CS 455/655 – Computer Networks
Fall 2017

Programming Assignment 1: Implementing a Social Media App
Due: Thursday, October 5th, 2017 (1pm)

To be completed individually. Please review Academic Honesty noted in the syllabus.
This assignment is part of BU CS 455/655 material and is provided for educational purposes. Please do NOT share or post this assignment handout or your solution, on any public site, *e.g.* github. Of course, you are not allowed to share your solution with classmates.

Overview

In this assignment, you will be putting together knowledge of writing multi-threaded programs (server and client), broadcast and multicast communication, and application-layer protocols. By the end of this assignment, you will have created a social media app consisting of one server and multiple clients. The clients all connect to the server who is then the means through which the clients are able to receive status updates from each other. A client (user) can post a status update that gets pushed as notification either to all other users (broadcasting) or to only a subset of users who are the user's friends (multicasting). We have split this entire assignment into four parts so as to walk you through progressively more sophisticated functionality. Only CS 655 students are required to complete the last part (Part IV).

We strongly recommend that you use Java, however you can use the programming language of your choice (C, C++, Python). Skeleton Java code is provided at http://www.cs.bu.edu/fac/matta/Teaching/cs455/F17/SM-handout

Part I: Writing a Client-Server Application

Overview:
For this part, you will implement a client and a server that communicate over the network using TCP. The server simply prints out the message that it receives from the client. Here is what the server and client must do:

- The server should accept, as a command line argument, a port number that it will run at. **To run your server on our csa1, csa2, or csa3.bu.edu, we have opened up ports 58000-58990 so please pick a port in this range for your server.** After being started, the server should repeatedly accept an input message (text string) from a client. If the message is "valid", i.e., it starts with the string #status followed by the actual user status, the server prints out the user status on the screen. For example, a valid protocol message sent to the server over the network could be #status I just woke up! The server would then send back a #statusPosted message to the client and prints out 'I just woke up!'

- The client should accept, as command line arguments, a host name (or IP address), as well as a port number for the server. Using this information, it creates a connection (using TCP) with the server, which should be running already. The client program takes a user status text message as input. This user text is used to form a protocol message to be sent to the server of the form #status <user status text> as described above. Once the client program receives a #statusPosted message from the server, it closes its side of the connection and exists.

In Part I, the exchange protocol between the client and server defines two message types / formats that start with keywords #status and #statusPosted. A skeleton Basic/Server.java and Basic/User.java are provided for you.

Implementation:
Use the socket interface. Familiarize yourself with PrintStream and BufferedReader for sending and receiving through a socket over the network. The goal of this part is to familiarize you with the socket interface. You don't need to implement multi-threading in this part.

What to turn in: The programs you submit should work correctly and be well documented with a record of the information exchanged between your client and server. It would be a good idea to test your client program with the server program implemented by a classmate, or vice versa. This is possible because your programs should adhere to the same exchange protocol described above (and below for the other parts of this assignment). This is also an interesting way to see how protocol entities can "interoperate" if they accurately implement the same protocol specification. *You must, however, implement both the client and server on your own*. See the syllabus handout for guidelines on electronic submission using *gsubmit* to submit your documentation and Part I client and server programs under a "pa1part1" folder—to save trees, you need not submit a hard copy of your program listings (code).

Note: You can develop your programs on non-CS machines, however, you ultimately need to port and test your programs on our CS Linux machines (i.e., csa1, csa2 or csa3.bu.edu) to ensure they will be graded correctly.

Part II: Implementing a Broadcast Status Update Using Multi-threading

Overview:
For this part, you will use multi-threading to implement a social media app with one server and multiple clients. We assume here that all clients are interested in getting notified of any status update posted by any one of them. The clients and server will communicate over the network using TCP. Here is what the server and client must do:

- The server should accept, as a command line argument, a port number that it will run at and maximum number of clients it can accept. For simplicity we would like you to specify the maximum number of clients as 5 (it should not be hard coded though; use a global variable). After a client connects to the server and sends a status update message, the

server should broadcast this message to all other clients. The server has to also announce a new client joining or leaving to all other clients.

- The client connects to the server using the server's port number and enters its username. Once the connection is established, the client should be able to send and receive status messages as well as be able to quit the social media app.

Implementation:

You should use multi-threading to allow the server to communicate with all clients. We have provided a file called TestThread.java for you that implements a simple multi-threading example that you can use as an example for how to implement multi-threading. We also suggest reading through this tutorial.

Server.java:

Create a file called Server.java. This will handle all the tasks that the server needs to do. Similar to how we have done it in TestThread.java, we suggest creating a class called userThread that will handle all individual user threads and then creating a private array of user threads in the Server class. The userThread class should extend the Thread class to handle a single instance of a user communicating with the server and, consequently, with other users. As a result, the instance of the userThread should have access to all other threads that the server has access to. This is what the server will use to send broadcast (and later, multicast messages) from one user to all other users (or a subset of users, i.e., only the user's friends).

