INFO 341 Project TWIT - Part 2

Due: Friday November 16, 2012 by 10:00pm

During this project you will be building a peer-to-peer version of a microblogging application. An example of microblogging is Twitter.

For Part 2 of this project you will extend the code you wrote in Part 1 so that it can do three critical things: maintain a persistent user profile, maintain a list of individuals who are "following" the local user (peer), and process incoming requests from remote peers. Part 2 of the project has been broken into 4 tasks. When you have completed all of the tasks you will have a solution to Part 2.

You may use the code from Part 1 provided by the instructor and build your own Part 2 based on the instructor's solution to Part 1. However, if you choose to use the instructor's code your solution to Part 2 will be charged 10% of the total possible points for Part 2. If you were not successful with Part 1, this will allow you to continue working on the project to complete Part 2. Note that you can *read* the instructor's solution for free.

Part 2, Task 1 – Persistent User Profile

The software running at the local peer needs to maintain some persistent state. We will keep a "profile" of the local user. A profile is a set of named fields with field values. A user profile for our system will consist of six fields:

`UserName` – the user's name, an arbitrary string.
`UserID` – a unique ID number assigned to each user in hex.
`UserMotto` – a motto that the user can set, a string.
`UserURL` – a URL to the user's website.
`DNSSDServiceName` – a string service name to be used in Part 3 of the project. For now, you should set this String to the value "`twit`".
`ProtocolPort` – the protocol port number that is used by this peer to listen for incoming connections.

The user profile should be an editable text file, stored in the local directory with your other Java files. The file should be called "`profile.txt`".

You should modify your `Console` class so that it can accept five new commands:

`setName <username>` – to set the username for the current user profile.
`setID <id>` – to set the unique ID for the current user profile.
`setMotto <motto>` – to set the Motto for the current user profile.
`setURL <url>` – to set the URL for the current user profile.
`showProfile` – should print/list all of the fields of the current profile.

There are some special characters that should **not** be allowed. The "#", ",", "=" and "@" (hash, comma, equal, at sign) should not be allowed in any of the username, userID, motto, nor URL strings. This restriction simplifies parsing and processing. In a commercial system, this restriction could be removed.

Modifications made to the user profile through commands at the `Console` should persist from one run to the next. That is, you need to make sure you can save and reload the profile fields each time the code is run. A sample `profile.txt` file is available from the project web page. The structure of your profile file does not have to be identical to the sample file.

Part 2, Task 2– Persistent Follower List

Since we're building a peer-to-peer microblogging system (similar to twitter), each local peer needs to remember who is "following" it so that when a status message should be distributed it can look for those peers on the network and deliver the status message. Thus, you need to create a mechanism for maintaining a persistent list of peers who are following the local peer.

For each follower, the local peer needs to minimally maintain a friendly representation as a String. For convenience, we will call the friendly representation "userName". A userName is not guaranteed to be unique, so the local peer should also maintain a special unique ID for each follower. In Part 3 we will describe how those unique IDs are assigned, for now you can use any value you like as long as your follower list enforces uniqueness for the ID.

You should modify your `Console` object so that it can accept an additional command:

`showFollowers` – should print/list all of the remote peers who are following the local peer.

The entries in the followers list come from protocol commands that are received by the local peer from remote peers. The protocol commands that add and remove followers is specified in Part 2, Task 3 – Building a Protocol (below).

The follower list should be saved as a user editable text file, stored in the local directory with your other Java files. The file should be called "`followers.txt`" to indicate what it is and that it is user readable. Your persistent follower list should allow the insertion and removal of followers. And, like the profile, you need to be able to save and reload the list each time the code is run. A sample `followers.txt` file is available from the project web page. The structure of your followers file does not have to be identical to the sample.

Part 2, Task 3 – Building a Protocol

You need to implement a protocol for the peer-to-peer service. A protocol specifies exactly how two peers are to communicate with each other over the network. The protocol specifies what commands a peer should be able to send/receive to/from another peer as well as how a peer should respond to a command that it receives.

The following description of the protocol is based on what would happen for the full peer-to-peer implementation of the protocol. However, in Part 1 you built a simple chat application that was designed to communicate with exactly one peer. For Part 2 you will build the protocol with the assumption that there is only one peer active during each session. This is a simplification to make development easier during Part 2. You should assume that the active peer is the one that you connect to with the `connect` command from the console. This will be expanded in Part 3 to allow many possible peers at once. The protocol specification is written assuming many peers. However, since you have only one peer for this part (Part 2) you can make some simplifying assumptions.

The protocol has three basic commands that are sent as text strings and which are structured to be one line and one line only. The commands are `follow`, `tweet`, and `profile`.

`follow [remove] userName#userID\n`
The follow protocol command has two forms. In the first form, the protocol command is requesting that the specified user (`userName#userID`) would like to receive `tweet` messages from the user at the local peer. This is a persistent follow and the `userName#userID` pair should be maintained at the local peer in the follower list. An example `follow` command might be:

```
follow Tommy's Tweets#123456ABCD
```

This is a request that the local peer send future status messages to the user "Tommy's Tweets" with a unique `userID` of "123456ABCD". If a user with the same unique `userID` already exists in the list of followers then the user should not be inserted twice. However, if the `userName` is different then the `userName` should be updated to the new username. The peer that receives the `follow` command does not need to reply to or acknowledge the command.

