CS3251 Computer Networks I

Spring 2020

Programming Assignment 2

Assigned: Feb 24, 2020

April 07, 2020, 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. Moreover, clients can request a list of all online clients' usernames from the server. Then, clients can get all the history tweets of a specific client by his/her username.

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 must work in groups of 2 or groups of 3. No individual work.**

Please register your group on canvas on the group set "PA2", this is necessary for us to track who is in your group.

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. Your program should not print this and it is purely for instructional purposes of the document.

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:

1. Server IP: This can be **ANY** valid IP address (not just loopback addresses)
2. Server Port: This can be **ANY** valid port number
3. Username: This is a string consisting of only **alphabet characters(upper case + lower case) and numbers**

An example can be seen below. Please follow this order of arguments exactly.

$ ./ttweetcli <ServerIP> <ServerPort> <Username>

Be sure to handle all possible exceptions, never trust the user to provide correctly formed input.

Use the following error messages **EXACTLY**, marks will be deducted if feedback messages do not conform:

1. Server IP invalid: "error: server ip invalid, connection refused."
2. Server Port invalid: "error: server port invalid, connection refused."
3. Username is invalid: "error: username has wrong format, connection refused."
4. User is already logged in: "username illegal, connection refused."
5. Wrong number of parameters: "error: args should contain <ServerIP> <ServerPort> <Username>"
6. Illegal message length(>150): "message length illegal, connection refused."
7. Illegal message length(=0 or None): "message format illegal."
8. Illegal hashtag: "hashtag illegal format, connection refused."
9. Maximum hashtags reached: "operation failed: sub <hashtag> failed, already exists or exceeds 3 limitation"

The success messages are as follows:

1. If the user manages to log in: "username legal, connection established."
2. User exits successfully (no full stop): "bye bye"

3. For any subscribe/unsubscribe operations that succeeds (no full stop): “operation success”
4. For tweet operation, no feedback needed

After you run the above command to start your client it should use system standard input to get user input and output corresponding result through system standard output. **Username should be unique for each client. After a client starts, it should contact the server to check if the username has been taken.** If so, the client should exit directly and notify our user. The same username could only be used after the previous client logout.

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 immediately who are subscribed to the hashtag**.
        - Here “immediately” means server will push message to other clients immediately
    - You should only allow up to [1,150] character tweets, you are expected to handle the illegal messages(length illegal).
    - We ensure the message will be inside a pair of “” which hashtag isn’t.(“” shouldn’t be considered as the length of message.) **Message itself doesn’t have “”, but could contain other symbols**.
    - The hashtag can be any string without spaces starting with a # symbol.
        - For example the hashtag could be #3251.
        - Specifically, **hashtag only has alphabet characters(lower case + upper case) and numbers**. Only # is illegal.
    - The hashtag can have multiple units embedded in it.
        - For example, #cs#3251, each single unit of hashtag has at least 2 characters(including #). So this case is legal: #1#2#3#4#5#6 and this case is illegal: ##1#2
    - Hashtags are case sensitive.
        - For example #cs3251 is different from #CS3251
    - There is a limitation of 5 on the number of hashtag units for each message. Also, for the length of unit, the limitation is 15(# counts as 1 character). **Our test driver will follow this limitation to test your program and you don’t need to do extra check on these two.**
- **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 test driver won’t test a tweet with hashtag #ALL
    - **A client should be able to subscribe to up to 3 hashtags**, when a client reaches its limitation, the client should notify the user this operation fails. **#ALL** only counts as 1 hashtag. Multiple subscriptions to the same # only count as one subscription.
        - Your program is expected to output this error message if fails:
        - operation failed: sub <hashtag> failed, already exists or exceeds 3 limitation
    - **A client may subscribe to a non-existing hashtag.**
    - **Your program should do deduplications on the subscribed messages.**
        - E.g. a client A subscribes to #ALL and #1. When another client B run ‘tweet “message” #1’, A should only output this message once. Note that if a client sends the same message twice on the same #, then the subscriber to the hashtag should also print 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** will unsubscribe **THIS CLIENT** from all hashtags (including the other 2 hashtags that could be added on top of the #ALL)
    - 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; each line has only 1 tweet.
    - The output should be in this **EXACT** format (note the spaces and semicolon):
        - <sender_username>: “<tweet_message>” <origin_hashtag>
        - Here origin_hashtag means the original hashtag on this message, not the hashtag client subscribed.
        - The message body is wrapped by quotes
        - There is an ‘:’ after sender username
    - If the client subscribes to a new hashtag, it’s previous timeline messages won’t be updated.
        - If A & B are online, A doesn’t subscribe to any tags and B sent a message, then A subscribes to #ALL, if you input timeline for A, it will show no message because A doesn’t receive this message from server when the message was sent, though A subscribes to it later.
- **getusers**

