Relazione sul Progetto di Reti di Calcolatori e Laboratorio

A.A. 2021/2022

Luca De Paulis

10 gennaio 2022

1 Introduzione e informazioni generali

Il progetto di Reti di Calcolatori e Laboratorio dell'Anno Accademico 2021/2022 consiste nella realizzazione di un Social Network chiamato **Winsome**, la cui particolarità consiste nell'elargire una criptovaluta, chiamata **Wincoin**, agli utenti in base alla popolarità dei loro post.

Il Social Network è basato su un'architettura di tipo *client-server*: le due parti comunicano attraverso la rete grazie ad un'API, che permette al client di interfacciarsi con il server.

Il progetto è stato realizzato in Java (versione 17) ed è stato sviluppato e testato in ambiente Linux.

2 Compilazione ed esecuzione

Il progetto è interamente compilabile ed eseguibile attraverso l'uso dei comandi `javac` e `java`; tuttavia, per semplificare il processo, è stato incluso un `Makefile` contenente alcune semplici regole.

`$ make` La regola di default compila il progetto: in particolare crea le directory necessarie al corretto funzionamento degli applicativi, compila il codice sorgente e li comprime in file `.jar`.

`$ make run-default-server` Questa regola, se lanciata dopo la compilazione, esegue il server usando come parametri (ovvero come path del file di configurazione e della directory di logging) i valori di default.

`$ make run-default-client` Analogamente, esegue il client usando i parametri di default (in questo caso solamente il path del file di configurazione).

`$ make clean` Rimuove i file `.class`, i file `.jar` e i file di log del server contenuti nella directory `logs/`.

Per eseguire il client o il server passando dei parametri a riga di comando si possono usare gli script Bash `run-server.sh` e `run-client.sh`, entrambi contenuti nella directory `scripts/`. Per far ciò è necessario innanzitutto renderli eseguibili, tramite ad esempio `\$ chmod a+x scripts/run-server.sh scripts/run-client.sh`.

A questo punto il server può essere invocato con le seguenti opzioni:

```
$ ./scripts/run-server.sh [--help] [--config=<CONFIG_PATH>] [--logs=<LOGS_PATH>]
```

dove `<CONFIG_PATH>` è il path verso il file di configurazione (che di default si trova in `configs/server-config.yaml`), mentre `<LOGS_PATH>` è il path della directory in cui verrà memorizzato il file di log (default: `logs/`).

La sintassi per eseguire il client è la seguente:

```
$ ./scripts/run-client.sh [--help] [--config=<CONFIG_PATH>]
```

dove `<CONFIG_PATH>` è il path del file di configurazione (di default si usa il file `configs/client-config.yaml`).

L'unica dipendenza del progetto è la libreria GSON (`gson-2.8.6.jar`), che si trova nella directory `lib/`.

3 Architettura generale del progetto

Il progetto è diviso in tre moduli principali:

- il **server**, implementato nel package `winsome.server` e avente come *entry point* la classe `winsome.server.WinsomeServerMain`: esso viene raccolto nel file `bin/winsome-server.jar`;
- il **client**, implementato nel package `winsome.client` e avente come *entry point* la classe `winsome.client.WinsomeClientMain`: viene compresso nel file `bin/winsome-client.jar`
- le **API**, implementate nel package `winsome.api`: insieme alle funzionalità di *utility* del package `winsome.utils` le API vengono compresse nel file `lib/winsome-api.jar`.

La scelta di isolare le API dall'applicativo client ha lo scopo di separare il più possibile i vari moduli: così facendo si lascia aperta la possibilità di creare un client diverso che riusi le API fornite per interfacciarsi con il server.

Inoltre, dato che le API sono compilate in una libreria in formato `.jar`, possono essere diffuse e utilizzate in modo indipendente dal resto del codice del client e del server.

4 Il package `winsome.utils`

I tre moduli descritti nella Sezione 3 dipendono da alcune funzionalità messe a disposizione dal package di utilities:

- il file `winsome.utils.ConsoleColors` contiene i codici ASCII per usare i colori nei terminali: viene usato soprattutto dal client nella propria Command Line Interface;
- il package `winsome.utils.cryptography` contiene la classe `Hash`, che permette l'hashing di stringhe (utile per inviare e memorizzare le password), e l'eccezione `FailedHashException` per indicare il fallimento di un tentativo di hashing;
- il package `winsome.utils.configs` contiene le classi `AbstractConfig` e `ConfigEntry`: esse consentono di parsare file nel formato descritto nella Sezione 5.

