vorrà tornare al menù dovrà premere 0 ed invio, questa piccola aggiunta è stata fatta per garantire una migliore leggibilità del risultato di ogni operazione.

Immagine che contiene testo

Descrizione generata automaticamente

Immagine che contiene testo

Descrizione generata automaticamente