Networking Lab. Project Report - Word Quizzle

Sergio G. Attanzio, 491107

March 2020

1 Introduzione

Questa relazione sul progetto di Laboratorio di Reti "Word Quizzle" servirà a chiarire scelte implementative e architetturali prese per realizzare l'intero progetto. La relazione è suddivisa in sezioni e sotto-sezioni.

2 Architettura generale

Il progetto è formato da un server locale e da un client con una interfaccia a linea di comando. Per la realizzazione dei protocolli di comunicazione ho tenuto in considerazione tutte le specifiche di implementazione richieste e fornite con il documento in pdf che detta le specifiche del progetto.

2.1 Client

Il client è una classe ed è un processo multithreaded formato da un thread principale e un thread gestore UDP.

- Il thread gestore UDP è avviato dopo il login e si occupa di gestire la recezione dei messaggi UDP del server che servono per segnalare al client che c'è una sfida da un suo amico. Quando è presente una sfida pendente il client può eseguire i comandi yes/no. La gestione di questa informazione condivisa tra i due thread viene fatta attraverso un monitor lock che viene acquisito da questo thread per avvisare di una sfida in attesa settando un flag e dal thread principale per resettare il flag.
- Il thread principale si occupa invece di gestire tutto il resto. All'avvio si trova nello stato anonimo con cui è possibile fare poche cose tra cui registrarsi e accedere. Una volta acceduto si passa allo stato di logged e fornisce il resto delle funzionalità richieste nelle specifiche.

All'avvio il client mostrerà la guida ai comandi (interfaccia testuale) per il client "anonimo". Una volta registrato un utente è possibile usare le credenziali della registrazione per accedere e passare allo stato "logged".

Per la comunicazione tra un client loggato e il server ho sviluppato un protocollo specifico basato su TCP che invia solo l'essenziale dal client al server e viceversa dal server al client. Nella fase di invio il client invia un messaggio che comprende una richiesta e, in base alle circostanze, anche un payload.

La richiesta è un byte con uno dei seguenti valori:

- 0 - accetta sfida
- 1 - rifiuta sfida
- 2 - login
- 3 - logout
- 4 - aggiungi amico
- 5 - lista amici
- 6 - lancia una sfida
- 7 - mostra punteggio
- 8 - mostra classifica
- 9 - controlla se sfida è finita
- 10 - richiedi parola da tradurre
- 11 - invia parola tradotta
- 12 - chiedi di terminare la partita

Le operazioni fornite dal client sono principalmente quelle richieste dalle specifiche e sono associate al byte di richiesta mostrato sopra.

Il client si occupa anche di mostrare un'interfaccia testuale di sfida in cui viene mostrata, dopo un iniziale tempo di ricezione settaggi dal server e conto alla rovescia, una parola da tradurre alla volta e verrà acquisito l'input da tastiera. La lunghezza delle parole può essere impostata sul server che la comunicherà insieme al numero di parole da tradurre e durata del match.

Con l'invio della richiesta di ottenere una parola e l'invio della traduzione viene testato lo stato della partita. Quindi finchè l'utente non interagisce non saprà a che punto è la richiesta (è possibile implementare questa cosa inviando periodicamente una richiesta per ottenere lo stato della partita ma non mi sembrava il caso). Il problema sarà solo dal lato dell'utente che non interagisce in quando allo scadere del tempo l'utente che interagisce tornerà libero e potrà sfidare qualcun'altro.

Nella classe Client è inoltre presente una classe di utilità e alcuni metodi statici con visibilità sull'intero package usati anche dal Server.

2.2 Server

Il server è una classe ed è un processo multithreaded formato da un thread principale, un thread handler per ogni possibile sfida e un server RMI.

- i thread handler vengono avviati quando un utente A invia con successo una richiesta di sfida ad un altro utente B. Si occupa principalmente di controllare che la sfida venga accettata in tempo, altrimenti cancella la possibile sfida avvisando A.
- il thread principale, gestisce le richieste che gli arrivano tramite il multiplexing dei canali mediante NIO. Praticamente si occupa di tutto, risponde il più velocemente possibile e si mette in attesa di una prossima richiesta.
- il server RMI si occupa esclusivamente della registrazione di nuovi utenti al servizio

Ho usato alcune strutture dati per gestire alcune collezioni di dati vitali per il funzionamento del progetto.

