# CS 447 : Networks and Data Communications

**Programming Assignment #01**

**Total Points: 150**

**Assigned Date** : Thursday, May 17, 2018
**Due Date** : Tuesday, May 29, 2018 @ 10:59:59 a.m.

## Overview

Your first programming assignment this Summer is to **implement an elementary email application** using the socket interface. This assignment is a multi-objective assignment and, for the limited project time window, is fairly loaded. The assignment objectives are:

a. to start developing socket programming skills and knowledge;
b. to reinforce the concept of "*protocol*" using hands-on programming;
c. to refresh multi-threading, system calls, and other previous programming concepts;
d. to learn how to read, understand, and implement RFCs; and
e. to gain experience in developing both TCP and UDP -based applications.

## Back Story

Captain Haddock is having all sorts of issues with the carrier pigeon system he uses when he is on sea voyages. Often, he finds out that the messages he sends reach their intended recipients *very late*. On the receiving side, Haddock has observed that, most times, the messages he receives are *incomplete or illegible* because the pigeons like to peck on the special corn-based paper that's being used. He also does not enjoy writing messages on behalf of his crew-mates; right now, none of his crew-mates 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 receiving emails, be capable of supporting more than one sender at the same time, and to have super speedy (although sometimes unreliable) delivery when sending.

## 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/rfc2821` for exact command 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 `447.edu` domain. E.g. `calculus@447.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 run over TCP. Multi-threading is not required for this interaction.
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.
    - 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>)` Fundamentally, you are developing two servers, an SMTP server and a HTTP server. It's unto you to decide whether you want to combine the two servers into one through a driver or to leave them as two separate executables. Just make sure to include the appropriate instructions in your README file.

- 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`. 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.
6. Here's a sample `.email` file. Note the timestamp added by the `server`.

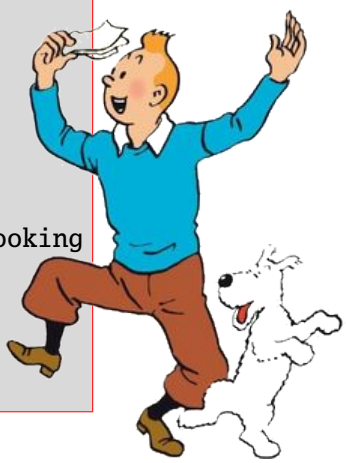
**Answer**

```
Date:  Tue, May 15 2018 10:44:20 -0500
From:  <tintin@447.edu>
To:  <haddock@447.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.
```



7. 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.

8. 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.

**Answer**

HTTP/1.1 200 OK
Server: <server-hostname>
Last-Modified: Tue, May 15 2018 10:54:19 -0500
Count: 1
Content-Type: text/plain
Message: 1

```
Date:  Tue, May 15 2018 10:44:20 -0500 -0500
From:  <tintin@447.edu>
To:  <haddock@447.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.
```

9. Print all reply codes, on all interactions, to the standard output (only).
10. At the end of your implementation, you should be able to:
    - Compile and run your code on a typical Linux machine. Include a README file with clear compilation and runtime instructions.
    - Run your server program(s) first.
    - Run one or more clients to connect to the server and be able to send emails.
    - Run a receiver (either concurrently or sequentially with sending) 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://github.com/google/styleguide`, if you don't already follow one.
- The due date of this assignment is **Tuesday, May 29, 2018 @ 10:59:59 a.m.**. A dropbox will be opened for submission on Moodle.
- Based on past student experience, the UDP multi-threading is not so obvious. I suggest you to first get a single-threaded version working correctly and then think about extending it to multi-threading.

## Deliverables

A complete solution comprises of:

- A short report (max 5 pages) in PDF format of the design and implementation of your system. Your report, at minimum, should include the following subsections:
  - 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 and run instructions.
- A `makefile` to compile your code especially if it involves compiling multiple executables with flag options..
- A compressed tarball of the directory containing your source code, report, REAME, and makefile. **Absolutely do not** include executables, folders created by your programs, your version control repositories, or your test emails in this tarball. To create a compressed tarball of the directory source, use the following command: `tar -zcvf siue-id-pr1.tar.gz source/`.
  e.g. `tar -zcvf tgamage-pr1.tar.gz PR01/`.
- File formatting standards (**`PDF, README, .email, .txt, .tar.gz`**) will be strictly monitored and is subject to penalties.

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 in C. 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/rfc2821