UNIVERSITÀ DI PISA
CORSO DI PROGRAMMAZIONE DI RETI A.A. 2020/2021
RELAZIONE PROGETTO
"WORTH"

Alessio Matricardi – Matricola 580159 – aggiornato al 14/01/21

Descrizione dell'architettura

Il progetto si compone di 2 parti:

- Componente server
- Componente client

Comunicazione

Le due entità comunicano tra loro eseguendo la maggioranza delle operazioni, ad eccezione della registrazione, utilizzando una connessione TCP persistente per tutta la durata della comunicazione. Attraverso la connessione TCP instaurata, client e server utilizzano un protocollo di comunicazione per lo scambio di messaggi, costruito sulla struttura dei messaggi HTTP. Infatti, client e server si scambiano messaggi di richiesta-risposta con la seguente struttura:

Messaggio di richiesta (RequestMessage)

- Command: operazione che il client richiede al server
- Arguments: sequenza degli argomenti che il client fornisce al server per elaborare la risposta

Messaggio di risposta (ResponseMessage)

- Status code: codice di risposta alla richiesta
- Response body: (opzionale, dipende dal comando) risorsa richiesta dal client

La lista dei comandi e degli status code fanno anch'essi parte del protocollo di comunicazione fra le due entità, così come le informazioni riguardanti IP e porte su cui il server è disponibile.

Il server, in base al comando ricevuto, proverà ad elaborare una risposta per il client. Questa risposta sarà comprensiva di status code (se la richiesta è terminata con successo o, se no, con quale errore). Il Response Body, ove presente, è una risorsa strutturata e con un formato concordato da client e server: il client sa che dovrà ricevere una lista piuttosto che una struttura map.

Oltre alla connessione TCP esplicita, il servizio di registrazione è affidato a RMI.

Per garantire un livello di sicurezza aggiuntivo, la registrazione di un utente al servizio include la cifratura della sua password mediante l'algoritmo SHA3-256 e seed casuale: anche se due utenti dovessero utilizzare la stessa password per accedere al servizio, i message digest prodotti saranno differenti tra loro. Ovviamente, per permettere che l'utente possa effettuare il login, il sistema memorizza, per ognuno di loro, la tripla <nome utente, hash password, salt utilizzato>.

Quando l'utente effettua il login, il server include nel messaggio di risposta (nel campo response body) la lista degli utenti e il loro rispettivo stato. Da qui in poi, l'utente verrà aggiornato in modo asincrono dal server attraverso il meccanismo delle RMI Callbacks. Fatto il login, infatti, l'utente si registra al servizio di Callback e viene aggiornato fino al momento del logout, in cui avviene la deregistrazione.

Oltre a TCP, nel progetto vi è anche l'utilizzo del protocollo UDP per il servizio di chat. Quando un utente desidera leggere la chat di un progetto richiede al server di indicargli quale indirizzo IP Multicast e numero di porta sono associati al progetto.

Dopo aver ricevuto una risposta (positiva) dal server, il client aprirà una MulticastSocket su cui ricevere e inviare messaggi sulla chat.
Dal momento in cui il client richiede di leggere quella chat, questa rimarrà attiva per tutta la sessione dell'utente (fino al logout). Questo implica che avrà la possibilità di recuperare eventuali messaggi non letti (poichè magari era impegnato a lavorare su di un altro progetto).

Eventuali messaggi antecedenti alla richiesta di lettura della chat NON saranno disponibili, e lo stesso vale per eventuali messaggi successivi al logout o alla chiusura dell'applicativo.
Inoltre, i messaggi non sono persistenti e non possono essere recuperati una volta fatto il logout o chiuso l'applicativo.

Si è deciso per questa forma parziale di persistenza per ovvie ragioni:

- Far persistere l'intera chat sul server comporta un enorme mole di lavoro su quest'ultimo
- Il server sarebbe implicato nell'intera comunicazione UDP (lo è, ma in parte, vedi dopo)
- Lo spreco di risorse sarebbe eccessivo

Server

Il server è implementato mediante lo schema del NIO Selector e SocketChannel non bloccanti. Questa scelta è stata effettuata poiché:

1. È una soluzione che scala bene quando il numero di connessioni instaurate (e, conseguentemente, il numero di richieste) cresce
2. Garantisce un ottimo livello di concorrenza
3. Garantisce la sequenzialità delle operazioni, questione che strizza l'occhio sulla questione della integrità dei dati.

