CS 447: Networks and Data Communications

Project #03

Assigned Date	: Tuesday April 21, 2015
Due Date	: **Tuesday May 5, 2014 @ 11:59:59 p.m.**

Overview

For the final project you will be implementing two important features, *multithreading* and a primitive user *authentication* capability, into the mail server that you've been constructing over the semester.

Important

1. Please note that this assignment WILL NOT have the usual 48-hour grace period following the submission deadline. Tuesday May 5, 2014 @ 11:59:59 p.m.is the absolute deadline.
2. Ensure your code compiles and runs in Deterlab. I will download your submitted code from *Moodle* to *DeterLab*, and perform a fresh compile and a run from DeterLab. Code that does not compile and run in this manner will result in automatic 0 points.
3. Provide specific instructions on how to compile and run your code in your report. You may, in addition, provide screenshots where appropriate.
4. Only tarballs with the `tar.gz` extension and reports with the `.pdf` extension are considered in-spec submissions. Other formats (including `.zip, docx, etc.`) are considered out-of-spec and will result in penalties. Contact the instructor immediately if you don't know how to create the correct submission format.
5. Follow the instructions in the Deliverables section below and submit a single tarball with your source code and the report. DO NOT submit executables.
6. Absolutely DO NOT make your users type in protocol commands. Your human-machine interaction should reflect that this is a command-line based email application (to send as well as to receive); your programs should prompt the user to enter the appropriate fields but should not make them type in protocol commands. Failure to do so will definitely considered out-of-spec for this final project.
7. This is an individual assignment. There are no additional requirements for graduate students.
8. Do NOT terminate your experiment on DeterLab when finished.
9. Start early. There will not be any time left in the semester for extensions.
10. Students may use the language of their choice provided it runs in the nodes they have set up on Deterlab. The only limitation is that they may not use libraries that abstract away the lower level mechanisms used by sockets. If you have questions contact Dr. Gamage via e-mail.

Requirement Solicitation #01: Multithreading

You may have already noticed from testing your previous mail server implementations that currently, the mail server is only able to handle a single incoming connection from either a sending client or a receiving client. To allow for more reasonable server usage your first task will be to implement support for concurrent users.

To support multiple concurrent connections you will be implementing a multi-threaded main server program. You must be able to support multiple senders, however you will only be required to support a single receiver. To support this functionality you will need to spin off multiple threads from your primary server process. For the senders you could consider these senders to be the start of a very primitive session, handling all responsibilities for sending email from a single connection. It is highly suggested you have the original thread for the program provide initialization, spin off a receiver thread, and then **listen** to manage creation of sender threads based on **accept**.

Additional Notes on Multi-Threading

One of the primary challenges with multi-threading programs is a situation called a **Race Condition**. This occurs when multiple threads attempt to alter some sort of shared resource simultaneously resulting in a non-deterministic result. For example: two e-mail server threads attempting to write to the same file simultaneously and interleaving two e-mail messages together, or a sender thread writing an e-mail and a receiver thread attempting to read the e-mail before the sender thread is finished writing the information. To handle these sort of situations you will need to **synchronize** your threads appropriately to handle accesses to shared resources.

For this assignment something as simple as a spin-lock is acceptable but you are free to use a mutex, semaphore, monitor, or other synchronization structure of your choice. Due to the wide variety of languages in use for this assignment I won't provide an examples in the text but a quick search of "<language> multi-threading semaphore" for any language I tried resulted in useful results. If you are having issues finding useful resources for this stage for your specific language please let us know.

The second task will be to implement reliability over the existing UDP connection such that when users download their messages they are guaranteed to get all of their e-mails fully intact.

Requirements Solicitation #02: Authentication

Your current email server accepts emails from any user going out to any user. Similarly, it allows any receiver to also download all emails stored in the server. For the second task, we will try to fix these problems with a very primitive authentication mechanism.

Sender Authentication

For sender authentication, your server now will only accept emails from the *cs447.edu* domain. For example, `tgamage@cs447.edu` is valid email while `tgamage@blahblah.blah` is not. Before beginning the SMTP exchange, your server will prompt the sending client to enter his/her username and password. For this assignment, the sender's `cs447.edu` email address is considered the username.

The very first time a sending client attempts to send an email is considered a registration connection, during which, the server will greet the user and reply back with a 4-digit **randomly** created password. The server stores these usernames and passwords in a hidden password file named "`.user_pass`"*. The server also instructs the user to close the connection and login again with the newly created password. Correspondingly, your sender process will:

1. close the current connection (gracefully);
2. wait for 10 seconds;
3. initiate a new connection; and
4. re-attempt to login

this time using the password provided. If the password is accepted by the server, the sender enters the email protocol to send his/her email.