Once a client decides to join the social media app (connecting using the server's port), the user application asks the user for his/her name. Once the user enters his/her name, the client sends a protocol message to the server of the form #join <username>. If server can entertain the user, the server will reply back to the client with protocol message #welcome, and the server will also send a broadcast protocol message #newuser <username> to all other users informing them that a new user has joined. If the server is busy (i.e., number of users reaches a threshold, say 5), the server replies back to the client with protocol message #busy. All user programs, upon receiving a new user message, should print that a new user has joined the social network along with the name of the user. When a client sends a (status update) message, #status <user status text> (which it really sends to a thread in userThread), the server must broadcast this message in protocol message #newStatus <username> <user status text> to all other users. User programs, on receiving this message, will print '[<username>] <user status text>'. The server should also reply back to the sender with protocol message #statusPosted telling the client that the status has been posted.

Since we are in the multi-threading universe, we must make sure that when a single thread is trying to send a message to all other users, the execution of sending that message (and accessing the shared array of user threads) does not get interrupted. This becomes very simple in Java—we can just use a synchronized statement. Essentially, when a thread executes a synchronized statement, the statement under the

synchronized header is executed to completion and no other thread can interrupt it. As we see when running TestThread.java, when we have two threads running, they interrupt each other's execution of commands (thus the alternating print statements). By using a synchronized statement, we tell all other threads to wait for the execution of what is under the synchronized header to finish before they can interrupt.

When a user wants to leave by typing ‘Exit’, the client needs to somehow signal to the server that this user wants to exit the app. To do this, the client program then sends the protocol message #Bye. The server will then immediately send a broadcast message saying that the specified user is leaving. Specifically, the server program sends a protocol message to each client of the form #Leave <username>. The server also acknowledges to the departing client by sending a #Bye message, so that the client closes the connection on its end. Also, the server needs to set the user thread to null (i.e., remove it from the array of threads) and close the socket so that the server may accept other users.

User.java

Create a class titled User.java. This class should extend the Thread class. In the main method, we establish a socket connection with the server at the server’s port number and then start a User listening thread before accepting (broadcast or multicast) status messages from the user. While the listening thread is running, it listens to protocol messages from the server and prints appropriate messages for the user.

When a client joins the social network (by sending #join <username> protocol message to the server), the server replies back with #welcome message. After receiving this message, the client should print that connection has been established with the server. If the server is busy, the server replies back with #busy protocol message in which case the client should inform the user that the server is busy and the user should try later.

If the server acknowledges that the user’s status message has been successfully posted (i.e., a #statusPosted message is received from the server), then the client should print out that is the case.

If the client receives #newuser <username> message from the server, the client should print that a new user has joined.

If the client receives #newStatus <username> <user status text> message from the server, the client should print the username along with the user’s status on the screen.

If the server informs the client that some other user has left (i.e., a #Leave <username> message is received from the server), then the client should print out that is the case.

Finally, if the server has signaled that the connection should be closed (i.e., the server sent a #Bye message), then the user listening thread closes the connection.

What to turn in: Follow guidelines on using *gsubmit* to electronically turn in your Part II client and server program listings under a "pa1part2" folder. Again, to save trees, you need not submit a hard copy of your program listings (code). A skeleton Broadcast/Server.java and Broadcast/User.java are provided for you.

NOTE: To help you develop your code, you can find a server, which implements parts II, running on csa2.bu.edu on port **58992** that you can use to first test your user/client code, *before* you implement *your own* server. Implementing your own client first and testing it against our server should be a good strategy. Our server is configured to accept a maximum of 100 users, so if you get the error message "Server too busy", please try again later. *Remember that you will need to also implement the server yourself!*

Part III: Adding Friendship / Multicast Capability to the Social Media App

Overview:

In this part you will extend Part 2 by adding multicast capability. A client should be able to post status updates to only her friends (a subset of all clients). Thus clients should be able to add each other as friends.

Implementation:

A user specifies a friend by typing '@connect <username>', where <username> is the name that this user wants to friend. To send this friend request, the client program sends the server a protocol message of the form #friendme <username>, where again username is the name of the friend from whom the client wants to receive status updates. Once the server sees this, it will send the friend request to the specified username using protocol message #friendme <requesterUsername>, where <requesterUsername> is the user name of the requester. The friend request receiver will type '@friend <requesterUsername>' to accept a friendship request. The response back to the server will be the protocol message #friends <username> if the user accepts to be friends with username. Upon receiving this response from the client, the server should update the list of friends of both clients and send a confirmation message #OKfriends user1 user2 to both of them. If the user does not want to accept the friend request, the user can type in '@deny <requesterUsername>'. The protocol message sent to the server for rejecting a friend request is #DenyFriendRequest <username>, where username is the name of the user whose request is denied. Upon receiving a negative response from a client, the server replies back to the requester with #FriendRequestDenied <username>, where username is the name of the user rejecting the friend request. *It is assumed that the friendship is mutual, i.e., both friends will receive each other's status updates from now on*. Each user program should confirm this friendship by printing out a message 'user1 and user2 are now friends' on the console of each user. If a user denies a friend request, the requester should print that 'user2 rejected your friend request'. You should also allow users to unfriend each other by typing '@disconnect <username>'. Then the client program should send the server a protocol message of the form #unfriend <username>. A friend who is being unfriended should be notified by the server that

