Sommario

Winsome Relazione – Benedetti Gabriele matr. 602202

Introduzione

La specifica del progetto richiede di implementare un Social Network simil Steemit, in cui gli utenti possono interagire tra di loro scambiandosi like, dislike, commenti e seguendosi a vicenda. La caratteristica principale di questo social è che le interazioni vengono premiate mediante valuta WinsomeCoin.

Il progetto fa uso di una interfaccia grafica e quindi NON è a riga di comando.

GitHub: https://github.com/gbenedetti22/Progetto-Reti-Winsome

Scelte generali effettuate

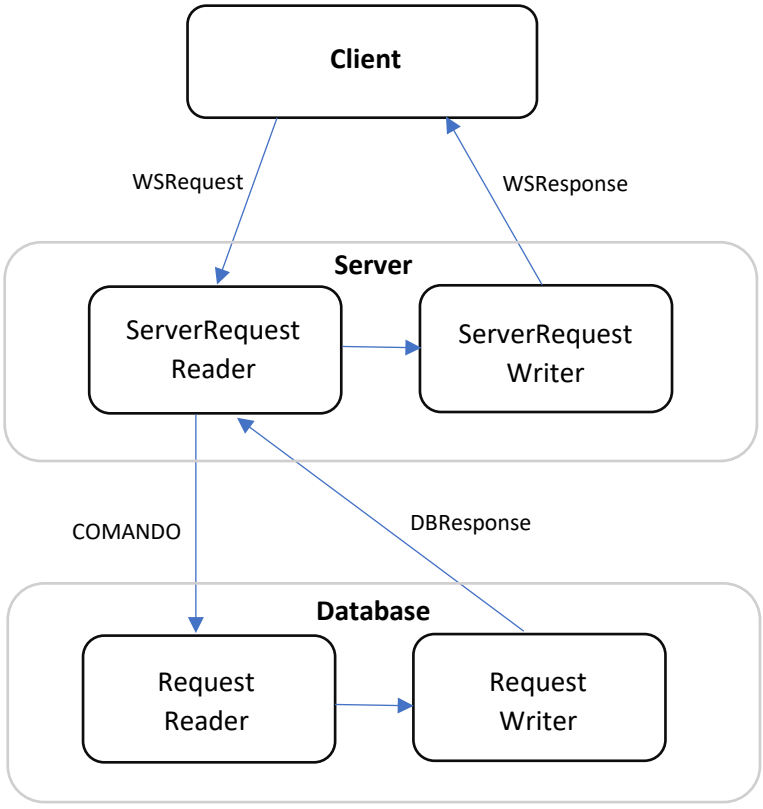
- Per la generazione di ID è stata usata la classe UUID
- Il progetto fa uso solo di Java NIO, quindi selettori e canali. In particolar modo, viene usato il PipedSelector e i ChannelLineSender/ChannelLineReceiver, questo per garantire maggior efficienza (grazie all uso del threadpool) e minor overhead nell allocazione dei buffer. Per maggiori informazioni, vedere la javadoc inclusa nel progetto al percorso: "javadoc\index.html"
- Come richiesto dalla specifica, la rete sociale è composta da Server e Client, dove il Server è a sua volta suddiviso in 2 processi: processo Database e processo Server
- I file di configurazione vengono cercati nella cartella corrente e non è possibile cambiare questa cosa

I ruoli sono così ben definiti:

Database
- Gestione e persistenza sul disco dei dati

Server
- Gestione degli utenti loggati
- RMI
- Calcolo delle ricompense
- Ordinamento dei dati ricevuti dal Database
- Gestione dei gruppi Multicast

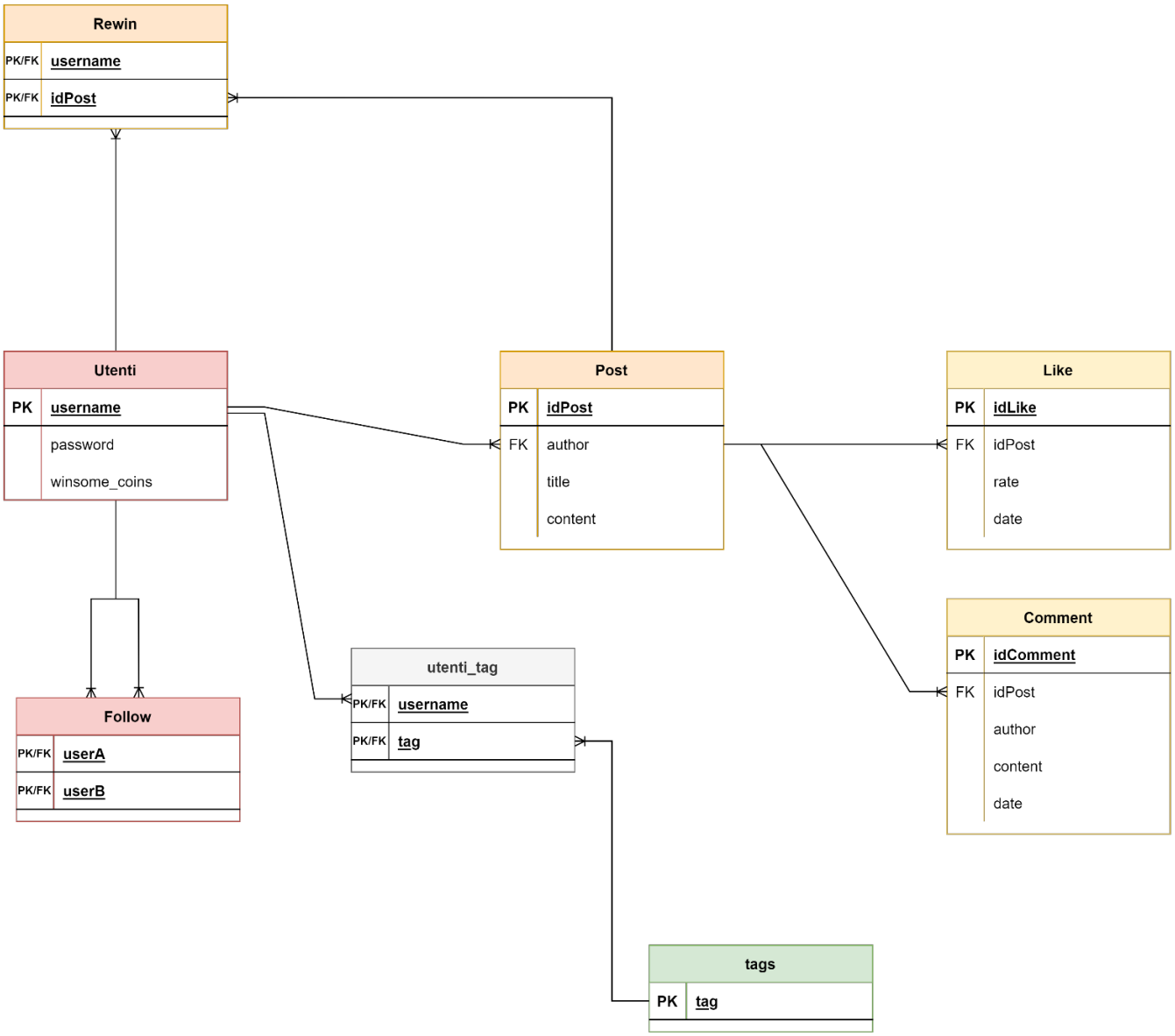
Client
- Fa richieste al Server
- Gestione UI



