CS3251 Computer Networks I

Spring 2019

Programming Assignment 2

Assigned: March 5, 2019

April 11, 2019, 11:59pm

PLEASE READ THIS CAREFULLY BEFORE YOUR PROCEED

1. Overview

The aim of this project is to have you build on the twitter concept from project 1. Like before, you will be developing both a client and server program. Your server will be able to handle multiple client connections at once. Clients will connect to the server and then handle user commands until the user decides to exit. Clients will be able to tweet as before but this time they will include a hashtag. Clients will also be able to subscribe to a hashtag. Instead of making another client 'download' the message your server will send the tweet to all clients who have subscribed to that hashtag. Clients will store these tweets until the user asks for them to be output using the **'timeline'** command.

This project is much more complicated than the first so don't wait until the last minute.
Please read through the whole description before starting. You are allowed to work with at most one other student.

2. Details

First, a (perhaps obvious) note on command syntax for this instruction:

- "$" precedes Linux commands that should be typed or the output you print out to the stdout.

2.1 Twitter Client

Instead of only completing a single action your client will now need to maintain a connection with the server until the user decides they'd like to exit. When first starting your client it should accept 3 command line args: the server ip, port and a username you would like to use. An example can be seen below. Please follow this order of arguments exactly.

```
$ ./ttweetcl   <ServerIP>   <ServerPort>   <Username>
```

After you run the above command to start your client it should use system standard input to get user input and output corresponding results through system standard output. Here **username only consists of alphabet characters(upper case + lower case) and numbers**. Username should be unique for each client. After client start, it should contact with server to check if the username has been taken. If so, the client should exit directly and notify our user. The same username can only be used after the previous client exits.

Your client should be able to handle the following commands.

- **tweet** "<150 char max tweet>" <Hashtag>
  - The tweet should be uploaded to the server and delivered to all the clients who are subscribed to the hashtag.
  - You should only allow up to [1,150] character tweets, you are expected to handle the illegal messages(length illegal).
  - Messages may be input inside a pair of "". ("" shouldn't be considered as the length of message.)
  - The hashtag can be any string without spaces starting with a # symbol. For example the hashtag could be #3251. Specifically, **hashtags may only contain alphanumeric characters (lower case + upper case)**.
  - The hashtag can have multiple units embedded in it. For example, #cs#3251.
  - Each single unit of hashtag has at least 2 characters (including #) and has at most 25 characters. So this case is legal: #1#2#3#4#5#6 and this case is illegal: ##1#2.
  - There may be a max of 8 hashtag units embedded in a single hashtag.
  - Hashtags are case sensitive. For example #cs3251 is different from #CS3251
- **subscribe** <Hashtag>
  - This allows a client to subscribe to a certain hashtag.
  - Here we ensure we will only give a single unit of hashtag as input, you don't need to handle multiple units.
  - **#ALL** subscribes the client to all hashtags so it should receive all new tweets. Since **#ALL** serves a special purpose here, you cannot use it as a hashtag while tweeting. (This command is not allowed: $ tweet "hello world" #ALL and we won't test this)
  - **A client should be able to subscribe to up to 3 hashtags**, when a client reaches its limitation, your client should notify the user this operation fails. **#ALL** only counts as 1 hashtag. Multiple subscriptions to the same # only count as one subscription.

    - **A client may subscribe to a non-existing hashtag.**
    - **Your program should do deduplications on the subscribed messages.** E.g. client A subscribes to #ALL and #1. When another client B runs ‘tweet “message” #1’, A should only receive this message one time. Note that if a client sends the same message twice on the same #, A subscriber to the hashtag should also receive this message twice.
    - If a client has not subscribed to any hashtags it will not receive any tweets from the server, even if it is the message sender.
    - If a client sends a tweet to a hashtag that it is subscribed to it will also receive this tweet from the server.
- **unsubscribe** <Hashtag>
    - This allows a client to unsubscribe from a certain hashtag.
    - **#ALL** only unsubscribes itself.
    - This command will prevent the server from sending any new tweets corresponding to the hashtag to the client.
    - Unsubscribe command should have no effect if it refers to a # that has not been subscribed to previously.
- **timeline**
    - The client will output all tweets that have been sent to it by the server since the last time the user has run the **‘timeline’** command.
    - The client should then clear these tweets from its memory, you don’t need to store messages in file system.
    - The output should contain the following 4 pieces of information:

  <client_username> <sender_username>: <tweet_message> <origin_hashtag>
    - Here origin_hashtag means the original hashtag on this message, not the hashtag the client is subscribed to.

- **exit**
    - Clean up any necessary state (especially any subscriptions on the server) and then close the client gracefully.

2.2 Twitter Server

Your server should take in the port number as its only command line arg like so:

```
$ ttweetsrv   <Port>
```

Your server will need to be responsible for managing up to 5 concurrent client connections. Your server will be receive uploaded tweets from a client and then forward the tweet to the appropriate clients based on the hashtag. Your server does not need to store these tweets, just forward them correctly.

2.3 Example Program Flow

The below example assumes there is a ttweetsrv running at 127.0.0.1 port 8080.