the friendship is terminated; the server sends her a protocol message of the form #NotFriends user1 user2. Each client should also display the message 'user1 and user2 are no longer friends' on the console of each user. Note that now with this new friendship functionality, a status update message cannot start with @.

This friendship functionality requires a modification of Server.java.

Server.java

To add this capability, when a user sends a status update message, the server needs to only output that message to this user's list of friends. The server also sends the message #statusPosted to the user who sent it so as to let her know that the message was sent. Recall that it is still important to put statements under the synchronized header.

What to turn in: Follow guidelines on using gsubmit to electronically turn in your Part III client and server program listings under a "pa1part3" folder. Again, to save trees, you need not submit a hard copy of your program listings (code).

NOTE: To help you develop your code, you can find a server, which implements part III, running on csa2.bu.edu on port **58993** that you can use to first test your user/client code, *before* you implement *your own* server. Implementing your own client first and testing it against our server should be a good strategy. Our server is configured to accept a maximum of 100 users, so if you get the error message: "server is too busy", please try again later. *Remember that you will need to also implement the server yourself!*

Part IV: Adding Group Capability

Only CS 655 students are required to complete Part IV.

Overview:

In this part you will extend Part 3 so a user can create groups among her friends.

Implementation:

A user specifies a group by typing '@add <groupname> <username>', where <username> is the name of a **friend** that this user wants to include in her friends' group <groupname>. To send this request, the client program sends the server a protocol message of the form #group <groupname> <username>. On the server side, if a group with name <groupname> does not exist, the server will create a new group <groupname> and add <username> to the group, along with the user who requested to add <username>. The server will also echo the group request message (#group <groupname> <username>) to all the members of the group. Each client who is part of the group should print out a message '<username> is now in group <groupname>' on the console of each user. Note that only group members can add other users to the group and a user can add only her friends to the group.

To send a status update to a particular group, any user who is a group member can type '@send <groupname> <status update message>', where <groupname> is the name of a group

that has been earlier created. The client sends a protocol message #gstatus <groupname> <username> <update message> to the server, where <username> is the name of the user associated with the client. On the server side, if the group exists and the user is part of the group, the server echoes the message to all the members of the group. The client, upon receiving this message, should print '[<groupname>] [<username>] <update message>' on the console.

You should also allow a user to remove another user from an existing group by typing '@delete <groupname> <username>'. Note that both users must be members of the group and the user you are removing may not be your friend, as your friend who is part of the group might have added her to the group. The client program should send the server a protocol message of the form #ungroup <groupname> <username>. The server should echo this message to all group members (including the member who is being removed). Upon receiving this message, each client should also display the message '<username> is no longer member of the group <groupname>' on the console. For simplicity, there can be a maximum of 5 groups in the system.

Note that any member of the group can add other members (who are her friends) to the group. So you might have "friends of friends" as part of the group, since your friend, who you have added to the group, can add her friends to the group. Also, once you are member of a group, you remain a member even if you are no longer friend with the user who initially added you to the group. Of course, any existing member can delete you from the group.

What to turn in: Follow guidelines on using gsubmit to electronically turn in your Part IV client and server program listings under a "pa1part4" folder. Again, to save trees, you need not submit a hard copy of your program listings (code).

Attachment 1: A Test Run of the Social Media App, Part II

Server	First User	Second User	Third User
>>java Server Usage: java Server <portNumber> Now using port number=8000 Maximum user count=5	>> java User Usage: java User <host> <portNumber> Now using host=localhost, portNumber=8000 Enter your name: User1 Welcome User1 to our Social Media App. To leave enter Exit on a new line. New user User2 has joined !!! New user User3 has joined !!! I just woke up! Your status was posted successfully! Exit	>>java User Usage: java User <host> <portNumber> Now using host=localhost, portNumber=8000 Enter your name: New user User1 has joined !!! User2 Welcome User2 to our Social Media App. To leave enter Exit on a new line. New user User3 has joined !!! <User1>: I just woke up! The user User1 is leaving !!! Exit	>>java User Usage: java User <host> <portNumber> Now using host=localhost, portNumber=8000 Enter your name: New user User1 has joined !!! New user User2 has joined!!! User3 Welcome User3 to our Social Media App. To leave enter Exit on a new line. <User1>: I just woke up! The user User1 is leaving !!! The user User2 is leaving !!! Exit