5 I file di configurazione

Sia il client che il server necessitano di un file di configurazione per inizializzare i vari parametri. La sintassi dei due file è la stessa, e corrisponde alla sintassi base del linguaggio di markup YAML:

```
# comments and empty lines are allowed
key: value # comments after values are allowed
```

Due esempi di file di configurazione sono presenti nella directory `configs/` e di default, a meno di non specificare altrimenti negli argomenti a riga di comando del client/server, tali file sono usati come file di configurazione.

I campi dei file di configurazione devono essere tutti presenti una e una sola volta: in caso contrario il sistema segnala l'errore.

5.1 File di configurazione del server

Il file di configurazione del server deve avere i seguenti campi:

`tcp-port` Un intero non negativo che rappresenta la porta usata dal socket TCP.

`udp-port` Un intero non negativo che rappresenta la porta usata dal socket UDP.

`multicast-addr` L'indirizzo di multicast sul quale i client ricevono le notifiche del calcolo delle ricompense.

`multicast-port` La porta sulla quale i client ricevono le notifiche in multicast (è un intero non negativo).

`registry-name` Nome del registry RMI creato dal server.

`registry-port` Porta del registry RMI.

`reward-interval` L'intervallo di tempo in secondi tra due esecuzioni dell'algoritmo di calcolo delle ricompense.

`reward-percentage` La percentuale (come numero *floating point* compreso tra 0 e 100) di ricompensa che va all'autore del post.

`persistence-dir` La directory che contiene lo stato corrente del server. Tale directory può essere vuota all'avvio del server, ma deve esistere.

`persistence-interval` L'intervallo di tempo in secondi tra due esecuzioni dell'algoritmo di persistenza dei dati.

`keep-alive` Il tempo in secondi per cui i *worker thread* del thread pool possono rimanere attivi senza essere impegnati in alcun task.

`min-threads` Il minimo numero di *worker thread* sempre attivi.

`max-thread` Il massimo numero di *worker thread* attivi contemporaneamente.

`pool-timeout` Il tempo di timeout (in millisecondi) prima di chiudere forzatamente il *thread pool* alla chiusura del server.

5.2 File di configurazione del client

Il file di configurazione del client deve avere i seguenti campi:

`server-addr` L'indirizzo su cui si trova il server.

`tcp-port` Un intero non negativo che rappresenta la porta del server.

`registry-name` Il nome del registry RMI del server.

`registry-port` La porta del registry RMI.

`socket-timeout` Il tempo di timeout (in millisecondi) per il socket di comunicazione con il server.

6 Il package `winsome.client`

Il package `winsome.client` contiene la Command Line Interface sfruttata da un utente del Social Network per interagire con gli altri utenti. In particolare questo package contiene le seguenti classi:

- la classe `ClientConfig` eredita dalla classe astratta `winsome.utils.configs.AbstractConfig` e permette di effettuare il parsing del file di configurazione del client, descritto sopra;
- la `enum` `Command`, insieme all'eccezione `exceptions.UnknownCommandException`, si occupa di descrivere i comandi della CLI; implementa inoltre alcuni metodi di utility che permettono di ottenere il comando a partire dalla stringa digitata dal client e di ottenere degli help messages per ogni comando;
- la classe `WinsomeClientMain` è l'*entry point* del client: si occupa di leggere il file di configurazione, istanziare le API per collegarsi al server e infine interpretare i comandi scritti dall'utente per fare richieste alle API e restituirne i risultati.

6.1 La Command Line Interface del client

Per interagire con il Social Network l'utente inserisce in modo interattivo dei comandi nel prompt fornito dall'applicativo client. I possibili comandi sono i seguenti:

`help [<command>]` Stampa il messaggio di aiuto: se viene indicato un particolare comando stampa il messaggio di aiuto di quel comando, altrimenti stampa il messaggio di aiuto generale.

`quit` Chiude il client.

