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

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).

Struttura del progetto: i packages.

Il progetto si articola in più packages, ognuno con delle funzioni ben definite:

api

Il package contiene metodi e classi **condivise dal client e dal server**:

- `CommandCode`: contiene le definizioni dei codici che identificano univocamente un comando inviato dal client al server.
- `Communication`: contiene le definizioni dei metodi che implementano il protocollo di comunicazione usato tra client e server. Si prevedono i seguenti passaggi per il mittente:
  - Si trasforma il messaggio in una stringa S leggibile dal destinatario, si distinguono due casi:
    - Il messaggio contiene una entità (e.g. il server sta inviando al client un post, il client sta inviando una richiesta al server). In questo caso, il messaggio viene trasformato in un dato JSON.
    - Il messaggio non contiene una entità, è il caso dei messaggi di risposta inviati dal server al client per tutte quelle operazioni per cui viene richiesto dalla specifica di restituirne solo l'esito. In questo caso, non si fa nulla.
  - Si codifica S in US_ASCII, l'output è un vettore B di bytes.
  - Si antepone a B un valore intero che ne indichi la lunghezza, ottenendo un nuovo messaggio M.
  - Si invia M al destinatario.

  Il destinatario compie i seguenti passaggi:
  - Legge un valore intero dal messaggio M ricevuto.
  - Decodifica il messaggio ricevuto, si distinguono due casi:
    - Il messaggio contiene una entità: nuovamente, si distinguono due casi:
      - Il messaggio proviene dal server: il messaggio sarà formato da un solo dato JSON o da una serie di dati JSON. Nel secondo caso, a ogni dato il mittente avrà anteposto la lunghezza del vettore di bytes ricavato dalla sua codifica: si usa questa informazione per determinare inequivocabilmente i confini di ciascuno.

        - Il messaggio proviene dal client: il messaggio sarà formato da un solo dato JSON.
      - Il messaggio non contiene una entità: in questo caso, si legge la stringa ricevuta.

    Ovviamente, per entrambi (sia mittente sia destinatario) risulta necessario conoscere il formato del dato JSON ricevuto.
- `ResponseCode`: contiene la definizione del formato dei messaggi di risposta inviati dal server: ognuno di questi sarà formato da un codice che ne indichi l'esito, il separatore CRLF e una Stringa che ne forma il corpo. Nel caso in cui il messaggio di risposto abbia un codice diverso da OK (200), il corpo del messaggio sarà sempre e solo la descrizione dell'errore avvenuto nell'elaborazione della richiesta.

Inoltre contiene il package **rmi**, dedicato alle interfacce delle classi remote utilizzate per RMI e dalle eccezioni checked che vengono lanciate dai metodi remoti:

- `RMICallback`: interfaccia per le callbacks RMI implementata dal server.
- `RMIFollowers`: interfaccia per le callbacks RMI implementata dal client.
- `UserRMIStorage`: interfaccia per il metodo remoto `register` utilizzato per registrare un utente a WINSOME.
- Le altre classi sono tutte e sole le eccezioni checked lanciate dal metodo register.

client.

Il package contiene le classi e i metodi utilizzati **solo dal client**:

- `Colors`: contiene le costanti utilizzate per stampare stringhe colorate sul terminale.
- `Command`: contiene le definizioni dei metodi che:
  - trasformano le richieste inviate da CLI dal client in un formato leggibile dal server;
  - ricevono la risposta dal server e la gestiscono adeguatamente:
    - Se il messaggio di risposta contiene una entità, modificano le strutture dati in input in modo tale da poterle salvare.
    - Se il messaggio di risposta non contiene una entità, decide - sulla base di un flag booleano - se stampare o meno su STDOUT il messaggio di risposta.
- `MulticastInfo`: corrisponde al dato JSON che viene inviato dal server al client quando viene richiesta l'operazione di "RETRIEVEMULTICAST".
- `MulticastWorker`: implementa le funzionalità di un thread worker che resta in ascolto di nuovi messaggi sul canale di Multicast.
- `Response`: classe dichiarata package private che implementa il parsing del corpo di un messaggio di risposta (che si ricorda poter essere - in concreto - o una `String` o un `Set<String>`).
- `RMIFollowersSet`: classe che implementa il meccanismo di RMI callback per il client.

