CS 447 : Networks and Data Communications

Programming Assignment #02

Total Points: 150

Assigned Date : Thursday, March 20, 2019
Due Date : Thursday, April 03, 2019 @ 11:59:59 a.m.

Overview

Your second programming assignment is to **implement an authenticated email application** using the socket interface. There are several objectives of this assignment. These are:

a. to build up on socket programming knowledge from first assignment;
b. to further reinforce the concept of "*protocol*" using hands-on programming;
c. to learn how to read and understand RFCs;
d. to gain experience in developing both TCP and UDP -based applications; and
e. to get a basic understanding of user authentication mechanisms.

Back Story

The success of *Professor Calculus's* online calculator is spreading like wildfire and his friends are reaching out to him for help with *"network communication problems"*. Captain Haddock, for one, is having all sorts of issues with the carrier pigeon system he uses when he is on sea voyages. Often, he is finding out that the messages he sends do not reach their intended recipients *in their original form* because the pigeons like to peck on the special corn-based paper Haddock uses. On the receiving side, Haddock has observed that, most times, the messages he receives are *late*. He also does not enjoy writing messages on behalf of his crewmates; right now, none of his crewmates can afford to own their own carrier pigeons. Professor Calculus has a brilliant solution. "Let me introduce you to email, Haddock", said Calculus. He promises Haddock that this new email solution will provide reliable delivery when sending emails, be capable of supporting more than one sender at the same time, and to have super speedy (although sometimes unreliable) delivery when receiving. As an added security measure, Calculus plans to provide basic user authentication functionality through his email system.

Technical Requirements

1. Just as in PR01, you will need to write a client-server application to support Calculus's email system.

2. Your `sender` → `server` interaction should follow the SMTP protocol, and must support the following SMTP commands: **`HELO`**, **`AUTH`**, **`MAIL FROM`**, **`RCPT TO`**, **`DATA`**, and **`QUIT`**. Read Section 4.1 of the SMTP RFC found at https://tools.ietf.org/html/rfc2821 for exact command specifications. The **`AUTH`** command is described in Section 4 of https://tools.ietf.org/html/rfc4954. Read item 10 and 11 below for additional specifications.

   - Closely related to the SMTP commands are the corresponding reply codes. Read Section 4.2.2 of the SMTP RFC, select appropriate reply codes, and implement them. I anticipate you to find at least 5-6 reply codes necessary for your implementation. Explain your reply code selection and the justification in your report.

3. Email addresses should have the typical email format, i.e., should include the @ sign. For the purpose of simplicity in parsing, assume all send and receive email addresses are from the `cs447.edu` domain. E.g. `calculus@cs447.edu`
4. Emails are written at the `server`, not at the `sender`. In other words, your SMTP interaction should not be a file transfer.
5. The `sender` → `server` interaction should run over TCP.
6. The `sender` → `server` interaction should support **multi-threading**; more than one client should be capable of sending emails at the same time.
7. Your `server` → `receiver` interaction should follow the HTTP/1.1 protocol and implement it's **`GET`** method. Read Section 4.3.1 of the HTTP/1.1 RFC found at https://tools.ietf.org/html/rfc7231 for exact specifications.
   - The corresponding HTTP/1.1 reply codes (referred to as status codes in the RFC) are found in Section 6.1 of the RFC 7231. Bare minimum, you must implement reply codes **`200`**, **`400`**, and **`404`**. Explain how you used these reply codes (and any additional reply codes you decided to use) in your report.
8. The `server` → `receiver` interaction should run over UDP. Multi-threading is not required for this interaction.
9. To simply the implementation, we'll use a slightly modified **`GET`** request and response format for this project. Please find more information under the Logistics section below.
10. Sender Authentication:
    - The **`AUTH`** command is issued between the **`HELO`** and **`MAIL`** commands in the SMTP sequence.
    - Following the **`AUTH`** command, the server replies back with code **`334 dXNlcm5hbWU6`**[†] prompting the user to enter his/her username. (e.g. `username@cs447.edu`).
    - Once the user sends his username, the server then responds back with code **`334 cGFzc3dvcmQ6`**[‡] prompting the user to enter his/her password.
    - A successful (or failed) authentication is marked by the appropriate reply code. Read Section 6 of RFC #4954 for status codes and implement the appropriate ones.
    - Only when the authentication is successful the user is able to proceed to the next command in the SMTP sequence, i.e., **`MAIL FROM`** command.
    - See Logistics section below for additional information on password generation.
11. Receiver Authentication: This is similar in function to the sender authentication. Due to time constraints, we'll use a simplified HTTP authentication mechanism instead of following HTTPS standards.

