THE UNIVERSITY OF SYDNEY

COMP2121 Assignment 2

Due: 10AM, October 23rd, 2013

The goal of this project is to implement a basic Peer-to-Peer (P2P) Twitter application. Like the original Twitter application, the user is allowed to post a status message of at most 140 character. While the original Twitter uses a centralised server (farm) to store the requests, your P2PTwitter program will communicate with other P2PTwitter programs running remotely without using any centralised server. Note that the subsequent tasks below only indicate the expected chronological progression; only one solution that solves all tasks is expected to be submitted.

Important requirements:

- Note that for this project, only the JDK packages can be used, starting with **`java*`**.
- The code should be written in Java 7 or earlier. (Make sure it works in the lab rooms on Linux.)
- All P2P servers should be listening on port 7014 for incoming requests.

Task 1 P2P Twitter

As the program is P2P, it comprises a client thread and a server thread. (You may implement more threads.) The client will send requests to the server of remote P2P programs whereas the server will serve the requests sent by remote P2P programs. As all servers must use the same port (7014), testing is only possible if at least two P2P programs are running concurrently on remote machines.

The main file (containing the main method) is **`P2PTwitter.java`**. The file containing the class of the server (resp. client) will be named **`P2PTServer.java`** (resp. **`P2PTClient.java`**). Two other classes might be necessary (they are optional so you may omit them if your program works without): a class **`Communication`** handling the communications and a class **`Profile`** to store the information regarding other participants of the system. All classes will be stored in the same default package (no package header in java files) and all files should be located in the same **`src`** folder. Hence, to compile the code one would simply use the following command:

```
$ javac -cp . *.java
```

Of course, all present **`.java`** files should be correct. Therefore, beware of not forgetting dummy files that do not compile so that this command does not return errors. The **`src`** folder contains all **`.java`** files and the properties file and will be placed into a folder named with your unikey. Please zip this latter folder and submit the resulting archive by the deadline given above.

Task 2 Communication protocol

The communication protocol is UDP, and you will use DatagramSocket (http://docs.oracle.com/javase/7/docs/api/java/net/DatagramSocket.html) for communication as messages get sent repeatedly. The encoding format of String into bytes is **`ISO-8859-1`**. Statuses must be encapsulated in a message containing also the unikey of the sender in the following form where the participant with unikey **`bob1234`** sends status **`Not too bad today.`**:

```
bob1234:Not too bad today.
```

Note that the unikey is separated from the status by a single colon but without spaces; it is assumed that unikey cannot contain colon, however, you should make sure that a colon in a status should be converted into "\`:`" before transmission and decoded appropriately upon reception. Message are sent periodically to all participants every X milliseconds where X is a random variable whose values are integers between 1000 and 3000, to avoid flooding the network.

Task 3 Output format

The program output should respect the following format for the program to be tested automatically.

The user starts his/her own local program by giving his/her unikey, then the program asks his/her current status and lists the status of each participant. More precisely, the user must first start the program by giving his unikey (e.g., **`bob1234`**) as a parameter:

```
$ java P2PTwitter bob1234
```

Then the program must immediately ask the user for his/her status by outputting **`Status:`** on the next line. After the user type some status (e.g., "Not too bad today.") and 'Enter', the program should output the information about the status of other machines, where status lines start with '#' and are followed by the pseudo of the user and his/her unikey (or **`myself`** in case of the current user) separated by single spaces, then a colon followed by the corresponding status.

```
Status: Not too bad today.
### P2P tweets ###
# bob (myself): Not too bad today.
# alice (alic0123): just voted...
### End tweets ###
```

In the example above, there are two participants in the system. The user on the current machine has pseudo **`bob`** whereas the remote participant has pseudo **`alice`** and unikey **`alic0123`**. A status should not be empty (resp. longer than 140 characters), or the peer would write "Status is empty. Retry." (resp. "Status is too long, 140 characters max. Retry.") on the error output. In case, the local peer has never heard from one of the other peer, it should write a default status like not yet initialized and encapsulate the whole line using square brackets. (Note that encapsulating the entire line is important to differentiate the case of not receiving anything from Alice from the case where Alice sets her status to "[not yet initialized]" herself.) For example, if Bob did not hear from Alice yet it outputs:

```
Status: Not too bad today.
### P2P tweets ###
# bob (myself): Not too bad today.
# [alice (alic0123): not yet initialized]
```

```
### End tweets ###
```

The program should execute continuously, re-asking the user to enter his/her new **`Status`** before re-printing the updated statuses. The program will be terminated manually by typing **`Ctrl-C`** (from a terminal).