Database

Rappresentazione in memoria

Un primo approccio possibile per strutturare il database è usare una visione “relazionale”.



Database approssimativo

Per rappresentare questo modello in memoria, si può pensare che ogni tabella sia una classe e che ogni relazione sia una struttura dati

Esempio: “Utenti” -> “Post” = HashMap<Utenti, Post>

Vantaggi:

- Facile da implementare
- Maggior flessibilità nella serializzazione (salvo solo le tabelle “modificate”)

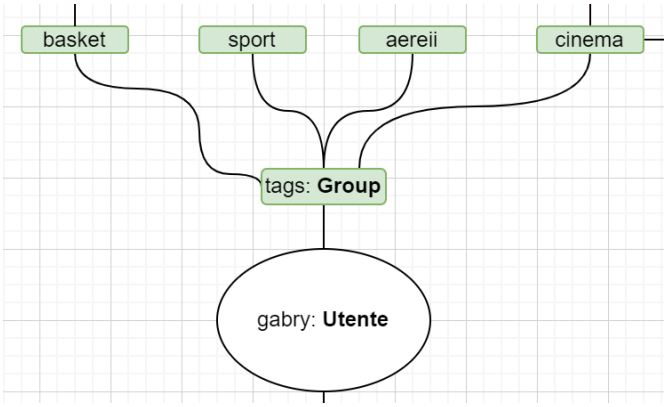
Svantaggi:

- Maggior costo in memoria (9 strutture dati + altre di supporto per l’ordinamento, il calcolo delle ricompense ecc)
- Più difficile la gestione della concorrenza
    - vanno rese atomiche le operazioni (per operazioni si intende tipo la creazione di un Post) e, molte di queste, fanno uso di più strutture dati, che anch’esse devono avere la loro versione concorrente
- Possibili doppie copie tra le strutture dati
    - Questo vuol dire che l aggiornamento di una struttura dati, ne implica l aggiornamento di altre

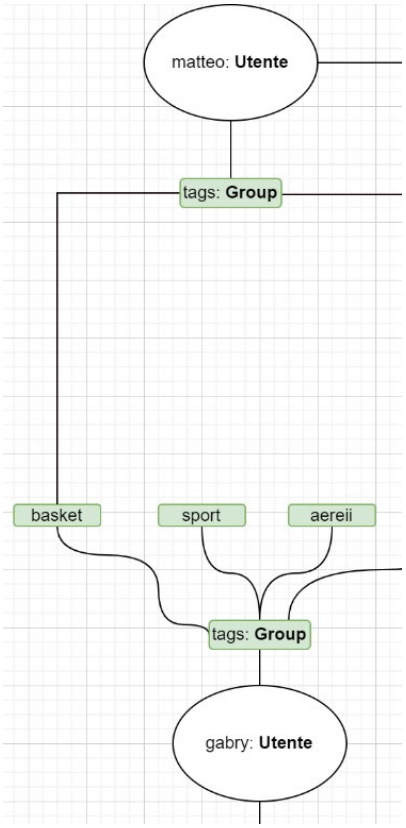
Un ulteriore modo per rappresentare il Database è mediante un grafo bidirezionale.

Creazione di un utente

La creazione di un utente è la semplice aggiunta di un nodo all interno del grafo, connesso ai suoi tag.



I tag non vengono aggiunti direttamente al nodo Utente, bensì vengono raggruppati (cioè collegati ad un nodo specifico denominato “GroupNode”). Questo mi permette di reperire i tag di un certo Utente in tempo costante (mi basta vedere i nodi adiacenti del nodo Group). Per vedere quali utenti hanno in comune un certo tag X, mi basta prendere i nodi adianceti di quel tag X