    - The client would output all online users' usernames (including itself); each line has only 1 username.
    - The client should then clear this list from its memory.
- **gettweets** <Username>
    - The client will output all historical tweets sent by the user who has the <Username>; each line has only 1 tweet.
    - The output should still follow the format in the description of the **timeline** function.
    - The client doesn't need to store the messages and these messages shouldn't be counted as timeline messages.
    - If there is no online client named <Username>, this client should output error message:
        - "no user <Username> in the system"
- **exit**
    - Clean up any necessary state (especially any subscriptions on the server) and then close the client gracefully.
    - If the client logs out, he will be unsubscribed from all hashtags and his account is essentially deleted from the server

2.2 Twitter Server

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

$ ttweetser <Port>

Your server will need to be responsible for managing up to 5 concurrent client connections. Your server will keep receiving uploaded tweets from a client and then forward the tweet to the appropriate clients immediately based on the hashtag. Your server should store all tweets from all online clients. When a client logs out, the server will remove all history tweets and other states of this client from its memory.

2.3 Examples & Test Drivers

Since this project is a little bit complex. We provide you sample test case outputs. They are divided into multiple pieces for better reading.

https://drive.google.com/drive/folders/1ZtjS9nE9Q0SLrj0K25o_J5zGV9211Ykf?usp=sharing

We have provided 2 scripts in the folder:

1. Judge2.all.py - this will put all output of all clients into a single file (client.txt). This would allow you to see the order of operations but due to concurrency and non-determinism the order of your output may not be the same as the sample output we have given (**that is fine**).
2. Judge2.seperate.py - this will put outputs of each client in individual files allowing you to see the commands and responses of each individual client. The output **should be in the same exact order** as the sample outputs we have provided. We will only grade your assignment based on this script and not Judge.all.py. The output file are in details folder in the google drive
3. **While grading, we will only use judge2.seperate.py to test your program and do string matching for the file output(only the client output, we don't care your server output, which is in server.txt).** Your program are expected to have the same output exactly the same as examples except the initial command launching your program(this will be different due to different programming languages)
4. **You may think we won't test other edge cases not shown in the script**, but we may try to upload more messages or different content to test your program's functionality.

3. Other Notes

Students **must** work in **groups of 2-3 (no groups of 4 or individual projects)**. If you are unable to find a teammate, use piazza to look for a teammate.

You need to implement two separate C/C++ (or **Python3**, Java) programs: one for the server (**ttweetser.x**) and another for the client (**ttweetcli.x**). Note that, here "**ttweetser.x**" could be **ttweetser.c**, **ttweetser.cpp**, **ttweetser.py**, or **ttweetser.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 ttweetser <port_number>
        - java -jar ttweetser.jar <port_number>
    - C/C++
        - ./ttweetser <port_number>
    - Python3
        - python3 ttweetser.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:
    - "./ttweetser 8090".
- Your client program should accept **three** required input parameters: the server ip, server port, and the chosen username.
    - 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
    - ttweerser.x
    - ttweetcli.x
    - Makefile (if you use C/C++/Java)
    - README
- 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
- Note that, **We will compile your program by just doing “make”, if you use C/C++/Java.**

- Please add comments to your source code

3.2 Notes

1. **Please strictly conform to the input & output formats in your client side. 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. **For students who wish to use python, only python3 will be allowed for this project so do ensure your python code conforms to Python3 syntax.**
3. Please output necessary information in the output to help us identify the current client's identity.
4. We don't have a high requirement for the concurrency of your program but you still need to handle data races under a multithreading environment.
5. 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 include invalid message format, subscribing to more than 3 hashtags...etc, **your program should print the error message but keep running for the next command.**
6. For input format, we will only test for the handling of invalid messages.
7. Your programs are to be written in C, C++, Python3 or Java.
8. **We will NOT be using the Shuttle machines for this project. If you are using C/C++/Java please provide a make file with a "make all" and "make clean" command. Submissions without a makefile or submissions that are unable to compile will lead to -20 deduction.**
9. Use explicit IP addresses (in dotted decimal notation) in the client for specifying which server to contact. Do not use DNS for hostname 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 your program output for the following test scenario. In grading your submission we will use the same test sequence (except we will use our own messages).

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!

3.4 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.

Similar to Programming Assignment 1, requirements that are specifically in this document (such as format of error messages and format of success messages) will be penalised more than requirements not explicitly stated.

Failure to conform to the requested output messages will be considered a failure of the test case and lead to mark deductions without partial credit, so please do check that you follow the output requested **EXACTLY**.