UNIVERSITÀ DI PISA

Dipartimento di Informatica
Corso di Laurea in Informatica

Progetto di Laboratorio di Reti

Reti di calcolatori e laboratorio

Prof.ssa Laura Ricci
Dott. Matteo Loporchio

Giacomo Trapani
600124 - Corso A

Anno Accademico 2021/2022

1 Premessa.

Il progetto consiste nella progettazione e realizzazione di un Social Network chiamato **WINSOME** ispirato a STEEMIT e che prevede ricompense in una valuta chiamata **WINCOIN** per i propri utenti. L'architettura utilizzata è di tipo "client-server": il client comunica attraverso la rete le proprie richieste al server.

Il progetto è stato sviluppato utilizzando l'editor di testo **VSCode** e testato in ambiente Linux; di seguito, gli output dei comandi `javac -version` e `java -version`:

```
javac 11.0.13
openjdk version "11.0.13" 2021-10-19
OpenJDK Runtime Environment 18.9 (build 11.0.13+8)
OpenJDK 64-Bit Server VM 18.9 (build 11.0.13+8, mixed mode, sharing)
```

Per "orientarsi" tra le varie directories e avere un'idea approssimativa dell'organizzazione del progetto, si rimanda al file `README.md`.

All'interno della cartella `docs` viene resa disponibile la documentazione del codice.

Per la compilazione, viene messo a disposizione un Makefile con i target `build` per la creazione dei file .jar e `all` per compilare.

Per l'esecuzione vengono messi a disposizione due script `run_client.sh` e `run_server.sh` che si occupano (rispettivamente) di avviare l'esecuzione del client e del server (per entrambi, procedono anche alla compilazione nel caso in cui questa non sia ancora avvenuta).

2 Server.

Il server dipende dal file di configurazione di cui si passa il path da linea di comando (in caso contrario, proverà a usare il file di configurazione `./configs/server.properties`); il file passato dovrà essere del seguente tipo:

```
SERVERADDRESS=<valid_server_address>
TCPPORT=<tcp_port>
UDPPORT=<udp_port>
MULTICASTADDRESS=<multicast_address>
MULTICASTPORT=<multicast_port>
REGISTRYHOST=<rmi_registry_host>
REGISTRYPORT=<rmi_registry_port>
REGISTERSERVICENAME=<register_service_name>
CALLBACKSERVICENAME=<callback_service_name>
USERSTORAGE=<path/to/users.json>
FOLLOWINGSTORAGE=<path/to/storage/following.json>
TRANSACTIONSSTORAGE=<path/to/storage/transactions.json>
POSTSSTORAGE=<path/to/storage/posts.json>
POSTSINTERACTIONSSTORAGE=<path/to/storage/posts-interactions.json>
BACKUPINTERVAL=<time_between_backups>
LOGFILE=<path/to/log>
REWARDSINTERVAL=<interval_between_rewards_updates>
REWARDSAUTHORPERCENTAGE=<author_percentage>
COREPOOLSIZE=<core_pool_size>
MAXIMUMPOOLSIZE=<maximum_pool_size>
KEEPALIVETIME=<keep_alive_time>
THREADPOOLTIMEOUT=<thread_pool_timeout>
```

I parametri possono essere in qualsiasi ordine, non vengono ammessi argomenti non validi (i.e. non si permettono stringhe dove ci si aspetterebbe un valore numerico) e nessuno è opzionale. La sintassi adottata è quella dei file properties.

Struttura interna. Al momento dell'avvio, il server - sulla base dei parametri specificati nel file di configurazione - eventualmente ripristina utenti e post precedentemente caricati e inizializza le proprie strutture dati: tra queste si menzionano un thread pool dedicato alla gestione delle richieste provenienti dai client connessi, dei thread dedicati

per il setup di RMI, Multicast, Logging e Backup (non sottomessi al pool), un insieme concorrente utilizzato dai thread per condividere informazioni sui client e uno shutdown hook per gestire correttamente la terminazione (che - tipicamente - avviene inviando SIGINT).