`register <username> <password> <tags>` Permette la registrazione dell'utente di nome `<username>`, con password `<password>` e con tag di interesse `<tags>`. Non vi sono restrizioni sul nome, se non che deve essere unico; la password non deve essere vuota; i tag sono in numero compreso tra 1 e 5, devono contenere solo lettere minuscole e devono essere separati da uno spazio. Inoltre, non si possono registrare nuovi utenti se si è già loggati in qualche profilo: bisogna prima eseguire un'operazione di logout.

`login <username> <password>` Permette il login in un utente precedentemente registrato: può dare errore se l'utente non esiste, la password è sbagliata oppure il client è già loggato.

`logout` Permette il logout: come i successivi comandi, può fallire se nessun utente è loggato.

`list users` Permette ad un utente loggato di vedere gli altri utenti del Social Network con cui condivide degli interessi (rappresentati dai tag).

`list followers` Mostra ad un utente loggato la lista dei suoi *followers*, ovvero degli utenti che lo seguono.

`list following` Mostra ad un utente loggato la lista degli utenti seguiti.

`follow <username>` Permette all'utente loggato di seguire l'utente che ha come nome `<username>`: tale operazione va a buon fine se e solo se l'utente loggato e `<username>` hanno tag in comune. In tal caso, da quel momento in poi l'utente loggato può vedere i post di `<username>` nel suo *feed* e può interagire con essi (ad esempio votandoli, effettuando un *rewin* oppure aggiungendo un commento).

`unfollow <username>` Consente all'utente corrente di smettere di seguire un utente.

`blog [<username>]` Permette all'utente loggato di vedere i post pubblicati da un utente del Social Network: in particolare se nessun nome viene indicato l'utente vede i suoi post. In ogni caso, un utente può vedere solo i post pubblicati da utenti con cui ha interessi in comune, in modo da decidere se eventualmente seguirli.

`post <title> <contents>` Consente la creazione di un nuovo post. In particolare il titolo non può superare i 50 caratteri, mentre il contenuto non può superare i 500 caratteri; inoltre se sono presenti spazi il titolo e/o il contenuto del post vanno racchiusi tra virgolette. Alla fine dell'operazione il server restituisce l'identificativo del post appena creato.

`show feed` Mostra i post nel *feed* dell'utente corrente, ovvero mostra tutti i post pubblicati dagli utenti seguiti dal client.

`show post <idPost>` Mostra il post con identificativo `<idPost>`: in particolare mostra anche il contenuto del post, il numero di upvote/downvote e i commenti. L'operazione si conclude con successo solo se l'utente loggato ha interessi in comune con il creatore del post (che sia l'autore originale o il *rewinner*).

`delete <idPost>` Se l'utente è il creatore del post, lo cancella, altrimenti restituisce un errore. In particolare se il post è un *rewin* cancella solo il post, mentre se il post è un post originale cancella tutti i *rewin* del post.

`rewin <idPost>` Sponsorizza il post con identificativo `<idPost>`, ovvero lo rende visibile a tutti gli utenti con interessi in comune all'utente corrente, e quindi anche a coloro che non possono interagire con l'autore originale. Tale operazione termina con successo solo se l'utente corrente segue il creatore del post identificato da `<idPost>`.

`rate <idPost> <vote>` Aggiunge un like oppure un dislike al post con identificativo `<idPost>`, a patto che l'utente corrente segua il creatore del post. Il campo `<vote>` può essere +1 oppure −1 per indicare rispettivamente un like e un dislike; se viene indicato un altro valore il server restituisce un errore.

`comment <idPost> <contents>` Aggiunge un commento al post `<idPost>`, a patto che l'utente corrente segua il creatore del post. Se il commento contiene spazi va racchiuso tra virgolette.

`wallet` Restituisce la storia delle transazioni relative all'utente corrente, insieme al totale in Wincoins.

`wallet btc` Restituisce il valore totale accumulato dall'utente convertito in BTC. Questa operazione può fallire se il servizio che fornisce il tasso di cambio Wincoins-BTC non è disponibile.

7 Il package `winsome.api`

Il package delle API implementa uno dei due endpoint della comunicazione client-server e pertanto contiene molti dei metodi e delle classi usate nella comunicazione.