Receiver Authentication

The receiver also follows a similar registration connection as the sender above. Note that a particular sender may decide to send an email to his/her self. In such a situation, the previously stored password is used authenticate the receiver; no need to create a separate receiver password.

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:

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

Functionality Requirements

1. IP addresses/hostnames and port numbers should not be hard coded.
   - Your server executable will accept a single command line argument as follows:
     `./server <port-number>`
   - Your client executables will accept two command line arguments as follows:
     `./sender <server-host-name> <server-port>` (use `25052` as the port number)
     `./receiver <server-host-name> <server-port>` (use `11011` as the port number)
2. `sender-server` connection is over TCP while the `receiver-server` connection is over UDP.
3. You have two servers running: a SMTP server (over TCP) and a POP/IMAP server (over UDP). Consequently, each server should accept connections at two separate ports. *it doesn't make sense for two different protocols to accepts connections over a single protocol*
4. On the receiver end, the client should be given the option to either to download or not to download emails sent to him/her. If the receiver decides not to download the emails, they will remain the server for a future download attempt.
5. Emails, when downloaded by the receiver client should have the following format. There is no specific format requirement on how the server stores the messages:

*(note the "dot" (.) at the beginning of the filename; this is a hidden file)

```
Date:  Mon, 20 Apr 2015 13:04:20 -0500
From:  <tgamage@cs447.edu>
To:  <tgamage@cs447.edu>
Subject:  Email heading
Message:

Hello,

This actually looks like an email.

Sincerely,
CS 447
```

6. Since the default file space for all your DeterLab nodes is the same (i.e. your home folder) create a folder named `DB` to represent the server's file storage for received (but unclaimed) emails. Make each of your receiver clients create a separate folder for itself immediately before it initiates the connection to the server; should be done programatically NOT manually. If the receiver wishes to download, your `GET` command will fetch any emails from the server (over the network) and store them in the respective receiver folder.
7. At the end of your implementation you should be able to:
   - Log into one or more sender nodes on DeterLab
   - Start client programs on each node to send e-mails to the server.
   - The client programs should all proceed according to the requirements for project 02.
   - Once successfully connected the client(s) should prompt user to either enter the receivers e-mail(s), the origin e-mail, and the body of the e-mail.
   - The client(s) should then handle the transaction with the server and then inform the user of success or failure.
   - The client(s) should then end gracefully.
   - Log into the receiver node on DeterLab.
   - Run your client program to receive an e-mail.
   - The client program should prompt the user for a SMTP server to connect to.
   - Once successfully connected the program should display the number of unread messages and ask the user if they would like to download them.
   - The user should then download all messages from all sender clients full intact and receive a confirmation notification from the client program and a list of the files (e-mails) downloaded.
   - The client program should end gracefully, and the user should be able to see the files locally in the same directory as the client program.

Detailed Requirements

- All network communication between sender and server must be done using **TCP**.
- All network communication between server and receiver must be done using **UDP**.
- Ports for the client and server should be assigned in the dynamic range.
- Connections should be properly closed and not left open when exchanges are finished, details on whose responsibility this is located in the RFC.
- Your communication when sending should use the codes set out in RFC 821 4.2.

- You should follow the sending sequence set out in RFC 821 4.3.
- Your server and client should support the following commands: HELO, RCPT TO, MAIL FROM, DATA , QUIT, GET.
- Your server and client should support the following replies: 220, 221, 250, 354.
- The server should save all e-mails in files local to that machine until they are retrieved, at which point they should be deleted from the server.
- The end of the data portion of the email should be identified using a new-line, a period, and another new line.
- The header information on the e-mail should be preserved throughout the transfers.
- All programs should terminate gracefully, not leaving corrupted, half-finished or with open files.
- The program should be able to handle packet loss anywhere in the network.
- In the topology specified in the .NS file the sender should have no direct contact between it and the receiver. NOTE: to conserve resources we will continue to use a 3 machine topology and use the 'receiver' node to send e-mails as well as receive them.
- Make sure your threads terminate both themselves and their connections cleanly.
- the server should have only **one** program to handle sending and receiving.

Deliverables

The due date for is **Tuesday May 5, 2014 @ 11:59:59 p.m.** . A Moodle dropbox will be opened for your submission, which should include:

- A brief report (A README) in `.pdf` format of how you implemented multi-threading and authentication, a overview of your implementation, the language you used, and how to run your project, as well as any issues your final implementation had.
- Your .NS file used to set up the experiment.
- A compressed tarball of the directory containing your source code and the above two items. To create a compressed tarball of the directory cpt447xx, use the following command: `tar -zcvf name-pr1.tar.gz cpt447xx/`. Obviously, change the name to your own.
- Do not include executables, the `.user_pass` file or the `.server_log` in your tarball. The latter two files needs to be automatically created when the instructor compiles and runs your program(s).