[†] 'username' in base64
[‡] 'password' in base64

- Upon `receiver` → `server` connection, use the same procedure as item 10 above for receiver authentication. Make your server prompt for username and password, compare the values against the stored value in the password file. You will have to go through a registration phase just like in receiver authentication. Use the same reply codes.
- Only successfully authenticated users are allowed to download unread emails.

12. Server Incident Management: The server(s) keep an active log file (named `.server_log`) of the connection activities. Every time the server sends or receives a message, a log entry is added to this log file. More specifically, each log entry (single line) has the following format:

Answer

`timestamp from-ip to-ip protocol-command message-code description`

Logistics

1. IP addresses/hostnames and port numbers should not be hard coded.
   - Depending on how you decide to implement your server(s), your server executable will accept one (or two) command line argument to denote the corresponding listening port(s) as follows: `./server <tcp-listen-port> (<udp-listen-port>)`
   - Your client executable will accept two command line arguments as follows: `./client <server-hostname> <server-port>`.
     Your may assume that your client knows the server hostname/ip address and the correct port to connect to.
2. Your client and server should be able to run on two separate end systems. Bare minimum, you should verify an interaction between a client running on a lab machine (EB 1036 dual boots to Linux) and the "`cs home`" server and vice-versa. Depending on the firewall rules, you might also be able to test from off-campus using your own laptop/desktop as one end system as well.
3. I will test **at least** 2 simultaneous client connections, once as two senders, and once as one sender + one receiver. Make sure this is covered in your testing plans.
4. All clients should exit gracefully. Server process is permitted to be forcefully killed.
5. Use the following file management strategy:
   - When your SMTP server fires up for the first time, make it programmatically (not manually) create a folder named **`db`** to store emails.
   - Programmatically create a new subfolder inside **`db`** for each new recipient that's mentioned in the **`RCPT TO`** command above.
   - Store emails as sequentially numbered files. E.g. first email to Haddock will be stored as `/db/haddock/001.email`
   - When a new receiver fires up for the first time, programmatically (not manually) create a folder under receiver's name to store retrieved emails.
6. To keep the project within the time spec, the following simplified password strategy is proposed.
   (a) The first time the user responds to **`334 dXNlcm5hbWU6`**, the server replies back with a 5-digit randomly generated password over reply code **`330`** (instead of code **`334 cGFzc3dvcmQ6`**). The server adds 447 to this number, encodes it in base64, and stores the encoded password along with the corresponding username in a hidden master password file named "`.user_pass`" inside the **`db`** folder.

(b) Upon receiving the 330 code and the temporary password, the client immediately terminates the current connection, waits for 5 seconds, and re-initialize a fresh connection.
(c) On all successive connections, the client responds to server's request for password, i.e. **`334 cGFzc3dvcmQ6`**, by typing in the previously received password.
(d) The server adds 447 to the received password, encodes it in base64, and checks the encoded value against the stored value in the password file. If these two values match, the authentication phase is successful.
(e) Your client programs will take the plain text `username` and `password` entered by the user and encode it in base64 before sending. In otherwords, the server receives base64 encoded values, **NOT** plaintext `username` and `password`.

7. Make sure that only authorized users are allowed to send emails. What this means is basically is to check at the client end if the username that was entered during the login process matches with the **`mail from`** field.
8. Your sending client is not supposed to autofill the `@cs447.edu` suffix of the email address for the user.
9. All SMTP command validations, including the infamous end of message dot (.), and error code generation should be done at the server side and not at the client side. For all practical purposes, the sending client is just a message relay.
10. Your client programs – both the sender and the receiver – should prompt the user to enter appropriate information, rather than making them type in the correct protocol commands; your programs should handle the correct protocol interaction internally. In other words, assume the user of your program is only interested in writing and retrieving emails, and have no knowledge of protocols and how they work. You, as the developer on the other hand, is well-versed in protocols, which would be reflected on your code. *If this is not clear, make sure to talk to the instructor immediately.*

    - The receiver is not expected to type the HTTP GET message syntax. Your program will do that for the user based on his preferences. Think about it. Have you ever typed in a the HTTP GET syntax on your browser before visiting a web page?

11. ~~For the receiver, when the user successfully logs in, he/she should be greeted with the unread number of messages in the server. The user should be given the option to either selectively read a message or read all messages.~~

    - Emails read by the receiver should be downloaded to his (mock) local machine. You are allowed either keep the original copy in the server (IMAP standard) or to delete it (POP3 standard) after it's downloaded. ~~However, on a successive login the number of unread message count should be updated accordingly~~.