Quando una connessione viene aperta, il server salva una struttura dati come allegato alla SocketChannel creata. Questa struttura dati, chiamata Attachment, non è altro che una coppia <buffer, username>. Il buffer è un ByteBuffer, utilizzato per le fasi di lettura/scrittura sulla SocketChannel. Molto più rilevante è, invece, lo username.
Quando un utente effettua il login con la socket instaurata in precedenza, il server salva lo username dell'utente: in questo modo, qualsiasi operazione effettuata da quella socket sarà a nome di quello specifico utente. Questo meccanismo permette al server di controllare che il client sia autorizzato a effettuare una specifica operazione: per esempio un utente X non può aggiungere una Card al progetto P se non vi è membro.

Nel momento in cui un client connesso chiuda l'applicativo, il server effettua il logout automatico dell'utente ed elimina la SocketChannel dal suo insieme. Questa procedura è stata preferita al logout automatico da parte del client nel momento della chiusura dell'applicativo per motivi di

robustezza e integrità: un crash improvviso del client avrebbe lasciato il suo stato (ONLINE) inalterato nel server.

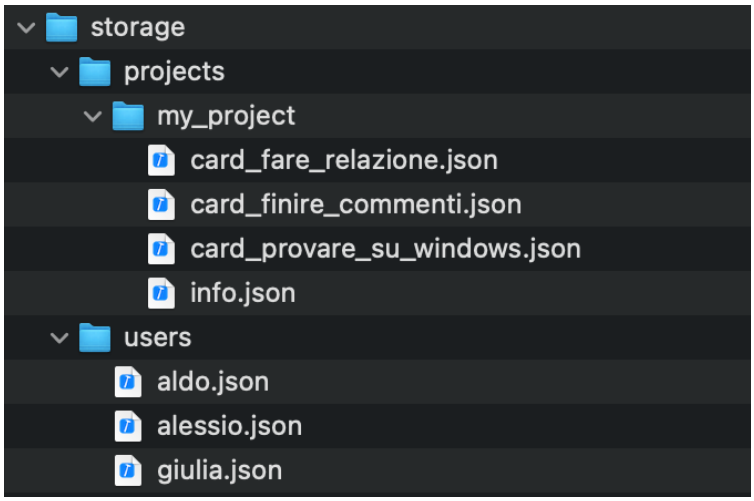
Questa scelta evita che l'utente risulti ONLINE quando non lo è. Inoltre, unita all'impossibilità di fare il login da più di una interfaccia utente, evita che l'utente non possa più accedere al suo account.

Quando un utente richiede lo spostamento di una card il server avvisa, mediante un datagramma UDP, l'avvenuto spostamento sulla chat del progetto. Oltre questo evento, il server informa la chat del progetto quando questo viene cancellato. Da questo momento in poi, qualsiasi operazione sul progetto restituirà messaggi di errore.

Persistenza dei dati

Al momento dell'inizializzazione, il server recupera tutti i dati dalla cartella "storage" e li carica in una sorta di "database" sotto forma di classe chiamato PersistentData.

La struttura di questa cartella struttura è la seguente



Ogni progetto ha una cartella dedicata. Al suo interno troviamo:

- File "info.json": dove sono salvate le informazioni generali del progetto (nome, lista membri, liste cards, data creazione)
- Lista di file "card_....json": dove sono salvate le informazioni specifiche della carta (nome, descrizione, stato attuale, lista movimenti)

Un progetto e i suoi dati sono variabili: è possibile (probabile) che vengano modificati/eliminati file al suo interno.

IMPORTANTE: Di un progetto non viene salvato in memoria permanente l'indirizzo IP Multicast e la porta della chat. Questi vengono assegnati in fase di inizializzazione del server (se si tratta di progetti già esistenti) oppure assegnati dinamicamente al momento della creazione dell'istanza di un nuovo progetto. Quando un progetto viene cancellato indirizzo e porta vengono liberati per essere riutilizzati in un nuovo progetto. (in seguito sarà spiegato chi ha questo compito)

Per ogni utente è associato un file “username.json” dove sono salvate le informazioni suddette (nome utente, hash della password, salt utilizzato). Gli utenti sono immutabili: una volta effettuata la registrazione, viene salvato in memoria il file legato all’utente e non sarà possibile effettuare alcuna modifica.