Task 4 Testing

You will have to test your application by running it on two or more machines that you can control. The information regarding the participants should be stored in a properties file copied on each participant machine.

```
# participants.properties

participants=peer1,peer2,peer3

peer1.ip=192.168.1.10
peer1.pseudo=paul
peer1.unikey=paul0123

peer2.ip=192.168.1.43
peer2.pseudo=bob
peer2.unikey=bob1234

peer3.ip=192.168.1.24
peer3.pseudo=steve
peer3.unikey=stev1234
```

While no code should be shared, note that you are allowed (and encouraged) at the final stage of the development to test with others whether P2P programs from different authors interact correctly.

Please, pay attention that the java program should find the properties file in the current folder – do not use an absolute path in the java code, otherwise it will not work on other machines and tests will fail.

Task 5 Fault tolerance

A peer-to-peer network is typically subject to churn, as peers may leave the network and new peers may join at any time. For this reason, P2PTwitter should tolerate failures. If messages get lost (remember that we use UDP) or some peers fail, then we would like to ensure that non-faulty peers work correctly. In particular, we assume that the user of P2PTwitter is aware that using a P2P application may be unreliable so your program should not print informative error messages or raise exceptions if some messages are not received. However, as the user is concerned about the relevance of the message his peer prints, it is important to identify the staleness of messages and appropriate actions must be taken.

More precisely, if a peer i does not receive any information from another peer j during 10 seconds, then i updates his own local copy of j's status to "idle" and prints Alice's information in a line encapsulated by square brackets. (Note that encapsulating the entire line is important to differentiate

the case of not receiving anything from Alice from the case where Alice sets her status to "[idle]" herself.) If a peer *i* did not receive any information from another peer *j* within the last 30 seconds, then it will stop printing the information corresponding to *j* (but *i* will keep printing the information regarding the responsive peers).

For example, if Bob received the last message from Alice seconds ago (with $10 \leq$ $<$ 30), then Bob's peer would simply print:

```
Status: Not too bad today.
### P2P tweets ###
# bob (myself): Not too bad today.
# [alice (alic0123): idle]
### End tweets ###
```

For example if Bob received the last message from Alice seconds ago (with $\geq$ 30), then Bob's peer would simply print:

```
Status: Not too bad today.
### P2P tweets ###
# bob (myself): Not too bad today.
### End tweets ###
```

A peer should be able to recover without the need to restart any other peer. As soon as Bob receives a new message from Alice, he starts printing Alice's information again as usual. For example, if Alice sets her status to "I'm back", Bob's peer would print:

```
Status: Not too bad today.
### P2P tweets ###
# bob (myself): Not too bad today.
# alice (alic0123): I'm back
### End tweets ###
```

Note that the local peer status is never stale and is always printed normally (as long as the local peer runs).

Assessment

The code will be tested using correct P2PTwitter programs, hence, it is crucial that you make sure your program does exactly what is described above. In particular, note that a single typo in the request from the client leads the server to not be able to understand the request.

The program must compile and run correctly.

Finally, note that the assessment will be done independently from the instructions you may provide and the technique you used to test your own program. Specifically, independent correct P2PTwitter programs should interact (exchanging status messages) with yours correctly, and vice versa.

A global demo will be done right after the submission deadline, in the lab of week 12. Please note that you will have to show the behaviour of your program during this lab to the tutor. The tutor will provide you with additional information at the beginning of the lab. Note also that your work will be assessed later based on the code you submitted and the features that were observed during the demo lab.