Logicamente, disaccoppiare il package client dal package api permette di separare il più possibile i moduli del progetto e di utilizzare - eventualmente - un client differente che chiami le funzioni della API sopra definita.

configuration

Il package contiene le classi (e le eccezioni lanciate da queste) che si occupano del **parsing del file di configurazione**.

Innanzitutto, si definisce il formato del file di configurazione per il client e per il server:

```
# CLIENT
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>
```

```
# SERVER
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.

Le classi all'interno del package sono le seguenti:

- `Configuration` : si occupa del parsing del file di configurazione usato dal client.
- `InvalidConfigException` : eccezione checked lanciata se il file di configurazione non rispetta la sintassi menzionata sopra.
- `ServerConfiguration` : si occupa del parsing del file di configurazione usato dal server.

server

Il package contiene le classi e i metodi utilizzati **solo dal server**. Si parte analizzando i package `user` , `post` e `storage` . Si rimanda la descrizione delle altre classi alla sezione dedicata alla descrizione del server.

user

Il package contiene le classi e i metodi utilizzati all'**interno del server** per la **gestione di un utente**:

- `Tag` : contiene la definizione dei tag, ossia gli interessi relativi a un utente.
- `Transaction` : contiene la definizione delle transazioni.
- `User` : contiene la definizione di un utente all'interno di WINSOME. I metodi messi a disposizione fanno uso di blocchi "synchronized" all'interno delle sezioni critiche in modo tale da poter permettere accessi in mutua esclusione (allo stesso oggetto di tipo User) a thread concorrenti. La classe è dunque thread-safe.
- Le altre classi sono le eccezioni checked lanciate dalle tre classi sopra menzionate.

post

Il package contiene le classi e i metodi utilizzati all'**interno del server** per la **gestione di un post**:

- `Post` : classe astratta che descrive lo scheletro di un post all'interno di WINSOME. Definisce la logica per la generazione degli identificativi univoci dei nuovi post: facendo uso di un `Atomic Integer` si permettono creazioni concorrenti di nuovi post da parte di più thread.

- `RewinPost`: implementa i metodi definiti nella classe astratta Post. Si noti come, poiché per il calcolo dei guadagni risulta necessario mantenere consistente lo stato di più parametri interni (nello specifico, `newVotes`, `newCommentsBy`, `newCurators` e `iterations`), si renda necessaria l'implementazione di un meccanismo di mutua esclusione per tutti e soli i metodi che modificano almeno uno di questi (gestita dichiarando i metodi `getGainAndCurators`, `addComment` e `addVote` "synchronized"). Tutti gli altri metodi che vanno a leggere o modificare lo stato interno di uno stesso post lavorano su variabili di tipo `ConcurrentHashMap` o suoi derivati, quindi la classe risulta thread safe.

storage

Il package contiene le classi e i metodi utilizzati all'**interno del server** per la **gestione dello storage**:

- `UserStorage`, `PostStorage`: interfacce che dichiarano i metodi che degli storage rispettivamente di utenti e di post devono implementare.
- `Storage`: classe dichiarata package private che definisce due metodi per il backup: `backupCached` che implementa un meccanismo di caching scrivendo in append nel file specificato tutti e soli i dati dichiarati nuovi (non va a verificare che questi siano veramente nuovi per non perdere in efficienza) e `backupNonCached` che sovrascrive l'intero file specificato andando a salvare tutti i dati. Usa `Gson` per convertire ogni elemento dello storage in un dato JSON valido.
- `UserMap`: classe che utilizza tabelle hash per effettuare lo storing di oggetti di tipo User al proprio interno.
  Si noti come utilizzare le tabelle hash `interestsMap` e `followersMap` in aggiunta alla tabella che salva gli utenti permetta di ridurre a $O(1)$ la complessità algoritmica in tempo di tutti i metodi messi a disposizione (nonostante si vada a introdurre una ridondanza nella rappresentazione dei dati e a "distribuire" lo stato interno dello storage su più variabili).