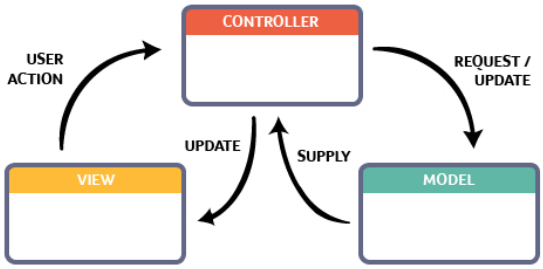
Dopo la fase di inizializzazione, il server sarà pronto a eseguire i comandi richiesti dai client e, se necessario, modificare la struttura dei dati. Per garantire la corrispondenza tra file salvati nello storage e contenuto del database, ogni qualvolta viene apportata una modifica ad un dato, questa modifica viene perpetuata anche sul file per mezzo di sovrascrittura. Operazioni che comportano la modifica dei file salvati in memoria sono quelle del tipo *createProject*, *addMember*, *addCard*, *moveCard*, *etc*…

Quando un progetto viene cancellato, è possibile creare un nuovo progetto con lo stesso nome.

Client

Per quanto riguarda il client, è stata sviluppata una semplice interfaccia grafica per interagire in modo più diretto con l’utilizzatore finale dell’applicativo.

Per una migliore gestione del progetto si è deciso di sfruttare il pattern di programmazione MVC, grazie al quale si è riuscito a mantenere distaccati, quanto più possibile, la “logica” dell’applicativo e la parte grafica. Questo permetterebbe, con un piccolo effort aggiuntivo, di sviluppare una versione di questo progetto interamente a linea di comando, sostituendo la View con la command line e eliminando dal Controller tutte le imposizioni grafiche.



La View dell’applicativo è realizzata utilizzando le API AWT (Abstract Window Toolkit) e il framework “nativo” di Java, Swing.

Sono presenti 3 controller:

1. Authorization controller: si occupa di gestire gli eventi legati alla registrazione ed al login dell’utente
2. Logged controller: gestisce gli eventi legati alle operazioni effettuate da un utente registrato e attualmente online (tutto il resto)
3. Frame controller: gestisce solamente l’evento legato alla chiusura della finestra

I controller catturano le azioni effettuate su caselle di testo e bottoni, recuperano tutti i parametri necessari e chiamano la funzionalità voluta nella logica dell’applicativo (model). Un risultato affermativo permette al controller di aggiornare la vista; nel caso contrario verrà lanciata una specifica eccezione e l’utente vedrà comparire messaggi di errore che possono aiutarlo a risolvere il problema.

Serializzazione

Tutte le risorse che viaggiano sulla connessione TCP tra client e server e tutti i file salvati nella memoria permanente del server sono dati strutturati secondo il formato standard JSON. Le procedure di serializzazione/de-serializzazione sono effettuate mediante l'ausilio della libreria esterna Jackson. La versione di tale libreria verrà indicata in fondo al documento.

Schema generale dei threads

Lato server

- **ServerMain**: classe main che istanzia gli altri thread e si preoccupa della fase di inizializzazione dello stato del server
- **RMITask**: thread che si preoccupa di pubblicare gli stub di registrazione RMI e Callback RMI sul Registry
- **SelectionTask**: server che adempie i compiti di accettare nuove connessioni TCP e comunicare con i client

Come già anticipato, l'implementazione del server è stata realizzata mediante NIO Selector e SocketChannels. Essendo un server sequenziale, la questione della concorrenza e dell'integrità dei dati diviene una problematica molto meno pesante. Infatti, tutto ciò che riguarda i progetti non è in alcun modo messo a rischio da eventuali operazioni concorrenti.

Gli unici dati condivisi tra più entità sono quelli riguardanti gli utenti. Questo perché la fase di registrazione è affidata al meccanismo RMI mentre la fase di login è dovuta alla comunicazione TCP esplicita. Per tale ragione, la struttura degli utenti è implementata con una collezione ConcurrentHashMap<String, User>. Questa struttura garantisce l'accesso concorrente da diversi threads, il tutto senza bloccarla interamente, come verrebbe fatto con una classica HashMap sincronizzata.

NOTA: Nonostante le strutture dati dei progetti e quella relativa allo stato attuale degli utenti (online, offline) non siano soggette ad accessi concorrenti, sono state implementate anch'esse con una ConcurrentHashMap.

Lato client

- **ClientMain**: classe main che istanzia model, views e controllers.
- **ChatReaderControllerTask**: thread che si occupa di mettersi in ricezione dei datagrammi UDP su una specifica chat (MulticastSocket). Quando arriva un datagramma, si occupa anche di aggiornare la View della chat a cui fa riferimento. Viene attivato da LoggedController quando l'utente richiede di visualizzare la chat in questione e rimane attivo per tutta la durata della sessione dell'utente. La sessione può essere interrotta facendo il logout oppure chiudendo bruscamente l'applicativo. In caso il progetto, a cui la chat fa riferimento, venga cancellato, il thread viene avvisato dal server con un datagramma UDP e termina spontaneamente.