- Il subpackage `winsome.api.codes` contiene due classi, chiamate `RequestCode` e `ResponseCode`, che indicano le possibili richieste che il client può inviare e le possibili risposte del server (inclusi gli errori). Dato che i messaggi inviati sono in forma di oggetti JSON (tramite la classe `JsonObject` fornita da GSON), le due classi forniscono anche dei metodi per aggiungere un campo "codice" ad un JSON e per ottere il codice salvato.
- Il subpackage `winsome.api.remote` contiene le interfacce `RemoteClient` e `RemoteServer` che consentono l'uso delle tecniche di *Remote Method Invocation* da parte delle API e del Server.

- Il subpackage `winsome.api.exceptions` contiene molte eccezioni, usate sia nel codice del server, sia per comunicare al client la risposta del server.
- Il subpackage `winsome.api.userstruct` contiene alcune strutture dati immutabili usate per racchiudere i dati inviati dal server, come i post (incapsulati nella classe `PostInfo`) e il portafoglio di un utente (descritto dalla classe `Wallet`).

La classe `WinsomeAPI` fornisce la logica delle API. Per utilizzarla bisogna

(1) istanziare un oggetto `WinsomeAPI`, fornendo i parametri necessari;

(2) invocare il metodo `connect()`, che si occupa di

  (a) stabilire la connessione TCP con il server,

  (b) immediatamente chiedere al server le coordinate per unirsi al gruppo di multicast su cui si ricevono gli update delle ricompense,

  (c) ottenere dal Registry l'istanza remota del server e creare la propria istanza remota.

A questo punto i metodi delle API possono essere invocati dal client per eseguire le varie operazioni concesse. Le API fanno alcuni *sanity check* sugli argomenti prima di inviare la richiesta al server, e in caso vi sia un errore evidente viene immediatamente sollevata un'eccezione.

Per comunicare con il server vengono usati i metodi del package `java.io`. Il protocollo esatto di comunicazione è descritto nel file `docs/protocol.md` e consiste nell'invio di `JsonObject` (adeguantamente convertiti in bytes) che contengono il codice della richiesta/risposta e eventuali argomenti.

Contemporaneamente, per ricevere i messaggi di update delle ricompense, le API mandano in esecuzione un thread pool contenente un singolo thread che esegue un'istanza della classe `WalletUpdatesWorker`: il metodo `call()` (ereditato dall'interfaccia `Callable`) di questa classe consiste nell'attendere i messaggi del server in un ciclo infinito, e, nel caso un client sia loggato, inserirli in una `BlockingQueue`.

Quando il client ne fa richiesta, il metodo `getServerMsg()` delle API sposta l'intero contenuto della `BlockingQueue` in una lista e la restituisce al client, che potrà controllare tutti i messaggi inviati dal server dall'ultima richiesta.

8 Il package `winsome.server`

L'ultimo package implementa l'intera logica del server. Il suo *entry point* è la classe `WinsomeServerMain`, che si occupa di

(1) leggere e fare il parsing degli eventuali argomenti a riga di comando, che sono descritti successivamente;

(2) inizializzare il logger, utilizzando come directory di destinazione per i file di log la directory `logs/` se nessuna scelta specifica è stata indicata tra gli argomenti a riga di comando;

(3) istanziare un oggetto `WinsomeServer`;

(4) inizializzare i dati del server e i vari socket tramite i metodi `init()` e `start()`;

(5) far partire il server su un thread dedicato, mentre il thread principale rimane in attesa di un input da parte dell'utente per terminare il server.

8.1 Architettura del server

Il server deve eseguire contemporaneamente 4 tipi di operazioni:

(1) accettare nuovi client oppure nuove richieste dai client già accettati,

(2) soddisfare tali richieste e inviare i risultati,

(3) persistere periodicamente i dati,

(4) calcolare periodicamente le ricompense dei post.

Per far ciò la classe `WinsomeServer` contiene al suo interno altre 3 classi:

- la classe `Worker`, che si occupa di eseguire le richieste;
- la classe `ServerPersistence`, che si occupa di salvare lo stato corrente sul disco;
- la classe `RewardsAlgorithm`, che calcola periodicamente le ricompense per i post.

