# CS447 : Networks and Data Communications
## Programming Assignment #02
### Total Points: 150

**Assigned Date** : Wednesday, October 19, 2022
**Due Date** : Wednesday, November 02, 2022 @ 01:29:59: p.m.

## Overview

Your second programming assignment this semester is to **implement an elementary email application** using the socket API. This assignment is a multi-objective assignment and is fairly loaded. Moreover, your PR02 is expected to be an extension of PR01. Thus, start early.
The assignment learning objectives are:

a. to further reinforce the concept of "*protocol*" using hands-on programming;
b. to refresh multi-threading, system calls, and other systems programming concepts;
c. to learn how to implement protocols to their RFC specifications; and
d. to gain experience in developing application layer network programs utilizing both TCP and UDP.

Here the back story.

## Back Story

Captain Haddock typically uses a carrier pigeon system when he is on long sea voyages. In recent times, however, it that the pigeons have developed a liking to the corn field on their flight path and often takes "snack breaks" while on the job. Often, the messages sent arrive way too late the Haddock's liking. Haddock also does not enjoy having to borrow his carrier pigeons to his crew-mates; 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 lighting fast when sending and receiving emails, and also will support more than one sender and/or receiver at the same time.

## Technical Requirements

1. 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 at minimum must support the following SMTP commands: **`HELO`**, **`MAIL FROM`**, **`RCPT TO`**, **`DATA`**, **`HELP`**, and **`QUIT`**. Read Section 4.1 of the SMTP RFC found at `https://tools.ietf.org/html/rfc5321` for exact command specifications.
    - Closely related to the SMTP commands are the corresponding reply codes. Read Section 4.2 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 `447f22.edu` domain. E.g. `calculus@447f22.edu`. **Note**: you are not to auto-fill the domain but allow the user to enter it. Your program, however, should ensure correct domain.
4. Emails are written at the `server`, not at the `sender`. In other words, your SMTP interaction should not be a file transfer.
5. For all practical purposes, your client program is a strict message relay; Your sending client will take whatever the input the user types and send it to the server. This includes the user having to typing-in the appropriate SMTP commands as well. All protocol related validations should occur at the server. Make sure you strictly adhere to the specifications listed in the SMTP RFC. Any deviations from the RFC must be justified and specifically noted in your report.
6. The `sender` → `server` interaction should run over UDP.
7. The `sender` → `server` interaction should support **multi-threading**; more than one client should be capable of sending emails at the same time.
8. 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.
9. The `server` → `receiver` interaction should also run over UDP. Multi-threading is not required for this interaction at this stage.
10. To simplify 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.
11. Consider augmenting your **`HELP`** command to provide sufficient formatting information on each SMTP command. For example, **`HELP RCPT`** should provide the user with the correct syntax expected by your program.
12. Your receiving client will first prompt the user to enter his username and then the number of emails to download. Your program will use these arguments to craft the appropriate GET request and print it to the standard output for a confirmation from the user **before** being sent to the server.

## Logistics

1. IP addresses/hostnames and port numbers should not be hard coded.
    - You are fundamentally expected to implement two different servers, one to handle SMTP logic and another for HTTP logic, but combine the execution of both servers using a single driver program. This driver will accept the necessary listening ports from a configuration file[†] (e.g. `server.conf`) present in the same working directory. For example:

      ```
      ./server server.conf
      ```

      **`server.conf`**

      ```
      SMTP_PORT=
      HTTP_PORT=
      ```

      This is considered the default execution behavior for the servers. Any deviations should be properly justified and documented in your report. Your `README` should also list appropriate execution instructions if different from above.

[†]**Note**: A configuration file is simply a text file with a `.conf` extension.

- Your client executable also follows a similar pattern and accepts the server socket address from a configuration file (e.g. `client.conf`)

  `./client client.conf`

  **client.conf**
  ```
  SERVER_IP=
  SERVER_PORT=
  ```

2. Client(s) and server should be able to run on two separate end systems but still communicate with each other. Use the zone server `zone.cs.siue.edu` to spawn multiple containers to test simulated over-the-network communication behavior. At minimum, include the following two test cases in your testing plans.
   - Two concurrent senders able to send emails without interruption/interference from each other.
   - One sender and one receiver concurrently able to interact with the server(s) without interruption/interference from each other
3. All clients should exit gracefully. Server process is permitted to be forcefully killed.
4. Use the following email 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`. Note the file format "`.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.
5. Here's a sample `.email` file. Note the timestamp added by the `server`.