Qui non si ha alcun problema legato alla concorrenza poiché non viene condivisa alcuna struttura dati tra i threads dell'applicativo.

Classi definite

- Client
  - Controller
    - AuthController
    - ChatReadedControllerTask
    - FrameController
    - LoggedController
  - Model
    - RMI
      - RMICallbackNotify
      - RMICallbackNotifyImpl
    - **ClientModel**
  - UI
    - loggedPanels: Panels legati ad un utente online
    - AuthUI
    - HostsCardsContainer
    - LoggedUI
    - WorthFrame
- Data
  - **Card**
  - **CardNoMovs**
  - CardStatus: enum {TODO, INPROGRESS, TOBEREVISED, DONE}
  - **Movement**
  - **Project**
  - User
  - UserStatus: enum {OFFLINE, ONLINE}
- Exceptions
  - Tutte quelle necessarie allo sviluppo dell'applicativo
- Protocol
  - **CommunicationProtocol**
  - RequestMessage
  - ResponseMessage
  - **UDPMessage**
- Server
  - RMI
    - RMICallbackService
    - RMICallbackServiceImpl
    - RMIRegistrationService
    - RMIRegistrationServiceImpl
  - Attachment
  - **PersistentData**
  - Registration: interfaccia che definisce l'operazione di registrazione offerta
  - RMITask
  - SelectionTask
  - TCPOperations: interfaccia che definisce le altre operazioni offerte
- Utils
  - **MulticastAddressManager**
  - MyCardLayout: gestisce la grandezza dei panel nel layout Swing CardLayout
  - MyObjectMapper: ObjectMapper di Jackson personalizzato

    - PasswordManager
    - PasswordManagerImpl
    - **PortManager**
    - UIMessages: insieme dei messaggi che possono essere visualizzati nella UI
    - Utils: funzioni di utilità di diversa natura
- ClientMain e ServerMain

ClientModel

Logica dell'applicativo a livello client. Si occupa di comunicare con il server inviando le richieste e ricevendo le risposte sulla connessione instaurata. Contiene la struttura dati userStatus, che viene aggiornata dal server con RMI Callback e le informazioni legate alle chat dei progetti (indirizzo e numero di porta delle chat già aperte, le MulticastSocket delle chat già aperte e il thread pool che si occupa di ascoltare quelle determinate MulticastSocket.

Card

Rappresenta una card di un progetto. I suoi attributi sono nome, descrizione, stato attuale e lista dei movimenti

CardNoMovs

Interfaccia che definisce il comportamento di una Card che non possiede la lista dei movimenti. Utilizzata per serializzare una Card senza i movimenti quando l'utente richiede il servizio *showCard*

Movement

Movimento di una card. È una tripla
<stato di partenza, stato di arrivo, dato temporale sulla sua creazione>

Project

Progetto di WORTH. Possiede un nome, una lista di membri (solo nomi utenti), data di creazione, indirizzo IP Multicast della chat, porta della chat, liste degli stati delle card (per ogni stato sono memorizzati i nomi delle card che ne fanno parte), lista di tutte le card (oggetti Card).

CommunicationProtocol

Fondamenta del progetto. Qui sono descritti tutti i comandi e gli status code che possono essere prodotti. Lo schema di funzionamento è il seguente:

1. Client prepara messaggio di richiesta, lo serializza e lo invia al server tramite la socket TCP
2. Server riceve il messaggio, lo deserializza ottentendo un RequestMessage
3. Analizza comando e argomenti
4. Chiama funzione primitiva di PermanentData
    a. Se l'operazione viene effettuata, il server preparerà un messaggio di risposta ResponseMessage con statusCode = SUCCESS e, se richiesto dal comando di richiesta, serializzerà il risultato ottenuto dalla funzione primitiva e lo inserirà in responseBody

    b. Se l'operazione lancia un eccezione X, questa viene catturata e viene generato uno statusCode = VALORE_ECCEZIONE_X
5. Server serializza il messaggio di risposta e lo invia al client tramite la socket TCP
6. Client riceve risposta, la deserializza ottenendo un ResponseMessage
    a. Se statusCode == SUCCESS vuol dire che l'operazione è andata a buon fine
    b. Se statusCode == VALORE_ECCEZIONE_X allora il client lancia l'eccezione X
7. Questa eccezione viene catturata dal controller. Quest'ultimo visualizzerà a schermo un banner con un messaggio di errore specifico per l'eccezione X.