Il server funziona seguendo il modello "manager-worker"; il manager (ossia il `Main`) si mette in ascolto utilizzando un `Selector` (dunque, si usano sia **NIO** sia il **Thread pooling**) e si occupa di accettare nuove connessioni. Ogniqualvolta un canale risulta essere pronto in lettura o scrittura, istanzia un thread worker dedicato e lo sottomette al thread pool (nel caso in cui il canale sia disponibile in lettura, si occupa anche di istanziare un buffer da utilizzare per la gestione della richiesta).

Thread: RMITask. Un thread `RMITask` viene istanziato per la gestione di RMI. Provvede a esportare lo storage degli utenti e pubblicare il riferimento all'oggetto remoto e, se avvenuta con successo, si mette in idle chiamando una `sleep` all'interno di un ciclo infinito; ricevuta una `interrupt`, si risveglia, libera le risorse allocate e termina.

Thread: RewardsTask. Un thread `RewardsTask` viene istanziato per il calcolo periodico delle ricompense e il conseguente invio del messaggio (di avvenuto calcolo) via UDP Multicast. La periodicità viene implementata eseguendo una `sleep` per l'intervallo di tempo necessaria; se viene sollevata una `InterruptedException`, libera le risorse allocate e termina.

Thread: LoggingTask. Un thread `LoggingTask` viene istanziato per gestire il meccanismo di logging. Si occupa di prelevare da una coda concorrente condivisa coi thread workers il messaggio da salvare nel log e, ricevuta una `interrupt`, libera le risorse allocate e termina.

Di seguito, si indica con "[x]" il fatto che la stringa "[x]" sia presente se x è diverso da null. Un messaggio all'interno del log segue la seguente sintassi:

```
# se la richiesta viene gestita correttamente
[<timestamp>][<thread id>][<client id>]{[<username>]}[<comando>][<response code>]
# se il client si disconnette bruscamente
[<timestamp>][<thread id>][<client id>]{[<username>]}[DISCONNECTION]
# se viene sollevata IOException
[<timestamp>][<thread id>][<client id>]{[<username>]}[I/O ERROR <exception message>][
    DISCONNECTION]
```

Thread: BackupTask. Un thread `BackupTask` viene istanziato per gestire il meccanismo di backup. Si occupa di chiamare i metodi forniti dalle API degli storage (degli utenti e dei post) e si rimette in idle chiamando una `sleep` per l'intervallo di tempo specificato nel file di configurazione; ricevuta una `interrupt`, termina.

Thread: ShutdownHook. Il main provvede a installare uno shutdown hook che si occupa di inviare `interrupt` a tutti i thread menzionati sopra e di liberare le risorse allocate.

Thread worker: RequestHandler. Un thread `RequestHandler` viene istanziato per la gestione di un canale disponibile in lettura, è il vero e proprio backbone dell'intero sistema in quanto deve occuparsi del parsing di una richiesta e della conseguente costruzione del messaggio di risposta. La logica implementata è la seguente: si valida la richiesta e la si gestisce ricorrendo ai metodi forniti dalla API degli storage (degli utenti e dei post), si formatta ed invia la risposta alla coda concorrente condivisa col task dedicato al logging, si costruisce un messaggio di risposta formato da una coppia (codice, risposta) lo salva su un insieme condiviso e "avvisa il manager" (ossia risveglia il Main che, a questo punto, provvederà a registrare il canale in scrittura).

Thread worker: MessageDispatcher. Un thread `MessageDispatcher` viene istanziato per la gestione di un canale disponibile in scrittura. Si occupa di recuperare il buffer di risposta costruito dal RequestHandler e di inviarlo; infine, risveglia il Main che registrerà nuovamente il canale per eventuali letture successive.

Strutture dati condivise. Il server usa direttamente tre strutture dati condivise (e completamente slegate una dall'altra):

- l'insieme `toBeRegistered` (istanziato come un `ConcurrentHashMap.newKeySet` e utilizzato sia dal manager sia dagli worker) preserva canale, codice dell'operazione per cui deve essere registrato (i.e. lettura o scrittura) e il `ByteBuffer` utilizzato finora.
- la coda concorrente `logQueue` (istanziata come un `LinkedBlockingQueue` e utilizzata sia dagli worker sia dal thread che si occupa del logging) viene concorrentemente riempita dai thread worker e svuotata dal thread che si occupa del logging.
- la tabella hash `loggedInClients` (istanziata come una `ConcurrentHashMap`) preserva la coppia canale, nome utente con cui quel client ha eseguito l'operazione di login.