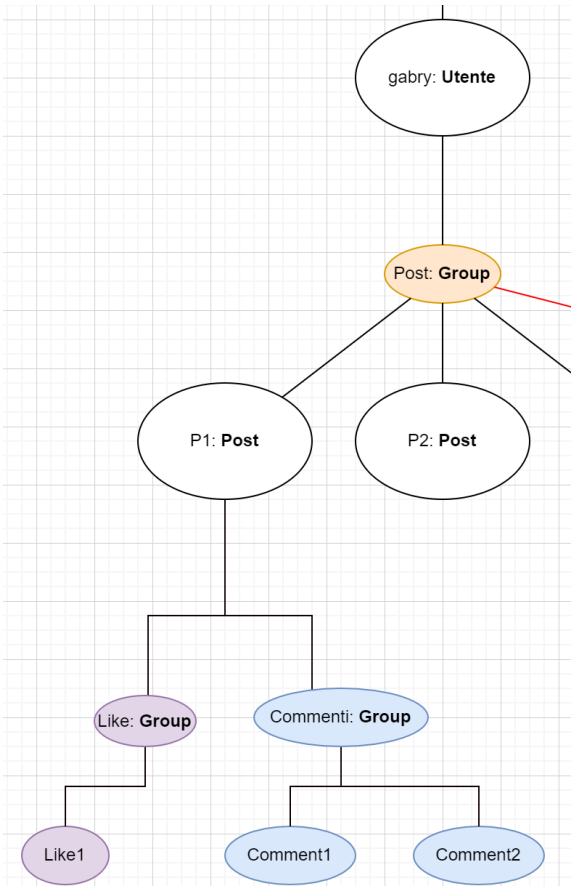
Supponiamo che si voglia sapere quali sono gli utenti che hanno almeno un tag in comune con l’utente “gabry”. Basta prendere i tag di “gabry” e vedere quali sono i nodi adiacenti.

In questo caso, l’utente “gabry” e l’utente “matteo” hanno in comune il tag “basket”. Prendendo i nodi adiacenti del tag “basket” ottengo:

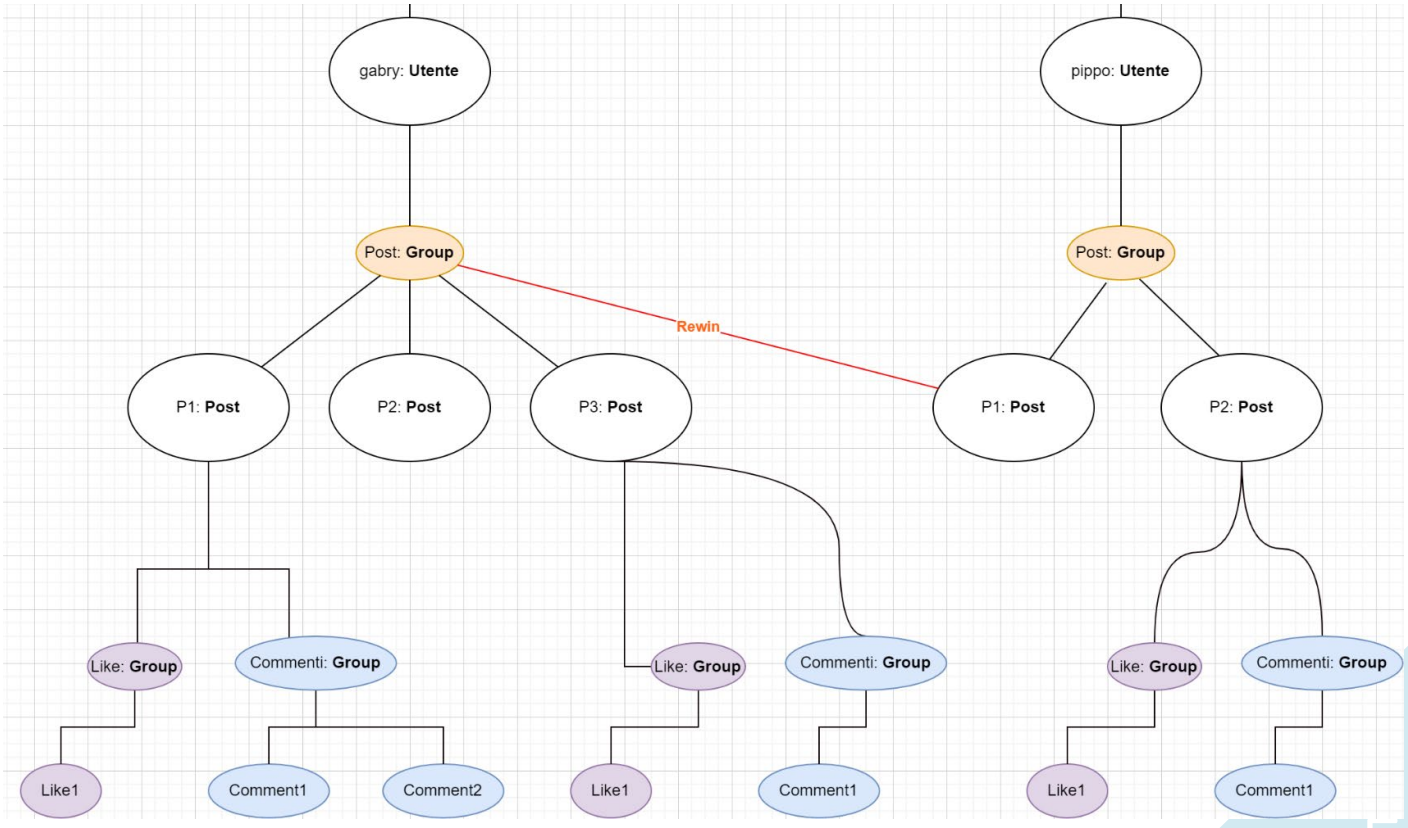
“gabry” (che viene scartato) e “matteo”

Pubblicazione Post

La pubblicazione di un Post avviene in modo simile alla memorizzazione di un tag: i post vengono raggruppati sotto un nodo GroupNode, i like e i commenti diventano nodi attaccati al nodo Post (anche questi, opportunamente raggruppati)



I Rewin diventano dei semplici collegamenti



Il grafo è una sola rappresentazione logica, i dati NON vengono memorizzati direttamente sul grafo (es. il nodo Post contiene solo l id del Post e non tutte le informazioni quali l autore, il contenuto ecc). Questo perché il grafo è difficile da serializzare e inoltre ho bisogno di poter reperire, in tempo costante, le informazioni su utenti e Post (dato lo username e l id rispettivamente parlando). Oltre al grafo, quindi, sono necessarie altre 2 strutture dati