UDPMessage

Struttura di un datagramma UDP. Contiene i campi

- Author: username di chi ha scritto il messaggio
- Message: testo del messaggio
- isFromSystem: campo booleano settato SOLAMENTE dal server per inviare messaggi sulla chat

PersistentData

Questa classe si occupa di realizzare le funzionalità di WORTH. È lei che possiede tutte le strutture dati su cui si basa il progetto. In particolare:

- Users: mappa concorrente dove l'insieme delle chiavi sono gli username degli utenti registrati, mentre users(k) è l'istanza del singolo utente k. Unica struttura dati ad essere acceduta concorrentemente nel sistema.
- Projects: mappa dove l'insieme delle chiavi sono i nomi dei progetti nel sistema, mentre projects(k) è l'istanza del singolo progetto k.
- userStatus: mappa dove l'insieme delle chiavi sono gli username degli utenti registrati, mentre userStatus(k) è lo stato attuale dell'utente k (online, offline).

MulticastAddressManager

Classe astratta che gestisce gli indirizzi Multicast. Genera indirizzi Multicast in sequenza da 239.0.0.0 a 239.255.255.255 (indirizzi Multicast locali). Utilizzata per attribuire ad un progetto un indirizzo Multicast. Non mantiene alcuna informazione in memoria persistente, gli indirizzi sono riassegnati dall'inizio ogni qual volta il server viene riavviato.

PortManager

Classe astratta che gestisce numeri di porta. Genera numeri di porta in sequenza da 30000 a 65535 (alcune porte registrate + porte effimere). Utilizzata per attribuire ad un progetto un numero di porta. Non mantiene alcuna informazione in memoria persistente, i numeri di porta sono riassegnati dall'inizio ogni qual volta il server viene riavviato.

Istruzioni di compilazione

Assicurarsi di essere dentro la directory principale di WORTH (dove sono src/, lib/, …).

Sistemi Linux/macOS

Da linea di comando

- Crea un file dove sono elencati tutti i file da compilare con javac
  `find . -name "*.java" > ./paths.txt`
- Genera bytecode
  `javac -d "bin" -cp lib/jackson-annotations-2.12.1.jar:lib/jackson-core-2.12.1.jar:lib/jackson-databind-2.12.1.jar:lib/jackson-datatype-jsr310-2.12.1.jar @paths.txt`
- Esegui server
  `java -cp lib/jackson-annotations-2.12.1.jar:lib/jackson-core-2.12.1.jar:lib/jackson-databind-2.12.1.jar:lib/jackson-datatype-jsr310-2.12.1.jar:bin worth.ServerMain`
- Esegui client
  `java -cp lib/jackson-annotations-2.12.1.jar:lib/jackson-core-2.12.1.jar:lib/jackson-databind-2.12.1.jar:lib/jackson-datatype-jsr310-2.12.1.jar:bin worth.ClientMain`

In alternativa, eseguendo in successione sul terminale

- `chmod +x *.sh`
- `./build.sh`
- `./execServer.sh`
- `./execClient.sh`

Sistema Windows

Da linea di comando

- Crea un file dove sono elencati tutti i file da compilare con javac
  `dir /s /B *.java > ./paths.txt`
- Genera bytecode
  `javac -d "bin" -cp lib/jackson-annotations-2.12.1.jar;lib/jackson-core-2.12.1.jar;lib/jackson-databind-2.12.1.jar;lib/jackson-datatype-jsr310-2.12.1.jar @paths.txt`
- Esegui server
  `java -cp lib/jackson-annotations-2.12.1.jar;lib/jackson-core-2.12.1.jar;lib/jackson-databind-2.12.1.jar;lib/jackson-datatype-jsr310-2.12.1.jar;bin worth.ServerMain`
- Esegui client
  `java -cp lib/jackson-annotations-2.12.1.jar;lib/jackson-core-2.12.1.jar;lib/jackson-databind-2.12.1.jar;lib/jackson-datatype-jsr310-2.12.1.jar;bin worth.ClientMain`

Informazioni utili

- A causa dell'utilizzo del gestore delle password, è necessario compilare ed eseguire il programma con una versione delle JDK >= 9. Nel mio caso, ho utilizzato la OpenJDK 15.
- Il codice è stato testato con successo su Windows 10 20H1 e macOS Big Sur 11.1
- Ho notato una differenza di comportamento tra Windows e macOS nel momento in cui viene chiusa una SocketChannel. In particolare mi sono concentrato sulla funzione socket.read(buffer).
    - Su macOS termina ritornando -1 (secondo la documentazione, EOS);
    - Su Windows lancia una SocketException.
- Versione di Jackson utilizzata: 2.12.1 (ultima stabile disponibile).
- Il codice sulla UI è meno commentato del resto semplicemente perché alle volte bisogna istanziare oggetti su oggetti e disporli nei modi più svariati per ottenere un risultato grafico soddisfacente.