   ```
   Date:  Mon, Oct 03 2022 15:09:42 -0600
   From:  <tintin@447f22.edu>
   To:  <haddock@447f22.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.
   ```

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

   ```
   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 the syntax.
7. Here's a corresponding successful `GET` response from the server. Store the response as a `.txt` file under the receiver's folder. **Note**: You must assume that the receiver client is executed on a separate end system thus create folders to download emails appropriately.

```
HTTP/1.1 200 OK
Server: <server-hostname>
Last-Modified: Mon, Oct 03 2022 15:17:42 -0600 -0600
Count: 1
Content-Type: text/plain
Message: 1

Date:  Mon, Oct 03 2022 15:09:42 -0600
From:  <tintin@447f22.edu>
To:  <haddock@447f22.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.
```

8. Print all reply codes, on all interactions, to the standard output (only).
9. **Minimum Testing Plan**: At the end of your implementation, you should be able to:
   - Compile and run your code on a typical Linux machine.
   - Run your server program(s) first.
   - Run several clients concurrently to connect to the server and be able make independent progress without interference from each other to send emails.
   - Run a receiver (either concurrently or sequentially with the sender) to retrieve email.
   - Exit the client(s) gracefully.

## Instructions

- **Start early!!**. This is a fairly loaded assignment.
- **Take backups of your code often!!**.
- Follow a good coding standard. Use one of Google's coding standard found here `https://google.github.io/styleguide/`, if you don't already follow one.
- Your code must compile and run on a typical Linux setup. Neither the instructor nor his graders will use (or entertain the use of) any IDE to test your implementation. Be sure to test command line compilation and run before submission.
- Implementation language must be C/C++.
- **Absolutely do not** include executables, folders created by your programs, hidden files, version control repositories, or any irrelevant files in this tarball. All project relevant file formatting standards (**`PDF, README, .txt, .tar.gz`**) will be strictly monitored and are subject to penalties.
- The due date of this assignment is **Wednesday, November 02, 2022 @ 01:29:59: p.m.**. A Moodle dropbox will be opened for submission.
- Based on past student experience, multi-threading is not as obvious as it may look. I suggest you to first get a single-threaded version working correctly and then think about extending it to multi-threading.
- This assignment can be fully developed using the socket API of your programming language and basic I/O API. Use of other packages/libraries without the instructors permission is not permitted.

## Deliverables

A complete solution comprises of:

- A short report of the design and implementation of your system. The report should be **PDF** format. At a minimum, your report should include the following sections:
  - Introduction: Your objective and what you hope to gain from the assignment.
  - Overall design, specific design choices, and reply codes used.
  - The output of a sample run. Include plenty screenshots wherever applicable. In situations were we can't verify expected behavior, your screenshots maybe considered for partial credit.
  - Summary and Issues encountered. What you were able to achieve from your own objectives (from the introduction) as well as project specifications. Make sure to explicitly list functionality you failed to implement (or buggy).
- A compressed tarball that contains:
  - a directory containing (only) your source code and config files. **Do not** include executables, folders created by your programs, or any other files not specifically listed here as required.
  - A short `README` file with compilation and run instructions.
  - A `makefile` (**mandatory**) to compile your code especially if it involves compiling multiple executables with flag options.

> 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-pr2.tar.gz PR01/`

Collaborating on ideas or answering each other's 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, including online sources. If you find a useful article or a source code online, firstly cite it in your report, and secondly, get the idea from it and do your own implementation. Absolutely do not directly copy-paste code from online sources.
The exercise is meant for you to learn network programming, not to test your googling abilities. The instructor actively uses MOSS http://theory.stanford.edu/~aiken/moss/ to check for software similarity. Issues related to academic integrity and plagiarism have **ZERO** tolerance and will result in a failing grade for the course.

## Some 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 for C/C++ https://beej.us/guide/bgnet/
- Linux Socket Programming In C++ – https://tldp.org/LDP/LG/issue74/tougher.html
- The Linux HOWTO Page on Socket Programming – https://www.linuxhowtos.org/C_C++/socket.htm
- Simple Mail Transfer Protocol RFC #2821 https://tools.ietf.org/html/rfc5321
- 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