Vantaggi:

- Meno strutture dati da usare:
  - HashMap<Username, Utente>
  - HashMap<idPost, Post>
  - Grafo
- Più facile la rimozione => eliminando un nodo Post, elimino in automatico possibili Rewin, like e commenti
- Nessun duplicato => come detto, il grafo si appoggia alle 2 strutture dati ausiliarie, quindi supponendo per assurdo che esista un idPost nel grafo ma non nella HashMap<idPost, Post>, il Post comunque non esiste
- Più facile da rendere thread-safe => basta rendere atomiche le poche operazioni che si eseguono su un grafo (aggiunta di un nodo, vedere i nodi adiacenti ecc.). Vedere sezione concorrenza

Svantaggi:

- Molto più difficile la serializzazione
- Più difficile da implementare

Come è facile intuire, ho optato per la scelta del grafo. Siccome in Java non esiste questa struttura dati, per rimanere in tema Google, ho utilizzato la libreria Guava.

Serializzazione

Il Grafo di Guava non è serializzabile ma, anche se lo fosse, non converrebbe comunque in quanto:

1. i grafi hanno una loro logica e potrei avere nodi all interno del grafo che non dovrebbero esserci (si pensi ad un Post cancellato: potrei avere nel grafo un nodo idPost che fa riferimento ad un post cancellato)
2. dovrei fare un traversing, ad ogni avvio, di tutto il grafo per controllare tutti i nodi (per via del punto 1)

E' preferibile, quindi, ricostruire tutta la logica ad ogni avvio.

Come richiesto dalla specifica tutti gli utenti, i Post, i commenti e i like vengono salvati in formato JSON. In particolare:

- L' HashMap<Username, Utente> viene serializzata nel file **users.json**
- L'HashMap<idPost, Post> viene serializzata nel file **posts.json**
- I commenti e i like vengono serializzati singolarmente, dove il nome del file corrisponde al loro id (es. idLike.json). Questi file vengono tutti messi dentro una cartella denominata "**jsons**"

Per ricostruire la logica del grafo, viene usata la seguente tecnica:

- Ogni utente ha il proprio file denominato con il suo username
- Dentro al file, vengono scritti dei record che hanno la seguente struttura: "tipologia;id" (es. LIKE;idLike)

Esempio. Supponiamo che:

- HashMap<Username, Utente> contenga l'utente Pippo
- HashMap<idPost, Post> contenga un Post pubblicato da Pippo con ID = 1234
- La cartella jsons (che ricordo, contiene tutti i like e i commenti in formato JSON di tutti i Post) contenga 2 file JSON:
  - abcd.json (commento)
  - ab123.json (like)
- Il file "Pippo" contenga i seguenti record
  - POST;1234
  - COMMENT;abcd
  - LIKE;ab123

All avvio:

- Il processo Database vede il file Pippo
- Crea un nodo Utente “Pippo”
- Comincia a leggere i record del file
- Vede che c’è un Post con ID 1234 dal record “POST;1234”: quindi accede all HashMap<idPost, Post> per controllare se esiste ancora
- Se esiste, viene creato un nodo Post, altrimenti la riga viene cancellata (vedi sotto)
- Vede che c’è un commento dal record “COMMENT;abcd”: quindi va nella cartella jsons e controlla se esiste un file json denominato abcd.json
- Se esiste
    - il file viene de-serializzato nell oggetto Comment (che contiene l id del Post a cui questo commento fa riferimento)
    - viene preso l id del Post e appeso un nodo Comment come spiegato sopra
- Infine vede che c’è un like dal record “LIKE;ab123”, quindi attua lo stesso procedimento eseguito per il record COMMENT: entra nella cartella jsons, de-serializza il file ab123.json e appende il nodo Like al nodo Post corrispondente

Osservazioni:
i record POST vengono SEMPRE prima dei record COMMENT e LIKE: questo perché è impossibile che esista un commento fatto ad un Post fantasma.

Cancellazione

Quando un Post viene cancellato, tutti i record POST, LIKE e COMMENT devono essere cancellati dal file. Un primo approccio, sarebbe quello di scorrere tutto il file alla ricerca dei record da togliere: questo approccio funziona, ma è molto inefficiente.
Di fatto, un singolo post può avere migliaia e migliaia di like e commenti! E’ quindi impensabile dover scorrere tutti questi record in continuazione.
E’ quindi stata utilizzata la seguente tecnica: quando il Database si avvia e analizza tutti i record, viene preso il numero di riga e salvato dentro l’oggetto corrispondente. Così facendo, quando devo eliminare un record (che sia COMMENT, LIKE o POST) mi basta eseguire un seek a quel numero di riga e cancellarla.
La cancellazione della riga avviene mediante una sostituzione delle lettere con il carattere #, questo perché:

1) Se muovessi le altre righe, altre classi farebbero riferimento a righe sbagliate
2) Per muovere le altre righe, dovrei scorrere l’intero file

Per pulire i file pieni di #, basta avviare il Database con l opzione “--clear”
I Rewin hanno un file a parte che viene letto per ultimo, questo perché sono dei semplici collegamenti.

Il salvataggio dei dati non avviene in parallelo, questo perché è molto difficile far salvare i dati ad N thread solo se è passato un tot di tempo (andrebbero sincronizzati tutti sulla stessa ora); inoltre mi servirebbe una lock per ogni file che andrei ad usare solo se il quantitativo di tempo è scaduto.
Ho quindi preferito fare in modo che sia un thread solo a salvare ogni tot i dati del Database sul disco, mettendosi in ascolto su una coda che viene svuotata se è scaduto il tempo o se è troppo piena.

Concorrenza

Guava di per sé, non offre un grafo concorrente. Di fatto, però, le operazioni sul grafo eseguite sono relativamente poche: aggiunta di un nodo, rimozione, vedere i nodi adiacenti e vedere se 2 nodi sono connessi. Per questo è stata usata una RWLock (read-write-lock).

