CS 352 Web Chat Server Assignment

Announcement: Due to time constraints, only the first test case needs to be developed. The remainder can be handed in for extra credit. See the grading section below.

Note: a modified client and a source code server are posted in Canvas. See the video for more information on how to use them.

In this assignment you will write a tiny chat/messaging server that works over the HTTP protocol in Java. You may program this project in groups of up to two people, or program it alone. You must have a separate README file with the netids and names of your group members.

Your server must run on the Computer Science ilab machines **without using any additional libraries.**

You will be given a Java binary in the form of class files of an example server and clients in Python3 source code. You must hand in your Java source code that implements the same functionality as the example Java server.

Running the examples:

Unpack the zip or tar file from the Canvas site. In the top level directory, run the server:

Step1 : `javac HTTPChatServer.java`

Step2 : `java HTTPChatServer <port>`

There are 2 client test programs, `client_test_1.py` and `client_test_2.py`.
Run a test client:

Format: `python3 client_test_1.py <address> <port> <username> <password> <message>`

Example = `python3 client_test_1.py 0.0.0.0 5050 user2 pwdB "hello”`

The Chat Server Protocol

You will build a basic chat server in Java that allows users to login with a username and password, and post messages to a chat log. The server uses HTTP as the base protocol. Recall from the beginning of class that HTTP uses strings as the base data-type and message have a structure with a header consisting of variable-value pairs, followed by a body with data.

Protocol Overview:

1: The client sends a request for the login page using either (1) TCP socket, or (2) a URL connection.

The Client program sends a URL requesting the login page. The path to the URL is under /login. For example, http://server-name:/login/

See the `getLoginPage()` method in `client_test_1.py`.

2: The client sends an HTTP POST message with the username and password with a URL path of /login. The server checks if the username and password are in a csv file called "`credentials.txt`" in the same directory as the server. If the server finds the username and password, the server creates a new cookie that can authenticate this user, then returns the chat page including a set-cookie in the HTTP header. Otherwise the server returns an error page. The client extracts the cookie from the HTTP response.

See the `postChatPage()` method in `client_test_1.py`

3: The client sends a post with a chat message with the path of /chat/, i.e. a URL might look like http://server-name:/chat/ . The client must include the cookie from step 2 in the HTTP header. The server must validate that this is a cookie it generated before when it authenticated a client. That is, posts to the chat log are not allowed without a valid cookie. See the `postChatPage` method in the client for an example.

4. The client sends an HTTP GET message to the server to get a list of all the posted messages. See the

Background on Cookies

In the HTTP protocol, cookies are a simple mechanism that allows the server and client to remember state between them. The basics of cookies are covered in this tutorial. When the server returns a payload/web page back to the client, it can include a sequence of set-cookie lines in the HTTP response header. For example, a set of lines in the HTTP header returned by the server might look like this:

```
HTTP/2.0 200 OK
Content-Type: text/html
Set-Cookie: userID:=7625145
```

Cookies are usually variables/value pairs. A cookie can also be just a variable with an implicit value. E.g., you can just use a "Set-Cookie" with no value, in which case the implicit value is true.

The client will return the names and values of the cookies back to the server on every web-request. Cookies can thus be used to build session state. See the links at the bottom of the page for a tutorial and definition of cookies.

Additional Resources:

[Java TCP Sockets tutorial]
[Example Java TCP server]
[RFC-1945]
[HTTP POST]
[HTTP POST vs GET]
[HTTP POST examples]
[Mozilla Cookie tutorial]
[Official RFC definition of Cookies to the HTTP protocol]

How to get started:

Write a very simple server that creates a server socket or servlet object. See the example Java TCP server above.

Connect the python3 client to your server
When the client connects, read the input HTTP message from the socket
Parse the message into lines and words.
Get the HTTP commands from the parsed message
Call a method to handle the HTTP command
Build a string of the HTTP response to the command
Write the response back to the client on the socket

Example server and client files:

Documentation files:
README
help-server.txt
help-clients.txt

Example server implementation:
HTTPChatServer.class
client_handler.class

Test clients:
client_test_1.py
client_test_2.py
utils.py

Data files and directory for the server:
credentials.txt
login/
login/login.html
login/credentials.txt
login/error.html
chat/
chat/chat.html

What to Hand in:

- You must hand in a single file upload that is either (1) a zip file or (2) compressed tar (tgz) with all the files and directories needed to run your server from the top-level directory.
-
- You must have a file called README that contains the netid and names of the group members. Each line in the file must be for the form: `netid:name`

- The server must compile in the top-level directory with the command: `javac *.java`

- The server must run with the command: `java HTTPChatServer <port-number>`

Grading:

Grading will be mostly auto-graded based on the following tests. Some points may also be awarded at the TA's discretion.

1. Server compiles without errors: 30%
2. Passes linear test of a user login and single message post for a single client: 40%
3. Overall programming style. How readable is the code? Does it have a logical structure and comments? (subjective): 30%

Extra Credit (8% of the total class grade):

1. Passes multiple linear tests for using multiple clients: 33%
2. Correct use of cookies to forbid access to the message log without a login: 33%
3. Overall programming style. How readable is the code given it passes the above 2 tests? Does it have an easy to follow logical structure and comments? (subjective): 33%