Client 1

```
$ ./ttweetcli   127.0.0.1   8080   Matt             # Connect to server
$ username legal, connection established.          # program output (optional)
$ Command: subscribe #3251                          # Subscribe to #3251
$ operation success                                 # program output (optional)
```

Client 2

```
$ ./ttweetcli   127.0.0.1   8080   Raf              # Connect to server
$ username legal, connection established.          # program output (optional)
$ Command: subscribe #ALL                           # Subscribe to special #All hashtag
$ operation success                                 # program output (optional)
$ Command: tweet “Hello from client 2” #3251        # Tweet to the #3251 hashtag
```

Client 1

```
$ Command: timeline                                 # Print all tweets stored on the client
$ Matt receive message from Raf: Hello from client 2 #3251
$ Command: timeline                                 # No new tweets to show
$ Command: tweet “Hello from client 1” #random      # Tweet to the #random hashtag
```

Client 2

```
$ Command: timeline                                 # Print all tweets stored on the client
$ Raf: Hello from client 2 #3251
$ Matt: Hello from client 1 #random
$ Command: exit                                     # Exit the client. Close gracefully.
```

Client 1

```
$ Command: exit                                     # Exit the client. Close gracefully.
```

3. Other Notes

Students can work in **groups of 2 max** for this project.

You need to implement two separate C/C++ (or Python, Java) programs: one for the server (**ttweetsrv.x**) and another for the client (**ttweetcli.x**). Note that, here **"ttweetsrv.x"** could be **ttweetsrv.c**, **ttweetsrv.cpp**, **ttweetsrv.py**, or **ttweetsrv.java**, it depends on which language you choose. Same rule applies to "**ttweetcli.x**".

- For running command, we will run each language as:
    - Java(both cases are ok)
        - java ttweetsrv <port_number>
        - java -jar ttweetsrv.jar <port_number>
    - C/C++
        - ./ttweetsrv <port_number>
    - Python
        - python2 ttweetsrv.py <port_number>
- Your server program should accept **one** required input parameter: the port number. For example, if we want the server to listen to port number 8090, we will start your program by doing:
    - "./ttweetsrv 8090".
- Your client program should accept **three** required input parameters: the server ip, server port, and the chosen username (we ensure the username are different for different client instances). For example, if the server ip is 127.0.0.1, port number is 8090 and desired username is Matt then we will start your program by doing:
    - “./ttweetcli 127.0.0.1 8090 Matt"

3.1 Deliverables

- You need to submit one compressed folder containing
    - ttweersrv.x
    - ttweetcli.x
    - Makefile (if you use C/C++/Java)
    - README
    - Sample output
- In the README please give a high level description of your implementation ideas.
    - Please include the names of both team members and an explanation of how the work was divided between the team members (i.e. who did what).
    - You should explain clearly how to use your code.
    - You should explain how to install dependent packages or any special instructions for being able to test your code on the shuttle servers.
- Note that, We will compile your program by just doing “make”, if you use C/C++/Java.
- Please comment your source code.

3.2 Notes

1. **Please strictly conform to the input & output formats. We use automated scripts to test your program. If you don’t adhere to the formats, our testing may fail to show that your program works.**
2. Please output necessary information in the output to help us identify the current client’s identity.

3. We don’t have a high requirement for the concurrency of your program but you still need to handle data races under multithreading environment.
4. For error handling, we require 2 states. The first one is **before the user successfully login**, these errors including the wrong IP address, invalid port number, duplicate usernames...etc, **your client program should exit gracefully with error message.** The second state is **after a successful user login**. These errors including invalid message format, subscribing to more than 3 hashtags...etc, **your program should print the error message but keep running for the next command.**
5. For input format, we will only test for the handling of invalid messages. Otherwise, you can assume all the input hashtags and usernames are valid in format
6. Your programs are to be written in C, C++, Python or Java.
7. **We will test your code at the shuttle machines.**

   See https://support.cc.gatech.edu/facilities/general-access-servers

   We strongly suggest that you test your code in those machines before you submit it. ‘

8. **For java, shuttle machines only support java 1.8 or lower. Note that there is not javac compiler in shuttle machine, so you need to upload runnable jar file to test. For python, shuttle machines only support python 2 version.**
9. Use explicit IP addresses (in dotted decimal notation) in the client for specifying which server to contact. Do not use DNS for host name lookups.
10. Make sure that you do sufficient error handling such that a user can't crash your server. For instance, what will you do if a user provides invalid input?
11. You must test your program and convince us it works! Please provide some sample output that demonstrates you have implemented all of the required functionality.

3.3 Implementation Tips

Here is an example of how to support multiple clients connecting to the same server:

Server: geeksforgeeks

The server side code uses select to support multiple clients connecting to the same server. Note that, this is not the only solution or the best solution, but maybe the simplest solution if you are not familiar with multi-threading.

If you have experience or want to learn about multi-thread programming, you can use multiple threads to handle your clients for this project. If going this route you may want to check out the pthreads library and skimming through this guide may be helpful as well. There are tons of other online resources on how to approach this so as always Google is your friend here!

3.5 Grading Guidelines:

We will use the following guidelines in grading:

0% - Code is not submitted or code submitted shows no attempt to program the functions required for this assignment.

25% - Code is well-documented and shows genuine attempt to program required functions but does not compile properly. The protocol documentation is adequate.

50% - Code is well-documented and shows genuine attempt to program required functions. The protocol documentation is complete. The code does compile properly but fails to run properly (crashes, or does not handle properly formatted input or does not give the correct output).

75% - Code is well-documented. The protocol documentation is complete. The code compiles and runs correctly with properly formatted input. But the program fails to behave gracefully when there are user-input errors.

100% - Code is well-documented. The protocol documentation is complete. The code compiles and runs correctly with properly formatted input. The program is totally resilient to input or network errors.