Per le altre 2 HashMap è stata usata la loro versione concorrente. Di fatto, tutte le possibili operazioni (creazione di un post, aggiunta di un like/dislike ecc) sono eseguite direttamente sul grafo, ergo rendendo thread-safe il grafo, rendo atomiche anche tutte le operazioni richieste dalla rete sociale.

READ-LOCK:

- Vedere nodi adiacenti
- Vedere se 2 nodi sono connessi

WRITE-LOCK:

- Aggiunta di una connessione tra 2 nodi
- Rimozione di un nodo e/o di una connessione

Come comunicare con il Database

Il Database accetta comandi con la seguente sintassi:
"Comando: parametri"
Esempio: "CREATE USER: user password [tag1, tag2, tag3]"
Il Database risponderà con l'oggetto DBResponse che conterrà i dati richiesti (lista di post, utenti ecc)

Server

Il Server è stato implementato come un Server Multithread che fa uso del PipedSelector per ricevere le richieste. Il Client, quindi, invia le richieste al Server e lui risponderà tramite JSON o tramite messaggi di conferma (es. se richiedo di visualizzare un Post, questo verrà mandato tramite JSON, mentre se ne richiedo la cancellazione, il Server manderà un semplice messaggio di conferma o errore). Il Server accetta richieste mediante l oggetto WSRequest e risponde mediante WSResponse (che contiene stato dell operazione e messaggio JSON)

Scelte implementative

Invio dei Post

Purtroppo i Post possono essere molto grandi singolarmente: basti immaginare al numero di like, dislike e commenti che un singolo Post può avere.

Quindi il Server non invia tutto il Post, ma solo una "preview", ovvero una versione del Post semplificata che contiene al suo interno solo id, titolo, autore, parte del contenuto e la data di pubblicazione. Il post vero e proprio verrà scaricato solo quando l'utente clicca sulla preview.

Tuttavia anche le preview possono essere tante e doverle scaricare in continuazione è un carico che il Server non dovrebbe gestire. Queste preview, quindi, vengono salvate localmente e il Client manda al Server, per ogni utente che segue, la data dell'ultimo Post ricevuto per quell utente specifico. Se il client ha tutti i post, allora il Server risponde con un semplice "[ ]" (lista vuota = non ci sono nuovi nuovi Post) per indicare che il client possiede tutti i Post.

Più nello specifico, il client costruisce una DateMap<User, Data> dove "Data" è la data dell' ultimo post pubblicato da User. Se Data = 0 allora il Server risponde con tutti i post pubblicati da User. La data è nel formato: "ANNO/MESE/GIORNO – ORE:MINUTI:SECONDI"

Per semplicità, dato un utente A, per reperire i post dei suoi follow e i post da lui pubblicati bisogna fare 2 richieste separate al Server. Così facendo, risulta più facile costruire la Home e il Profilo.

Per l invio dei commenti il procedimento è analogo a quello per i Post, ovvero se un utente apre la sezione commenti ripetutamente, questi non vengono scaricati sempre tutti bensì solo gli ultimi.

Cancellazione di un Post

A questo punto però sorge un problema: se un client contiene localmente un Post che è stato cancellato, come faccio a sapere che il Post non esiste più?

Per risolvere a questo problema, sono stati utilizzati 2 meccanismi:

- Se il client richiede la visualizzazione di quel Post, gli compare un messaggio di errore “il Post non esiste più” e viene cancellato anche localmente
- Viene inviato a tutti client connessi, mediante Multicast, un messaggio di avvenuta cancellazione del Post identificato da autore + ID. I client che riceveranno il messaggio provvederanno a cancellare localmente il Post se ce l hanno, altrimenti scarteranno il messaggio. Se un client non riceve quel messaggio (essendo UDP, può succedere) ma ha localmente quel Post eliminato, rimane comunque valida la prima opzione

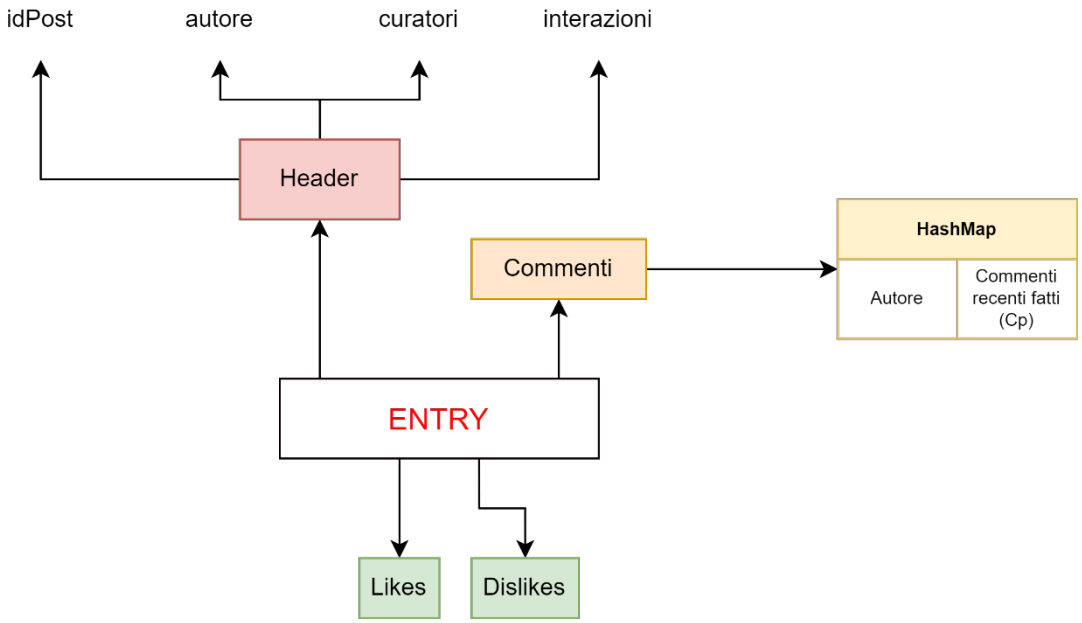
Calcolo ricompense

Il calcolo delle ricompense avviene sul Server, ma per eseguirlo è necessario che il Database gli restituisca i Post che hanno al più un commento o un like nuovo. La spiegazione di come le ricompense vengono calcolate, si suddivide quindi in 2 parti.