- userDB: rappresenta il database utenti che viene recuperato dal file json che memorizza le informazioni sugli utenti anche quando il server è chiuso. Ho scelto di implementarlo con una HashMap.
- socketToUer: serve a mappare gli utenti connessi a partire dal channel di connessione creato al momento del login dell'utente. Grazie a questa struttura dati posso facilmente riconoscere di che utente si sta parlando sapendo il channel di comunicazione creato per l'utente in questione. Ho scelto di implementarlo con una HashMap.
- locks: serve a mappare le lock per l'accesso alle sfide a partire dallo username, quindi vengono recuperate le lock e monitorate tramite blocchi synchronized per accedere a delle sezioni critiche. Ho scelto di implementarlo con una HashMap.
- possibleChallenge: si tratta di una collezione di sfide in cui sono inserite anche le nuove sfide che potrebbero essere rifiutate o scadranno. Viene aggiunta una nuova sfida quando un utente ne sfida un altro e in quel momento i due utenti, partecipando a questa sfida non potranno avviarne di altre o essere invitati a partecipare ad altre finchè quella attuale non verrà cancellata perchè terminata/rifiutata/annullata/scaduta. Questa classe è un wrap a una HashMap possibleChallenge e a due classi, Challenge che memorizza lo status della sfida e StatusChallenge che rappresenta lo stato della sfida di un singolo sfidante.

Il Server quando avviato recupera o crea, se non esiste, il file json che memorizza gli utenti. Avvia il server RMI per la registrazione e avvia un mainLoop.

Il mainLoop è il core del server: crea un selettore a cui registra i canali non bloccanti che deve monitorare come le connessioni aperte coi client e il canale

per il protocollo UDP e cicla sui canali in attesa che ci sia qualcosa da fare. Per questo tipo di struttura è stato necessario che il server svolgesse al più presto la richiesta fatta dal client e che, se la richiesta prevedeva delle attese perchè ad esempio il client deve aspettare l'input dell'utente, le richieste del client vengano divise in più richieste.

Ad esempio, una volta accettata una sfida e mandati i settaggi, il server si dedica a nuove richieste da parte, possibilmente, di altri client diversi. Quando uno dei due client che partecipa alla sfida di prima chiede una parola o invia una traduzione deve fare una nuova richiesta che verrà gestita coerentemente dal server che troverà di che partita si tratta, preleva una nuova parola e la invia oppure verifica che la traduzione sia corretta e aggiorna il punteggio. In una sezione a seguire verrà analizzata più in dettaglio la gestione di una sfida.

Questi sono i codici acknowledge TCP da Server a Client. In certi contesti questi stessi codici hanno assunto altri significati per comodità:

- 0 - ok generico (dipende dal contesto)
- 1 - utente già connesso (login)
- 2 - utente non registrato (login, add, userscore)
- 3 - password sbagliata (login)
- 4 - già amici (add)
- 5 - azione rivolta a se stessi (add, challenge)
- 6 - non amici (challenge)
- 7 - amico offline (challenge)
- 8 - amico rifiuta la partita
- 9 - analogo a 0 generico (dipende dal contesto)
- 33 - partita non ancora terminata
- 42 - partita annullata
- -1 - errore generico (dipende dal contesto), ad esempio ack = -1 in challenge.isFinished()

Le operazioni fornite dal server sono principalmente quelle richieste dalle specifiche.

2.2.1 Gestione sfida

La gestione di una sfida è stata una delle parti più complicate da fare perchè l'ho implementata in maniera asincrona.

Prima di tutto bisognava verificare che l'amico da sfidare fosse online. Se lo è viene creata una possibleChallenge che coinvolge i due utenti e li "impegna" in modo che non possano fare o ricevere altre richieste in quel momento. A quel punto viene inviato un messaggio UDP al client sfidato che potrà accettare o rifiutare e, se passa più tempo del previsto, far scadere la sfida.