Queste tre classi implementano le interfacce `Runnable` oppure `Callable<Void>` e vengono pertanto eseguite in parallelo al thread principale, che si occupa solamente di smistare le richieste. In particolare viene eseguito una sola istanza di `ServerPersistence` e di `RewardsAlgorithm`, mentre i `Worker` sono gestiti da un thread pool di dimensione variabile.

Nell'implementazione data è contemporaneamente attivo un ultimo thread, che è il thread della classe `WinsomeServerMain`, che però è in attesa dell'input dell'utente per terminare il server.

8.2 Gestione della concorrenza

Per gestire l'accesso concorrente alle strutture dati interne del server si sfruttano le classi e i metodi forniti dalla libreria `java.util.concurrent`, ed in particolare le classi `ConcurrentHashMap` e `ConcurrentLinkedQueue`.

Dato che la maggior parte delle operazioni compiute dal server consiste nel modificare un unico punto di un'unica struttura, oppure di iterare una sola struttura, le proprietà delle classi appena menzionate garantiscono una corretta sincronizzazione tra i thread. Nel caso in cui fosse necessario modificare più strutture contemporaneamente (come ad esempio nella registrazione di un nuovo utente, oppure nel rewin di un post) si sfruttano i monitor.

8.3 I/O del server

Per comunicare con i vari client connessi viene usato il package `java.nio`: il server ottiene le richieste dei vari client tramite un `Selector` e, se non è una semplice richiesta di connessione, la invia ad un `Worker` per eseguirla.

Per evitare di creare un nuovo `ByteBuffer` ad ogni scrittura/lettura, ogni `SelectionKey` ha associato un oggetto di tipo `KeyAttachment`, che contiene un `ByteBuffer` usato solamente nella comunicazione con il dato client e il nome dell'utente che ha eseguito il login su tale client: in questo modo si possono evitare semplici attacchi al sistema.

8.4 Strutture dati e persistenza

Il subpackage `winsome.server.datastructs` contiene le principali strutture usate dal server per memorizzare lo stato del Social Network. In particolare per modellare i post sono state create due diverse strutture dati:

- gli `OriginalPost`, che corrispondono ad un post del Social Network che non sia un rewin di un post precedente;

- i `Rewin`, che invece contengono al loro interno un riferimento al post originale.

In questo modo tutti i tipi di post hanno un loro identificativo, ma eseguire un'operazione su un post originale oppure su un suo qualsiasi rewin porta allo stesso risultato; inoltre fare il rewin di un rewin è equivalente a fare il rewin del post originale.

Per memorizzare tali dati sul filesystem vengono usate le funzionalità della libreria GSON. Tuttavia, invece di sfruttare la serializzazione automatica, le strutture dati del server vengono serializzate e deserializzate manualmente usando i metodi offerti dalle classi `JsonReader` e `JsonWriter`. Ciò consente di persistere solamente le informazioni necessarie, senza la ridondanza usata per ragioni di efficienza: ad esempio non viene memorizzato l'intero contenuto del post su cui si basa un rewin, ma solamente il suo identificativo.

Infine, per non bloccare il server completamente durante la persistenza, la classe `ServerPersistence` sfrutta gli iteratori *weakly-consistent* forniti dalle collezioni concorrenti: questo significa che i dati salvati mentre il server è attivo sono potenzialmente non aggiornati. Questo problema viene risolto alla chiusura del server, eseguendo un'ultima volta l'algoritmo dopo aver chiuso tutte le connessioni.

8.5 Chiusura del server

Per chiudere il server si usa un approccio *poison-pill*: appena il thread main legge un carattere inviato dall'amministratore del server viene chiamato il metodo `close()`, che sveglia il thread bloccato sulla `select()` e inizia la fase di chiusura.

In tale fase viene bloccato e terminato il thread pool, chiuse le connessioni TCP e UDP, eliminato l'oggetto remoto usato per la registrazione, e infine viene eseguito un'ultima volta l'algoritmo di persistenza dati, in modo da salvare su disco tutte le ultime modifiche.

9 Provare il progetto

Per testare il funzionamento del progetto senza dover partire da zero viene fornita la directory `.persisted-data-test`, contenente lo stato del server dopo la registrazione di alcuni utenti e dopo alcune interazioni.

Gli username degli utenti registrati si trovano nel file `.persisted-data-test/users.json` e ogni utente ha come password `pass`.