Lato Database

Il Database fa uso di una struttura dati ausiliaria per tenere traccia, in maniera efficiente, di quali sono i Post che contengono “qualcosa di nuovo”.

Definisco una Entry come un oggetto che contiene un Header, un Set di likes, di dislikes e una tabella commenti



EntriesStorage è la classe che gestisce la tabella di tutte queste Entry.

Quando viene messo un like o un dislike ad un Post, viene presa la Entry che ha quell idPost nell header e inserito nel Set più appropriato. Per il cambio da Like a Dislike (o viceversa), basta togliere il like dal primo Set e aggiungerlo al secondo.

Cosa diversa vale per i commenti, i quali vengono raggruppati per autore in fase di inserimento.

Per ricordarsi al riavvio quali sono le NEW ENTRIES, il Database appende ai record sul disco (vedi serializzazione) la dicitura “NEW_ENTRY” in fondo al record. Così facendo, quando il grafo viene ricreato, è possibile ricostruire anche l EntriesStorage.

Per reperire questa lista, il Server manda il comando “PULL NEW ENTRIES”: a questo punto, il Database esegue le seguenti azioni:

- Crea una copia della lista
- Pulisce la lista originale per accogliere nuove Entry
- Aggiorna i file sul disco (rimuovendo la dicitura “NEW_ENTRY dai veri record)
- Invia la copia al Server

La copia è necessaria in quanto il disco viene pulito nell istante in qui il Database riceve il comando, quindi se viene aggiunta una nuova Entry nel mentre che invio i dati al Server, potrei ricalcolare ricompense per Entry già valutate.

Un alternativa alla copia è attendere che il Database abbia inviato tutti i dati al Server, ma l’accesso al Socket è notevolmente più lento di un accesso in memoria.

E’ possibile aggiornare il profilo di un utente (o di una lista di utenti) mediante il comando “UPDATE USER”, specificando il numero coins e la data/ora del calcolo eseguito.

Lato Server

Il Server invia il comando “PULL NEW ENTRIES” e riceve la lista delle Entry spiegata nel paragrafo sopra.

Data la formula, definisco:

- **N1** = $\log\left(\max\left(\sum_0^{new\_people\_likes}(Lp), 0\right) + 1\right)$
- **N2** = $\log\left(\sum_0^{new\_people\_commenting}\left(\frac{2}{1+e^{-(Cp-1)}}\right) + 1\right)$

Per ogni Entry, il Server attua il seguente procedimento:

- Per il calcolo di **N1**..
  - Viene sottratto il numero di Like al numero dei Dislikes (prendendo come valore minimo 0), aggiungendo 1 al risultato ottenuto
  - Viene eseguito il logaritmo naturale
- Per il calcolo di **N2**..
  - Viene tenuta una variabile sommatore e, per ogni utente che ha inserito un commento di recente, viene eseguita la sommatoria applicando la formula: $\frac{2}{1+e^{-(Cp-1)}}$
  - Viene eseguito il logaritmo naturale sul risultato ottenuto + 1
- Guadagno = $\frac{N1+N2}{entry.HEADER.interactions()}$

Come è facile intuire, il calcolo di N1 è un operazione costante mentre il calcolo di N2 richiede di eseguire un loop sul numero di persone che hanno fatto al più un commento nuovo.

Il costo per il calcolo delle ricompense è determinato, quindi, sulla base di quanti utenti eseguono un nuovo commento: O($\sum_0^{entries.size()} new\_people\_commenting_i$)

Dove $new\_people\_commenting_i$ identifica il numero di utenti che hanno fatto al più un commento nuovo per quella Entry i (quindi per quel Post i)

Infine, viene eseguito il comando “UPDATE USER” sia per l autore sia per la lista di utenti (che viene passata come String, ci penserà poi il Database a riconvertirla)

Gestione Login

Quando viene eseguito il Login, il Server memorizza il Socket del Client all interno di una HashMap (insieme allo username) e invia al Client i valori del Multicast a cui connettersi. E' importante sottolineare che il Server identifica il Client mediante il Socket, quindi lui non dovrà MAI specificare lo username all interno dei comandi che invia al Server, in quanto sarà il Server stesso a sapere "chi è".