Il client sfidante rimane in attesa che l'amico sfidato accetti o rifiuti finchè non passa il tempo previsto a far scadere la sfida che rende di nuovo non impegnati i due client coinvolti; a questo scopo viene avviato un handlerPossibleChallenger Thread che si occupa, in caso di scadenza, di cancellare la sfida quando l'altro non risponde in tempo e di avvisare il client sfidante che è scaduta. Ho preferito avviare un thread che controllasse lo scorrere del tempo piuttosto che lasciar gestire al client in autonomia questa cosa più che altro per permettere al server di decidere se la sfida è scaduta o meno.

Come precedentemente scritto le parole vengono inviate con messaggi separati dalla richiesta di sfida. Questo permette al server di essere libero di svolgere altri compiti (come gestire altri utenti e sfide) mentre l'utente alla tastiera pensa alle parole da tradurre.

Il client viene avvisato che il tempo per la sfida è finito quando chiede una parola o invia una traduzione al server, quindi finchè non interagisce rimarrà fermo in partita. Inviare la traduzione permette al server di avvisare il client se le parole da tradurre fossero finite. Quando finite, il client lo segnala all'utente tramite un messaggio sulla console e ogni secondo chiede al server se anche l'altro ha finito o se il tempo della partita è finito. Questo permette di interrompere partite lunghe se entrambi hanno finito senza aspettare la fine del tempo. Nel caso solo uno dei due player interagisse con l'interfaccia di gioco quello che lo fa riceverebbe il risultato della partita e non sarebbe più impegnato. Ciò vuol dire che può essere coinvolto in altre sfide, al contrario del player che non interagendo non riceve il messaggio che la sfida è finita poichè è finito il tempo.

Un'altra questione da risolvere è stata quella di gestire il logout non previsto (chiusura del client ad esempio) di uno o di entrambi gli utenti quando coinvolti in una partita. Questo perchè in caso di logout la partita diventa annullata, non valida e a quel punto va cancellata senza che vengano apportate modifiche al punteggio dell'utente. La cancellazione vera e propria della sfida lato sfidante o lato amico avverrà quando questi giocatori vengono coinvolti in una possibile sfida, sia da sfidanti che da amici sfidati. Fino a quel momento quindi la sfida (intesa come una delle due metà, una StatusChallenge) rimane annullata ma pronta per essere rimossa.

2.3 User

Gli oggetti User rappresentano gli utenti registrati sul sito. Gli utenti vengono generati registrandosi tramite un server RMI. Le loro informazioni vengono man-

tenute persistenti tramite la memorizzazione delle info su un file DBUtenti.json.

Mantengono al loro interno informazioni utili alla loro gestione come:

- username: è il nome univoco dell'utente scelto al momento della registrazione. Il nome non deve essere più lungo di 16 caratteri.
- password: è la chiave d'accesso da usare per il login e scelta in fase di registrazione. Non deve essere più lunga di 8 caratteri.
- amici: una lista di persone che è possibile sfidare e che può essere facilmente ampliata
- online: indica se l'utente in questione ha fatto l'accesso.
- socketCh: rappresenta il canale di comunicazione tra il client dove l'utente è acceduto e il server. Per comodità si trova tra i campi utente, si potrebbe spostare su una map.
- UDPAddress: è l'indirizzo SocketAddress per le comunicazioni UDP dal Server al Client. Vale lo stesso discorso fatto per socketCh
- userscore: punteggio utente, ossia la somma dei punteggi partita ottenuti giocando

Ovviamente è possibile eseguire l'accesso di uno specifico utente su un solo client per volta.

2.4 Challenge

Le challenge sono le sfide tra coppie di utenti. All'inizio sono delle "possible challenges" poichè non sono ancora state accettate dall'utente sfidato. Quando un utente è coinvolto in una possibile sfida ha pochi secondi per accettarla e nel frattempo non può ricevere altre sfide da parte di altri utenti, né potrà sfidarne lui a sua volta in quanto gli utenti coinvolti sono considerati impegnati in questa possibile sfida.

Una sfida può essere:

- accettata: l'utente sfidato accetta di giocare, verrà dunque preparato il match. Entrambi risulteranno impegnati in una sfida.
- rifiutata: l'utente sfidato ha rifiutato di giocare. Entrambi gli utenti tornano non impegnati in una possibile sfida.
- scaduta: l'utente sfidato ha ignorato la richiesta di sfida o ha risposto troppo tardi, entrambi gli utenti tornano non impegnati.
- annullata: uno dei due utenti impegnati nella sfida abbandona la partita che viene quindi annullata. Gli utenti sono da subito disponibili per una nuova sfida anche se tecnicamente torneranno liberi solo quando verranno coinvolti in una nuova sfida.

