CMSC 433 Project 4: The Social Network

Due April 24th (Friday), 11:59:59pm

Goal: In this project you will simulate a basic social network using a client-server model with Java RMI. The implementation details for the network are left to you, and you may use any default Java library classes you need.

Getting started: First start the server by running `Network.java`, then start the client by running `Client.java`. Both will need the following VM argument: `-Djava.security.policy==java.policy`, which refers to the policy file included with the skeleton. If the default port specified in `Client.java` and `Network.java` do not work on your machine, change them to an open port and make sure the process isn't blocked by a firewall.

Givens: For simplicity, you may assume that only one `Client` is interacting with the `Network` at a given point in time (multiple `Clients` may interact with the `Network` over its lifespan, however).

Classes: A brief description of the classes follows below. The sections of these classes which require implementing code will be marked with `//TODO: IMPLEMENT CODE HERE`. See the Javadocs in each class for more detail.

- `Network.java`:
  Represents the server for the social network. Allows `Users` to sign up, add/remove friends, and post messages to the walls of other `Users` who have the poster as a friend ("friending" is asymmetric, e.g. Alice can have Bob as a friend, but Bob doesn't necessarily have to have Alice as a friend). Your design for the network should account for different `Client` behaviors and prevent them from abusing the system.
- `NetworkInterface.java`:
  Interface required for the `Network` to interact with the `Client` over RMI as a remote object.
- `Client.java`:
  The client in this model is analogous to a web browser, app, or third-party service in that it acts as a way to interface with the network.
- `User.java`:
  Represents a person's data on the social network, specifically their username and display name. Since this class contains "public" information which can be viewed by any `Client`, information such as friends and password should be handled by the `Network`. `Users` should be uniquely identified by their information.
- `Post.java`:
  Represents a text message from one `User` to another. `Posts` are aggregated for the receiving `User` on their "wall" (`Users` can also post to their own walls). This class contains "public" information which can be viewed by any `Client`. `Posts` should be uniquely identified by their information.
- `Handshake.java`:
  When a `Client` interacts with the `Network`, it is expected to call `getHandshake()` from the `Network` to receive a `Handshake` object and call `execute()` to get a result. The time it takes to compute the `Handshake` result acts as a "tax" on transactions and deters `Clients` from spamming. The `Client` then sends the result along with other appropriate information when it requests an action, and the `Network` will only process it if the result matches a currently pending `Handshake`. To prevent `Clients` from storing `Handshake` results, the `Network` should seed the `Handshakes` such that the probability of reusing a `Handshake` is low.

Server behavior: Most of your implementations will be in `Network.java`, and should anticipate potentially abusive `Client` behavior. To this end, all methods which request changes on the server are required to provide a `User` and accompanying `password` to authorize. In addition, every single method call has a parameter called `result`, which is the computation output of a `Handshake` (see above). The methods to be implemented are:

- `getHandshake()`:
  Should return a new `Handshake` seeded such that consecutive calls to this method have a low probability of yielding the same `Handshake` result. (The `Network` should also have some way to store the results of pending `Handshakes` to verify if a result sent by a `Client` is valid. Note that in an actual system these "pending results" would likely expire over time, but for this implementation it is fine for an unmatched result to remain in the system indefinitely. If a match occurs though, it should be removed from the pending set on the `Network` to prevent its reuse.)
- `logIn(username, password, result)`:
  Should return the `User` corresponding to the provided `username` and `password` if they are correct. If they are not or the `result` is invalid, this method should return `null`.
- `register(username, password, name, result)`:
  Should create a `User` corresponding to the provided `username`, `password`, and `name`. If the `username` is already registered or the `result` is invalid, this method should return `false`. Note that if successful, a `Client` still needs to call `logIn()` to obtain the `User` object.
- `unregister(username, password, result)`:
  Should remove the `User` corresponding to the provided `username` and `password`, from all relevant data structures so that they can no longer be logged in, their wall can no longer be obtained, and they are no longer on any `User`'s friend list. If a corresponding `User` does not exist or the `result` is invalid, this method should return `false`. (It is fine for `Posts` made by an unregistered `User` to keep their attribution, as the `User` object would be useless at that point.)
- `setPassword(user, password, newPassword, result)`:
  Should change the `password` for the `user` to `newPassword`. If the `user` and `password` do not match or the `result` is invalid, this method should return `false`.
- `addFriend(user, password, friend, result)`:
  Should add `friend` as a friend of `user` (you will need to implement a way to track this). If the `user` and `password` do not match, `friend` is invalid, `user` and `friend` are the same, `friend` is already a friend of `user`, or the `result` is invalid, this method should return `false`.
- `removeFriend(user, password, friend, result)`:
  Should remove `friend` as a friend of `user`. If the `user` and `password` do not match, `friend` is not a friend of `user`, or the `result` is invalid, this method should return `false`.
- `addPost(user, password, post, result)`:
  Should add `post` to `owner`'s wall (you will need to implement a way to track this). If the `user` and `password` do not match, the `post` is `null`, the `user`/`poster` is not on the `owner`'s friend list (if the `poster` is not posting to their own wall), `post` is already posted to `owner`'s wall, or the `result` is invalid, this method should return `false`.
- `deletePost(user, password, post, result)`:
  Should remove `post` from the `user`'s wall. If the `user` and `password` do not match, the `post` is `null`, the `post` is not on the `user`'s wall, or the `result` is invalid, this method should return `false`.
- `getUser(username, result)`:
  Should return the `User` corresponding to the provided `username`. If no such `User` exists or the `result` is invalid, this method should return `null`.
- `getFriends(user, stepsRemoved, result)`:
  Should return a `HashSet` of `Users` corresponding to the friends who are `stepsRemoved` connected to `user`. For instance, `stepsRemoved = 0` should return only directly connected friends, while `stepsRemoved = 1` should return all friends and friends of friends, etc. The `user` should not be included in the resulting list. If `stepsRemoved < 0` or the `result` is invalid, this method should return `null`.

- `getWall(user, result)`:
  Should return a `List` of `Posts` in chronological order corresponding to the wall of the provided `user`. If and only if no such `user` exists or the `result` is invalid, this method should return `null`.

Additionally, you may find it necessary to modify the specified methods in `User.java`. You may test your program by placing commands in `Client.java`.

Testing: We will test your implementation of `Network` with our own `Clients` and check the resulting data. There will NOT be public tests on the Submit Server. Sharing of tests for this project is encouraged.

Submission: Submit a .zip file containing your project files to the Submit Server.