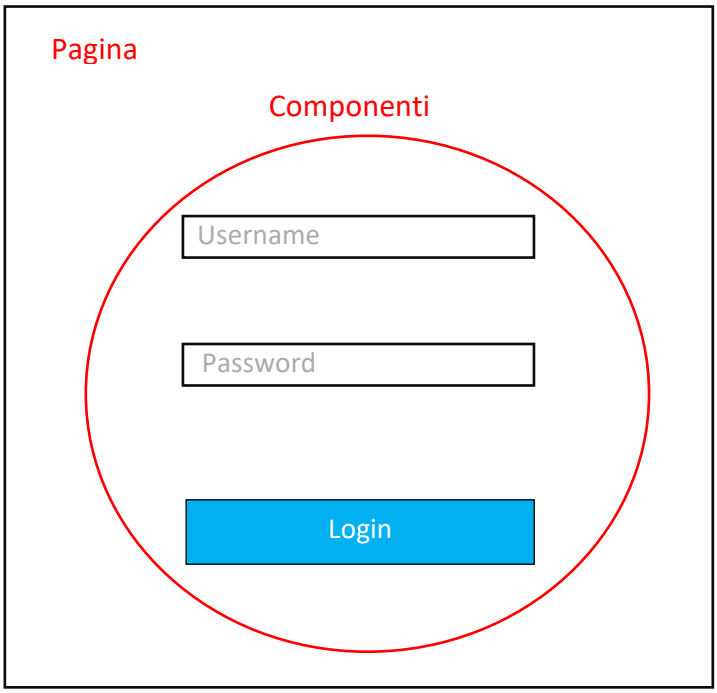
Client

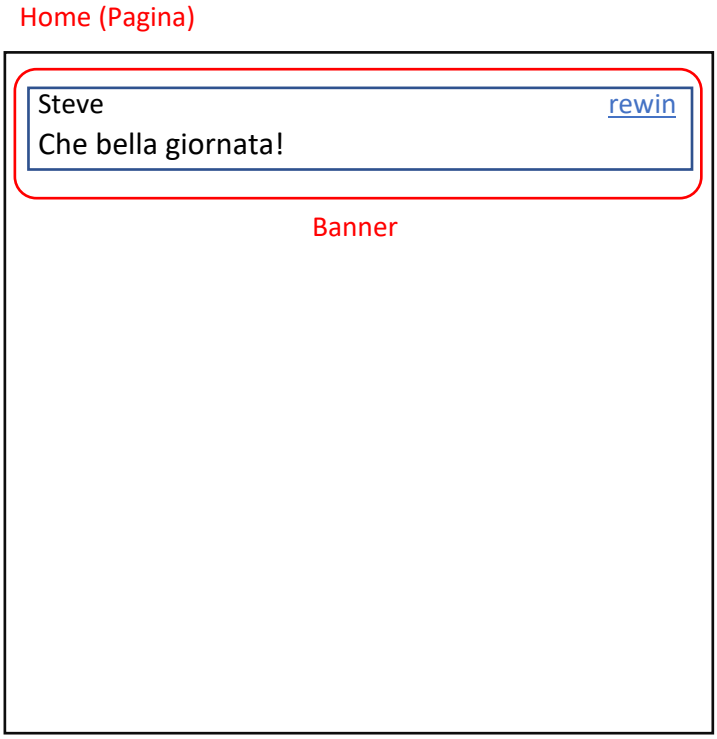
GUI

L interfaccia grafica è composta da vari componenti con cui l'utente può interagire.

Nello specifico, l interfaccia è suddivisa in:

- Pagine
- Componenti
- Banner





Come è facile intuire dai disegni, i componenti sono elementi semplici, tipo TextField o Button. I Banner, invece, sono elementi più complessi che al loro interno possono contenere altri elementi (per esempio, nel disegno sopra, l'utente può o cliccare sulla scritta per visualizzare il post in tutta la sua interezza o cliccare su "rewin" per ricondividere il Post).

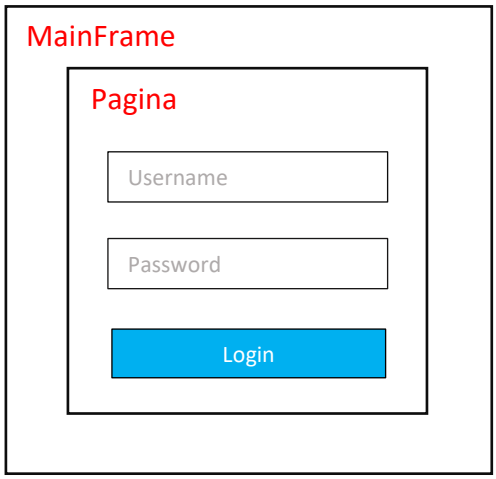
Le pagine servono a contenere tutti questi elementi.

Pagine

Le pagine corrispondono al numero di funzionalità offerte dal servizio Winsome, in particolare sono:

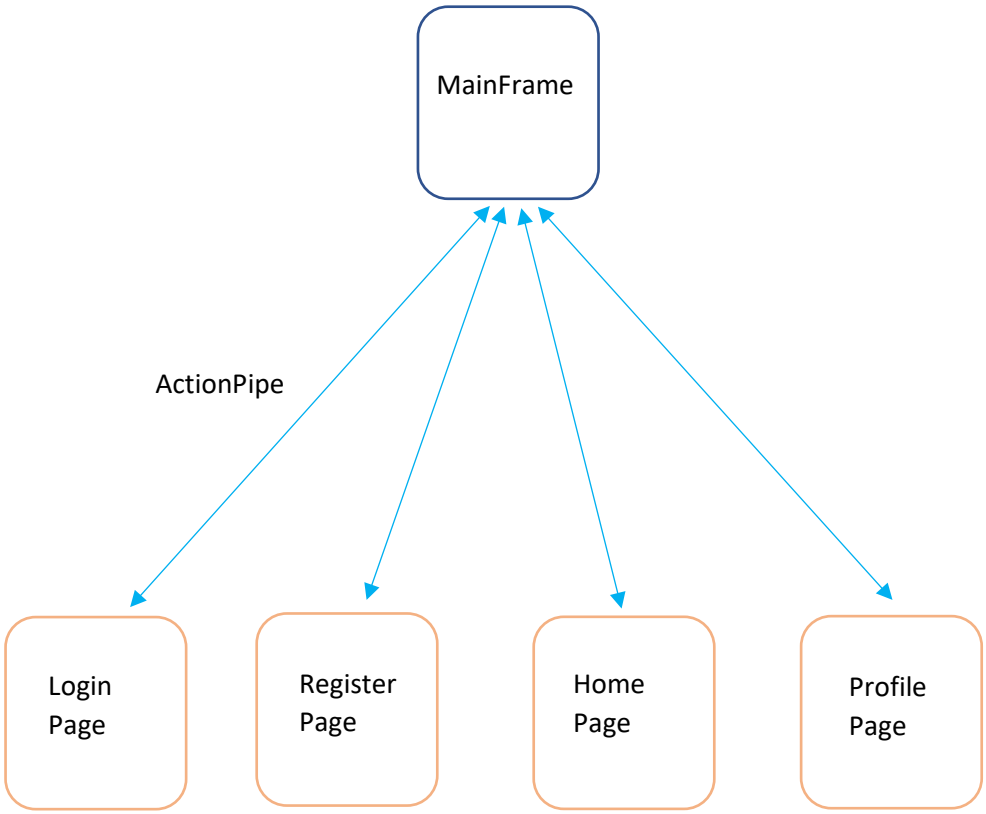
- Login
- Registrazione
- Home
- Post (serve a visualizzare un Post -> nella Home si ha solo una "preview" -> vedi sezione "Post")
- Profilo
- Discover (serve per trovare nuovi utenti)
- Wallet (serve a visualizzare lo storico delle transizioni)
- Comments (serve a visualizzare i commenti per quesl post)
- Followers