3 Storage.

Il server ricorre alle API fornite dalle classi `UserStorage` e `PostStorage` per la gestione di tutte le richieste (valide) del client. All'interno del package dedicato (i.e. il package `server.storage`), viene fornita la classe dichiarata package private `Storage` che implementa due meccanismi di backup: `backupCached` che implementa un meccanismo di caching scrivendo in append nel file specificato tutti e soli i dati dichiarati nuovi (non va a verificarlo, altrimenti si perderebbe tutto il vantaggio nell'efficienza) e `backupNonCached` che sovrascrive l'intero file specificato andando a salvare tutti i dati. Per effettuare i backup, nello specifico, sia usano `NIO` e `Gson` .

Di seguito, si andranno a descrivere le classi (e non le interfacce poiché meno interessanti e opportunamente documentate nel codice).

UserMap Una `UserMap` è uno storage di utenti che fa uso di tabelle hash. Per la descrizione dei metodi implementati, si faccia riferimento alla documentazione; di seguito si discuteranno le scelte implementative e la gestione della concorrenza.

Utilizzando le tabelle hash `interestsMap` e `followersMap` (oltre alle due ovvie tabelle username->user) si riesce a ridurre a $O(1)$ la complessità algoritmica in tempo di tutti i metodi messi a disposizione.

Poiché alcuni metodi modificano lo stato interno dello storage (nello specifico si tratta di `register` , `handleFollowUser` , `handleUnfollowUser` e `backupUsers`) e questo è "distribuito" su più variabili, si sceglie di implementare l'atomicità nelle modifiche utilizzando due strati di `ReentrantReadWriteLock` : il primo strato corrisponde a `backupLock` , acquisita in lettura da tutti i metodi a eccezione di quello per il backup e al secondo `dataAccessLock` , acquisita in scrittura da tutti e soli i metodi che modificano lo stato interno dello storage. La prima lock risulta necessaria poiché - come evidente dal codice - la classe implementa un meccanismo di caching e, durante il backup, deve trasferire nei dati cached i dati nuovi (i.e. spostarli da `usersToBeBackedUp` a `usersBackedUp`).

Si noti inoltre che modificare lo stato interno di un certo utente non corrisponda a modificare lo stato interno dello storage e che i metodi forniti dalla classe `User` permettano modifiche concorrenti allo stato del medesimo utente.

PostMap Una `PostMap` è uno storage di post che fa uso di tabelle hash. Analogamente all'analisi della classe UserMap, si rimanda alla documentazione per la descrizione dei metodi messi a disposizione e si discuteranno scelte implementative e gestione della concorrenza.

Utilizzando la tabella hash `postsByAuthor` (oltre alle due ovvie tabelle id->post) si riesce a ridurre a $O(1)$ la complessità algoritmica in tempo di tutti i metodi messi a disposizione.

Per quanto concerne la gestione della concorrenza, il discorso è lo stesso che per la classe UserMap; i metodi in gioco sono però `handleCreatePost` , `handleDeletePost` (che forza anche un flush della cache al backup successivo), `handleRewin` e `backupPosts` .

Come prima, si noti che modificare lo stato interno di un certo post non corrisponda a modificare lo stato interno dello storage e che i metodi forniti dalla classe `RewinPost` permettano modifiche concorrenti allo stato del medesimo post.

4 User.

La classe `User` descrive un utente all'interno dello storage. Per permettere modifiche in concorrenza allo stato interno di un utente si fa uso di blocchi `synchronized` all'interno dei metodi.

5 Post.

La classe astratta `Post` descrive lo scheletro di un post all'interno dello storage. Per la generazione concorrente di nuovi post da parte di thread differenti, si fa uso di una variabile `generatorID` dichiarata `AtomicInteger`.

