CS 455/655 – Computer Networks
Fall 2016

Programming Assignment 1: Implementing a Chat Room
Due: Thursday, October 6th, 2016 (1pm)

To be completed individually.

Overview

In this assignment, you will be putting together knowledge of writing multi-threaded programs (server and client), broadcast and unicast communication, and application-layer protocols. By the end of this assignment, you will have created a chat room 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 send messages to each other. The clients can either send messages to everyone else that is in the chat room (broadcasting) or they can send private messages to their individual friends (unicasting). We have split this entire assignment into four parts so as to walk you through how to create this infrastructure. 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/F16/Chat-handout

Part I: Writing an Echo Client-Server Application

Overview:
For this part, you will implement a client and a server that communicate over the network using TCP. The server is essentially an echo server, which simply echoes the message 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 58000-58998 so please pick a port in this range for your server.** After being started, the server should repeatedly accept an input message from a client and send back the same message.
- 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 then sends a message (text string) to the server using the connection. When the client receives the message back from the server, it prints it and exits.

In Part I, the message is simply a text string with no specific format. In other words, any text message will do.

Implementation:
Use the socket interface. Familiarize yourself with PrintStream. We do not specify any specific protocol that defines the interaction between the client and server here. 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 Chat Room Using Multi-threading

Overview:
For this part, you will use multi-threading to implement a chat room with one server and multiple clients. The clients and server will communicate over the network using TCP. Here is what the server and client must do:

- The server should be running on a certain port. 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 message, this message should be broadcasted to all clients by the server. 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 messages as well as be able to quit the chat room.

Implementation:
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. As well, we suggest reading through this tutorial.

Server.java:

Create a file called Server.java. This will handle all the things 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 ArrayList or 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, unicast messages from one user to other users.

Once a user decides to join the chat-room (connecting using the server's port), the server will send a broadcast message saying a new user has joined the chat-room. When a user sends a message (which it really sends to a thread in UserThread), the server must broadcast this message to all other users. 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 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. As we see when running TestThread.java, when we have two threads running, they interrupt each other's execution of commands (thus the mixed 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, it needs to somehow signal to the server that it wants to exit the chat-room. A protocol would specify a specific message that means a user wants to exit. In this protocol, use the message "LogOut." When a user enters that message, it signals to the server that the user wants to logout. The server will then immediately send a broadcast message saying that the specified user is leaving the chat-room. Then, we need to set all 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. As well, the server needs to signal to the running user thread to close the connection on its end. In this protocol, the server should send the message "### Bye <username> ###" to the departing client.

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 unicast messages from the user. While the listening thread is running, it simply prints what it reads in (i.e., what another user sent either as a broadcast or a unicast message). 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. A skeleton User_skeleton.java is provided for you.

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 Server_skeleton.java and a skeleton User_skeleton.java are provided for you.

NOTE: To help you develop your code, you can find a Chat server (that implements all parts II, III, and IV) running on csa2.bu.edu on port 58999 that you can use to first test your user/client code, *before* you implement *your own* Chat 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 an error message that the "server is too busy", try again later. *Remember that you will need to also implement the server yourself!*

Part III: Adding Unicast Capability to the Chat Room

Overview:

In this part you will extend Part 2 by adding unicast capability to the chat room. Clients should be able to send private messages. Private messages are sent to a specified user and are not broadcasted.

Implementation:

In terms of sending private messages, the client should type "@username" where username is the name of the friend to whom the client wants to send a message to. Once the server sees this, the server will no longer broadcast the message, but instead send the message to the specified user.

Server.java

To add this capability, we simply need to extend Server.java. First, we need to add a check when a user is joining the chat-room. Since the symbol "@" is going to be a special symbol, we need to ensure that any user joining the chat-room does not have their name start with that symbol.

Now, when a user sends a message, we must first check to see if the message begins with "@". If so, the server needs to only output that message to the specified user. We will also echo the message 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 II 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).

Part IV: Adding Friendship Capability to the Chat Room

Only CS 655 students are required to complete Part IV.

Overview:
In this part you will extend Part 3 so that clients can add each other as friends. You will modify the unicast capability so that only friends can send private messages to each other.

Implementation:
As before, the client should type "@username" where username is the name of the friend to whom the client wants to send a message. Once the server sees this, the server will no longer broadcast the message, but instead send the message to the specified user *given* that this user is a friend. To send a friend request, a client should type "#friendme @username" where username is the name of the friend to whom the client wants to send a message. Once the server sees this, it will echo the friend request to the specified username. The client who receives a friend request will respond "@username #friends" if the user wishes to be friends with username. Upon receiving this response, the server should update the list of friends of both of the clients and send a confirmation message to both of them. It is assumed that the friendship is mutual. You should also allow users to unfriend each other by typing "@username #unfriend". A friend who is being unfriended should be notified by the server that the friendship is terminated. Again, this only requires a small manipulation of Server.java.

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 a Chat Room

Server	First User	Second User	Third User
>>java ServerFriends 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.		
		>>java User Usage: java User <host> <portNumber> Now using host=localhost, portNumber=8000 Enter your name.	

			>>java User Usage: java User <host> <portNumber> Now using host=localhost, portNumber=8000 Enter your name.
	>>User1		
	Welcome User1 to our chat room. To leave enter LogOut in a new line.	*** A new user User1 entered the chat room!!! ***	*** A new user User1 entered the chat room!!! ***
		>>User2	
	*** A new user User2 entered the chat room!!! ***	Welcome User2 to our chat room. To leave enter LogOut in a new line.	*** A new user User2 entered the chat room!!! ***
			>>User3
	*** A new user User3 entered the chat room!!! ***	*** A new user User3 entered the chat room!!! ***	Welcome User3 to our chat room. To leave enter LogOut in a new line.
	>>Hello Everybody!		
	<User1> Hello Everybody!	<User1> Hello Everybody!	<User1> Hello Everybody!
		>>@User1 What is up? You are not friends with User1	
		>>#friendme @User1	
	<User2>Would you like to be friends? >>@User2 #friends User1 and User2 are now friends!	<User2>Would you like to be friends? User1 and User2 are now friends!	
		>>@User1 What is up?	
	<User2> What is up?	<User2> What is up?	
	>>@User2 nothing much.		
	<User1> nothing much.	<User1> nothing much.	

	User2 and User1 are not friends anymore! >>LogOut ### Bye User1 ###	@User1 #unfriend User2 and User1 are not friends anymore! *** The user User1 is leaving the chat room!!! *** >>LogOut ### Bye User2 ###	*** The user User1 is leaving the chat room!!! *** *** The user User2 is leaving the chat room!!! *** >>LogOut ### Bye User3 ###