Per passare da una pagina all'altra, l applicativo fa uso del MainFrame. Questo oggetto estende la classe JFrame nativa di Java e intercambia le pagine a seconda di dove l'utente vuole andare.



Per intercambiare le pagine, il MainFrame fa uso della ActionPipe.

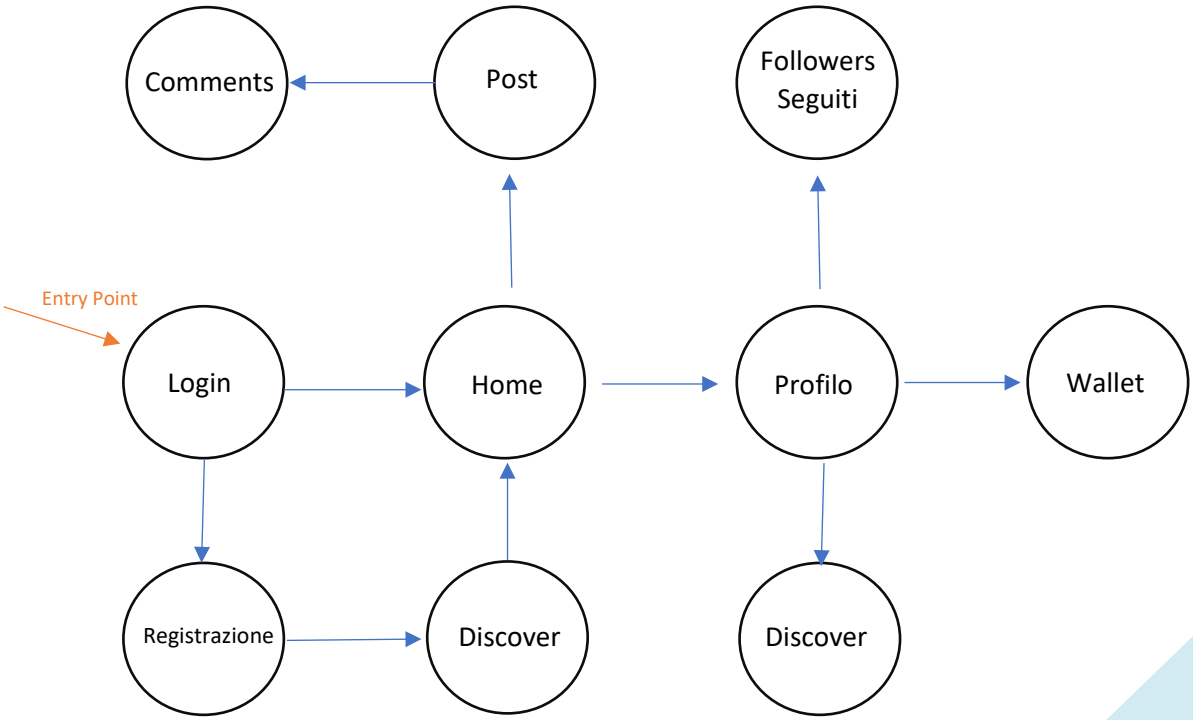
In pratica, il thread principale del MainFrame si mette in ascolto sulla ActionPipe e, quando una pagina richiede di eseguire un'azione, questa viene inserita nella ActionPipe e il MainFrame esegue l'azione richiesta.



La ActionPipe fa quindi da "tramite" tra le pagine e il MainFrame.

Attenzione: il MainFrame può eseguire la solita azione in modi differenti! Questo perché i componenti e i banner sono condivisi tra le pagine e quindi è necessario svolgere azioni differenti a seconda della pagina corrente.

Di seguito l automa a stati finiti dei possibili passaggi tra le pagine.



Per fare un esempio che riassume un po' tutto, la prima pagina che l'utente vede è la pagina di Login, quindi:

- Il MainFrame si mette in ascolto sulla ActionPipe
- L'utente inserisce username, password e preme il pulsante "Login"
- La LoginPage inserisce sulla ActionPipe l azione "PERFORM LOGIN"
- Il MainFrame esegue quindi l'azione richiesta contattando il Server e, eventualmente, "switchare" la pagina di Login con quella della Home

Possibili miglioramenti e conclusioni

Sul Database, le informazioni sui Post e gli utenti vengono tenuti tutti in memoria. Essendo un social è facile capire che questa soluzione non è per nulla efficiente, quindi sarebbe utile utilizzare un meccanismo di caching che va ad attenuare l enorme costo esponenziale tenuto per memorizzare questi dati.

Il caching, tuttavia, andrebbe fatto anche sul grafo (cosa non banale) ma ai fini del progetto ho reputato superfluo e non necessario dover implementare questa caratteristica, in quanto estremamente difficile e complicata visto che una semplice politica FIFO non è adeguata per un social network.

I Rewin di un utente, a differenza dei Post, vengono scaricati sempre tutti (quelli già presenti lato Client vengono scartati). Questo perché sono stati gestiti come Post normali, invece andavano trattati con classi apposite. Per mancanza di tempo, non ho potuto fare di meglio ma comunque il concetto da applicare è analogo a quello dei Post