12. You are programming two different servers (SMTP over TCP and HTTP over UDP), which could technically be running on two different end systems. Think about this when you design your server code.
13. Here's a sample `.email` file. Note the timestamp added by the `server`.

Answer

```
Date:  Tue, 19 Mar 2019 14:05:11 -0500
From:  <tintin@cs447.edu>
To:  <haddock@cs447.edu>
Subject:  The Last Unicorn
```

```
Dear Haddock,

Glad to hear that you found the last Unicorn.  We are looking
forward to your safe return.

Yours truly,
Tintin and Snowy.
```

14. Here's a sample `GET` request.

    Answer

    GET /db/haddock/ HTTP/1.1
    Host: <server-host-name>
    Count: 1

    Note: **`Count`** denotes the number of emails to download. Additionally, you may find reading the HTTP/1.1 Message Syntax and Routing RFC found at https://tools.ietf.org/html/rfc7230 helpful for understanding.

15. Here's a corresponding successful `GET` response from the server. Store the response as a `.txt` file under the receiver's folder.

    Answer

    HTTP/1.1 200 OK
    Server: <server-hostname>
    Last-Modified: Tue, 19 Mar 2019 14:06:08 -0500
    Count: 1
    Content-Type: text/plain
    Message: 1

    ```
    Date:  Tue, 19 Mar 2019 14:05:11 -0500
    From:  <tintin@cs447.edu>
    To:  <haddock@cs447.edu>
    Subject:  The Last Unicorn

    Dear Haddock,

    Glad to hear that you found the last Unicorn.  We are looking
    forward to your safe return.

    Yours truly,
    Tintin and Snowy.
    ```

16. At the end of your implementation, you should be able to:
    - Compile and run your code on a typical Linux machine. Include a readme.txt file with clear compilation instructions.
    - Run your server program(s) first.
    - Run one or more clients to connect to the server to send emails.
    - Run a receiver to retrieve email.

- Exit the client(s) gracefully.

Instructions

- **Start early!!**
- **Take backups of your code often!!**.
- Follow a good coding standard. Use the Google C++ coding standard found here http://goo.gl/1rC1o, if you don't already follow one.
- The due date of this assignment is **Thursday, April 03, 2019 @ 11:59:59 a.m.**. A dropbox will be opened for submission on Moodle.

Deliverables

A complete solution comprises of:

- A short report (max 5 pages) of the design and implementation of your system. Your report should include the followings:
  - Introduction
  - Design choices and protocol/reply codes used.
  - The output of a sample run (including screenshots where applicable).
  - Summary and Issues encountered (if applicable).
- A short `readme` file with compilation instructions. Also preferable is a `makefile` to compile your code.
- A compressed tarball of the directory containing your source code. **Do not** include executables, folders created by your programs, or your test emails in this tarball. To create a compressed tarball of the directory source, use the following command: `tar -zcvf siue-id-pr2.tar.gz source/`. e.g. `tar -zcvf tgamage-pr1.tar.gz PR02/`

Collaborating on ideas or answering questions is always encouraged. Most times, I find that you learn a lot from your peers. However, do not share/copy/duplicate code from others. If you use code found online, remember to site their source in your report. Issues related to academic integrity and plagiarism have **ZERO** tolerance.

Useful Resources

- Linux Man pages – found in all linux distributions
- Beej's Guide to Network Programming – A pretty thorough free online tutorial on basic network programming http://beej.us/guide/bgnet/output/print/bgnet_USLetter.pdf
- Simple Mail Transfer Protocol RFC #2821 https://tools.ietf.org/html/rfc2821
- Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content RFC #7231 https://tools.ietf.org/html/rfc7231
- Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing RFC #7230 https://tools.ietf.org/html/rfc7230
- SMTP Service Extension for Authentication RFC #4954 https://tools.ietf.org/html/rfc4954
- Base64 encoding/decoding:
  - C/C++

    - glib – https://developer.gnome.org/glib/stable/glib-Base64-Encoding.html
    - openssl http://fm4dd.com/openssl/manual-crypto/BIO_f_base64.htm
    - GNU coreutils http://www.gnu.org/software/coreutils/coreutils.html
  - Java 8 – https://docs.oracle.com/javase/8/docs/api/java/util/Base64.html
  - Python 2 – https://docs.python.org/2/library/base64.html
  - Python 3 – https://docs.python.org/3.5/library/base64.html
  - Additional Resource – http://rosettacode.org/wiki/Base64_encode_data#C

If you use any other resources, make sure to cite those in your report. Using online resources does not mean you are allowed to copy and use someone else's code for your purpose. Such incidents, if detected, will be treated as academic dishonesty.