Una challenge accettata è composta da due parti parallele, identificate nel codice come StatusChallenge che fanno capo ognuna a uno dei due player coinvolti nella sfida. Nelle StatuChallenge viene gestito il punteggio partita e le parole da inviare al giocatore. Tendenzialmente, quando nel codice si fa riferimento a challenger si parla di chi ha avviato la sfida e con friend si parla di chi è stato sfidato. Solitamnte quindi le coppie sono client-challenger, client-friend.

Alle challenge si accede tramite una classe che fa da interfaccia e che le colleziona, la classe PossibleChallenges. Per gestire l'accesso a queste risorse condivise bisogna entrare in possesso del monitor ossia di un Object raggiungibile mediante il nome dei due player coinvolti tramite una map.

2.5 Database JSON

Il database per mantenere persistenti le informazioni sugli utenti e i loro dettagli consiste di un file json che viene creato/recuperato a inizio partita e viene scritto tutte le volte che ci sono modifiche agli utenti. Al livello più esterno è formato da coppie Username - User. Viene mappato in una hashmap durante l'esecuzione.

2.6 Translator

Traslator è una classe del progetto. I Translator sono dei task Runnable che permettono di ottenere la traduzione delle parole scelte per la partita da giocare. Fanno una richiesta HTTP GET al sito delle traduzioni, ottengono il JSON che rappresenta la risposta del server delle traduzioni ed estraggono la parola tradotta. Si può migliorare il videogioco ottenendo tutte le possibili traduzioni a una determinata parola e quindi testare se è uguale a una delle parole valide come traduzione.

Vengono eseguiti da un ThreadPool Executor così da parallelizzare l'ottenimento delle parole tradotte. Ho usato delle BlockingQueue in modo che l'inserimento in una lista di parole tradotte fosse thread safe.

La classe contiene i metodi che servono per ottenere le parole in italiano e poi quelle in inglese. Le parole italiane, scelte causalmente, provengono da un dizionario che è un semplice foglio di testo in cui a ogni riga è presente un termine italiano comune. Possono essere aggiunte altre parole inserendole a capo, una per riga. Inoltre è possibile settare il tempo massimo d'attesa per ottenere le traduzioni che se scade invalida la partita.

2.7 Codici richieste UDP in entrambi i versi

Il primo byte identifica il tipo di messaggio.

- 0 - rende noto l'indirizzo client al server
- 1 - server invia richiesta di sfida all'amico

3 Guida all'utilizzo

Per lo sviluppo del progetto ho usato l'IDE Eclipse 9.19 e JavaSE-1.8. Ho inoltre utilizzato la libreria esterna Gson-2.8.6.jar.

Il progetto è stato esportato sotto forma di JAR e può essere importato su eclipse creando un nuovo progetto java vuoto e poi scegliendo l'opzione Import... In ogni caso il progetto è composto da 7 file .java (Client, PossibleChallenges, Server, SignUpServerRMI, Translator, User, UserSignUpInt), un dizionario di parole italiane "dizionario.txt" scritte una per ogni riga e fa uso della libreria Gson per la gestione dei Json.

Nella cartella in cui si trovano le cartelle .setting, bin e src va messo il dizionario; inoltre in questa cartella verrà creato il file json DBUtenti che può essere modificato e/o sostituito a mano (ovviamente senza corrompere la struttura del json).

Il progetto è composto da due programmi eseguibili, Server e Client.

Il server va eseguito una sola volta (altri processi server finchè ce n'è uno in esecuzione lanceranno eccezioni per porta occupata) lanciando la classe Server.java tramite Run As > Java Application (tasto destro sul file nel progetto esportato su eclipse). Non è previsto alcun argomento in ingresso.

I client si eseguono come si esegue il server e ce ne possono essere diversi in esecuzione simultaneamente. Per comodità è conveniente aprire più console su eclipse, 1 per ogni processo, e conviene bloccare (Pin Console) poichè quando ci sono stampe la console mostra l'ultima console che ha ricevuto un output.