RewinPost. La classe `RewinPost` implementa la classe astratta `Post`. Per permettere modifiche in concorrenza allo stato interno si fa uso di metodi `synchronized`.

6 Client.

Il client dipende dal file di configurazione di cui si passa il path da linea di comando (in caso contrario, proverà a usare il file di configurazione `./configs/client.properties`); il file passato dovrà essere del seguente tipo:

```
SERVERADDRESS=<valid_server_address>
TCPPORT=<tcp_port>
UDPPORT=<udp_port>
REGISTRYHOST=<rmi_registry_host>
REGISTRYPORT=<rmi_registry_port>
REGISTERSERVICENAME=<register_service_name>
CALLBACKSERVICENAME=<callback_service_name>
```

Valgono le stesse osservazioni fatte per il server (per quanto riguarda il file di configurazione).

Di seguito, si andrà a dare una descrizione piuttosto sommaria (sebbene sufficiente) del comportamento del client poiché per la descrizione dei comandi (sia a livello sintattico sia semantico) si rimanda al codice considerato che si tratta solo di inviare e ricevere stringhe di un certo formato via TCP (le uniche funzionalità davvero implementate dal client riguardano solo la formattazione delle risposte ricevute dal server).

Struttura interna. Al momento dell'avvio, il client si connette via TCP al server seguendo le indicazioni precisate nel file di configurazione. Una volta connesso, legge da linea di comando attraverso un loop infinito fino a che non viene inserito il comando di quit ":q!".

Per l'invio di messaggi al server e la gestione delle risposte ricevute si fa riferimento alla classe `Command` definita all'interno del package `Client`.

Concorrenza e strutture dati condivise. Al momento della login, il client invia al server una richiesta per recuperare i followers già presenti (per l'utente per cui si è effettuata la login) e una (richiesta) per ricevere le coordinate di multicast.

A questo punto, il client istanzia un `RMIFollowerSet` inizializzato con tutti gli utenti ricevuti e un thread di tipo `MulticastWorker` dedicato alla gestione dei messaggi di Multicast con cui condivide `multicastMessage` (una coda concorrente dichiarata di tipo `ConcurrentLinkedQueue`) e, prima di leggere da STDIN un nuovo comando, controlla se nella coda è stato aggiunto un messaggio e, in tal caso, lo rimuove e lo stampa su STDOUT.

Formattazione dei messaggi di risposta. Poiché i messaggi ricevuti dal server seguono una certa sintassi e, nel caso in cui contengano delle entità allora queste sono dei dati JSON validi, il client implementa delle classi definite private static corrispondenti a ciascuno dei possibili dati JSON ricevibili e utilizza `Gson` per effettuarne il parsing e, di seguito, istanziare oggetti del tipo appropriato.

7 Protocollo di comunicazione.

Per la comunicazione tra il client e il server viene messa a dispozione - all'interno del package "api" - la classe `Communication` (all'interno dell'omonimo file).

I messaggi inviati e ricevuti seguono in generale la seguente sintassi: $LUNGHEZZA + MESSAGGIO$ con LUNGHEZZA la lunghezza di MESSAGGIO convertito in bytes, + l'operatore di concatenazione e MESSAGGIO il messaggio in bytes: in questo modo si permette al chiamante di essere sicuro di aver inviato e/o ricevuto per intero un messaggio (evitando qualsivoglia lettura o scrittura parziale).

Nel caso particolare in cui un messaggio M sia formato da una collezione di entità, allora MESSAGGIO sarà a sua volta una collezione di messaggi con la struttura sopra menzionata e, dunque, M sarà del tipo $LUNGHEZZA_{TOTALE} + \sum_{i=1}^{n} LUNGHEZZA_i + MESSAGGIO_i$ (intendendo sempre + come operatore di concatenazione).

I metodi che si occupano dell'invio e della ricezione usano `NIO`.

Codici di risposta. Ogni messaggio inviato dal server al client è formato da due parti (<codice>, <corpo>) separate dal carattere di controllo CRLF, per avere informazioni sui codici implementati si rimanda alla classe `ResponseCode` definita all'interno del package `api`.