  Poiché alcuni metodi modificano lo stato interno dello storage (nello specifico si tratta di `register`, `handleFollowUser`, `handleUnfollowUser` e `backupUsers`), si sceglie di gestire la concorrenza introducendo due "strati" di `ReentrantReadWriteLock`: il primo corrisponde a `backupLock`, acquisita in lettura da tutti i metodi a eccezione di quello per il backup e il secondo a `dataAccessLock`, acquisita in scrittura da tutti e soli i metodi che modificano lo stato interno dello storage. La prima lock risulta necessaria poiché 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`).
  Infine, si vada a notare come modificare lo stato interno di un certo utente non corrisponda a modificare lo stato dello storage: aggiungendo a questo la classe User sia thread safe, concludiamo che la classe è thread safe.
- `PostMap`: classe che utilizza tabelle hash per effettuare lo storing di oggetti di tipo Post al proprio interno.
  Anche in questa classe si introduce ridondanza (con le mappa `postsByAuthor`) per ridurre a $O(1)$ la complessità algoritmica in tempo dei metodi messi a disposizione.
  Per la concorrenza, vale lo stesso discorso che per la classe UserMap: la differenza sta nel fatto che i metodi in gioco siano `handleCreatePost`, `handleDeletePost` (che forza anche un "flush della cache" al backup successivo), `handleRewin` e `backupPosts`.
  Come prima, si noti come modificare lo stato interno di un post non corrisponda a modificare lo stato dello storage: dunque, anche questa classe risulta thread safe.

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`).

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 il **multiplexing** 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 specificato; 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.
- Si effettua il parsing della richiesta e si chiama il metodo fornito dalla APi di uno degli storage in grado di elaborare la richiesta.
- 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) e lo si salva su un insieme condiviso.
- Si "avvisa il manager": si risveglia il Main (che era bloccato su una select): a questo punto, (il Main) 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`:
  - è l'insieme delle chiavi di una `ConcurrentHashMap`;
  - viene popolato dai thread workers e svuotato dal manager;
  - preserva canale, codice dell'operazione per cui deve essere registrato (i.e. lettura o scrittura) e il `ByteBuffer` utilizzato finora.
- la coda concorrente `logQueue`:
  - è una `LinkedBlockingQueue`;
  - viene riempita dai thread worker e svuotata dal thread che si occupa del logging;
  - contiene i messaggi da scrivere nel file di logging.
- la tabella hash `loggedInClients`:
  - è una `ConcurrentHashMap`;
  - viene popolata e svuotata dai thread workers;
  - preserva la coppia canale, nome utente con cui quel client ha eseguito l'operazione di login.

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`).

Non implementa molte funzionalità, è un thin client che si occupa di fare perlopiù "pretty printing" di oggetti di vari tipi.

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 al 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 `MulticastWorker` dedicato alla gestione dei messaggi di Multicast con cui condivide la coda concorrente `multicastMessage`: una `ConcurrentLinkedQueue` che il client controlla sia vuota (ed eventualmente, svuota) prima di leggere un nuovo comando da STDIN.

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.

Testing del progetto.

Ai fini del testing del progetto, si ricorda la presenza dei due script per agevolarne l'esecuzione. Inoltre, per non partire da uno storage completamente vuoto, all'interno della cartella `storage` si troveranno degli utenti già registrati; per ognuno valgono le seguenti regole:

- Il nome utente è formato dalla stessa lettera (che chiameremo l) ripetuta 3 volte.
- La password è uguale al nome utente.
- I tag a cui è interessato sono l, l+1, l+2, l+3, l+4.

Si fa un esempio: l'utente **aaa** ha come password **aaa** e segue i tag **a**, **b**, **c**, **d**, **e**.