The second form the `follow` command includes a command modifier "`remove`" which indicates that the specified user should be removed from the list of followers. That is, this user should be removed from the persistent followers list and should no longer receive status messages. An example of this command might be:

```
follow remove Tommy's Tweets#123456ABCD
```

This is a request that the local peer stop sending status messages to the user "Tommy's Tweets" with a unique `userID` of "123456ABCD".

If the user indicated after the `remove` command modifier is not in the followers list, then the command should do nothing.

`tweet userName#userID#StatusMessageString\n`
The `tweet` protocol command is how a peer sends or receives a status message. When the local peer receives a `tweet` protocol command, the text of the `StatusMessageString` should be shown to the local user as coming from the user with the specified `userName#userID`. The peer receiving the `tweet` does not need to reply to the `tweet` command in any way.

When the local peer wants to send a status message, it should prepend the local `username,` a hash character "#", the `userID` and a hash character "#" to the status message and send that message to all of the followers in the persistent follower list who are currently online.

The text of the `StatusMessageString` should be limited to a maximum of 140 characters. If a tweet arrives where the `StatusMesssageString` is greater than 140 characters, the local peer should truncate the text to 140 characters before the status message is displayed. But a properly behaved client should never send more than 140 characters in a `StatusMessageString`.

When displaying the `StatusMessageString`, the display should show that the message came from the user with `userName`. The local peer can decide individually whether or not it will also display the `userID`.

`profile [request userID | `*`<profile data>`*`]\n`
The `profile` protocol command has two forms. If the command `profile` is followed by the command modifier `request` and a `userID`, then this is a request from a remote peer for the profile of the user at the local peer. The `userID` indicates the online peer making the request. A sample profile request might look like:

```
profile request ff30116d00
```

This is simply a profile request by the user with the `userID` of `ff30116d00` for the profile of the user at the local peer.

In response to the `profile request`, the local peer should send a response starting with the protocol command `profile` followed by *<profile data>*. Profile data consists of the four user profile fields, `UserName`, `UserID`, `UserMotto`, and `UserURL` as a comma separated list of `key#value` pairs. For example in response to a `profile request` command, a local peer might send (note that the line is long for this printout, it is sent as a single line, no carriage returns and no newline characters in the middle):

```
profile UserName#Tommy's Tweets,UserID#123456ABCD,
UserMotto#WTF?,UserURL#http://www.surf-righteous-waves.net
```

The key#value pairs can occur in any order. If a field in the local profile is empty, then the corresponding key#value pair should be omitted from the response.

Note that there are several aspects of this protocol that have been underspecified. For example, what should happen if a "`profile`" response is improperly formatted? There are other gaps in this protocol. The protocol for this project is not meant to be a fully elaborated and complete protocol. This is just a small example for the class. When implementing this protocol you should check for error conditions and do something reasonable if your local peer receives an improperly formatted protocol command.

Part 2, Task 4 – Modifications to the Console

There are a number of modifications that you need to make to your `Console` in order for all of this to work. Many of those modifications are described above in the sections that are relevant to the specific commands, but the following restates the requirements for those commands. Your `Console` needs to be able to process:

`setName <username>` – to set the username for the current user profile.
`setID <id>` – to set the unique ID for the current user profile.
`setMotto <motto>` – to set the Motto for the current user profile.
`setURL <url>` – to set the URL for the current user profile.
`showProfile` – should print/list all of the fields of the current profile.
`showFollowers` – should print/list all of the remote peers who are following the local peer.

There are additional changes that you should make to the console. Now that you have implemented the protocol, you should modify the `Console` to process commands that require the protocol in order to communicate to a remote peer. For Part 2, you are only connecting to one peer, but in this Task you need to modify the Console to send and receive protocol commands by implementing the following commands.

`status StatusMessageString` – the text following the `status` command at the command line is set to all followers as a tweet in the protocol. In Part 1 the console assumed that any text entered at the command prompt, which did not start with a command, was text to be sent to the remote chat. Messages sent to the remote peer now need to be preceded by the status command on the console command line.
`getprofile` – when this is entered at the console command line, you should send the profile request protocol command to the remote peer, wait for the profile response and show that profile.
`follow` – should allow you to start following the specified user of some remote peer. This should add the local user/peer to the follow list of the remote peer.
`unfollow` – should allow you to stop following the specified user of some remote peer. This should remove the local user/peer from the follow list of the remote peer.

Unlike Part 1 of the Project, you can now assume that every line entered on the command line of your `Console` will begin with a command. If a line does not begin with a command,

then your `Console` should indicate that the line does not start with a command and list the available commands.

Turning In

Your Java code should compile and run correctly from the command line. If you use a development environment you should make certain that your code will compile and run from the command line.

All of your Java classes should be in a `package` that is named by your lastname and the last three digits of your student ID. So if your last name is "McStudent" and the last three digits of your student ID were "820" your code would include a statement like:

```
package mcstudent820;
```

If you have your Java CLASSPATH variable set correctly you would be able to run your Console from the command line by typing:

```
java mcstudent820.Console
```

You will turn in your Java source files, your profile.txt, and followers.txt files as one Zip file. Since packages are assumed to be in directories that are named the same as the package (e.g., in the example above, your code would be in a directory named "mcstudent820") you can just Zip that directory and submit the single file through the course Catalyst Dropbox. Absolutely no assignments will be accepted through email.