CS 4410 Operating Systems Spring 2016

MP3: SMTP Server

IMPORTANT: All students (CS 4410 and CS 4411) are required to do this project.
DUE: Monday, April 18, 2016 at 11:59 PM. Late submissions will NOT be accepted.

Frequency Asked Questions: See this post on Piazza

Description

Email is the backbone of modern-day communication. In this assignment, you will be building a multi-threaded mail server that supports basic features of the Simple Mail Transfer Protocol (SMTP). Along the way you'll get a taste of socket-oriented network programming.

For sake of simplification, you will build a mail server which simply accepts email for local delivery and appends the email to a single local inbox. Despite its limited functionality, your server must be able to support many concurrent clients without allowing any single client to impact any other client.

We've provided you with a bare-bones SMTP server to get you started. The bare-bones server accepts connections on port 8765 and immediately closes each connection without any data transfer. You will start with this bare-bones structure and implement a multi-threaded server that supports mail delivery.

The bare-bones server handles requests sequentially. While this simplifies implementation, it does not allow for concurrent requests. Your first task in this assignment is to change the server so that each incoming connection is passed off to a thread pool. Thread pools are a textbook example of a producer consumer relationship. The single main thread acts as the producer of incoming connections, while a constant number of threads (collectively called the pool) act as a set of consumers.

Your thread pool implementation must only rely on synchronization primitives from Python's threading library (Thread, Lock, Semaphore, Condition). Notably, you *must not* use Python's thread-safe queue module or its multiprocessing module.

Our bare-bones server does not support mail delivery at all. Each incoming connection is immediately closed. Your implementation must support a simplified SMTP protocol. Here's a typical interaction between a client and server in which the client sends mail from student@CS4410 to TA@CS4410:

Server	Client
220 *netid* SMTP CS4410MP3	
	HELO client
250 *netid*	
	MAIL FROM: student@CS4410
250 2.1.0 OK	
	RCPT TO: TA@CS4410
250 2.1.5 OK	
	DATA
354 End data with <CR><LF>.<CR><LF>	
	I made an awesome submission! Network programming is fun .
250 OK: delivered message 1	

SMTP Details

SMTP is a line-oriented protocol. Although it is not shown in the previous exchange, each line ends with the special carriage-return/line-feed combination '\r\n'. Notice that with the exception of the message body, each line triggers a response from the server. The special number codes begin each line from the server and provide feedback to clients about the success and failure of each command.

The communication between the client and server occurs over TCP. While TCP guarantees reliable delivery, it does not necessarily guarantee that data sent at the same time on the client will be received at the same time on the server (and vice versa). For instance, if the client sends the string "HELO abc12\r\n", the string can be chopped into three packets delivered separately at the servers, containing "HE", "LO", and "abc12\r\n". You will most likely want to write a "collect_input" function that performs recv() operations in a loop until a whole message has been received, instead of trying to parse the first packet (containing only "HE") received.

SMTP is not case sensitive: HELO, helo, and HeLO are the same command. Commands may contain whitespace around valid tokens, and there may be trailing spaces after arguments. It is *not* acceptable to allow whitespace in email addresses or hostnames.

A HELO only needs to be sent once per session and will carry over between messages. No other command persists after a message is delivered.

Multiple RCPT TO commands are allowed. Doing so will result in multiple "To:" headers in the delivered mailbox, but the email will only be assigned one number.

Note: There is a \r\n after DATA and after 200 OK.

Status and Error Codes

The numbers that precede the responses from the server are status codes. By convention, numbers in the 200-299 range indicate that acceptable progression through mail delivery. Numbers in the 500-599 range indicate errors, 400-499 indicate timeouts, and 300-399 indicate alert conditions (such as the mail server changing its input collection mode such that it is now seeking input until a "<CR><LF>.<CR><LF>" marker). When picking response codes, you must follow the conventions as indicated in various parts of this document (e.g. 220 for the first message, 421 for timeouts, 354 for the response to a successful DATA message, etc).

SMTP is intended to be human-readable, and it used to be the case that people would regularly send emails by hand. As such, the protocol is very tolerant of errors. In the event of an error, the server delivers a message, but does not prevent future commands from succeeding. There are several error cases we expect you to gracefully handle:

```
Recognized commands that do not obey the syntax we've described should receive a response
    501 Syntax: proper syntax

Unrecognized commands should receive a response of:
    502 5.5.2 Error: command not recognized

If a client sends multiple HELO commands, the appropriate response is:
    503 Error: duplicate HELO

If a message sends duplicate MAIL FROM commands, the appropriate response is:
    503 5.5.1 Error: nested MAIL command

Commands must come in the order HELO, MAIL FROM, RCPT TO, and DATA.  For commands out of
    503 Error: need XXX command

For simplification, we consider any email address that consists of consecutive, printable
    504 5.5.2 <bad email>: Sender address rejected

Similarly, if an email provided as an argument to RCPT TO contains significant whitespace
    504 5.5.2 <bad email>: Recipient address invalid

Clients which are slow, inactive, or continually sending invalid commands should be disco
    421 4.4.2 netid Error: timeout exceeded
```

Another Example

As mentioned previously, sending an error message does not terminate communication. Below we show an example SMTP conversations which almost succeeds despite being interrupted with errors.

Server	Client
220 netid SMTP CS4410MP3	
	HELO
501 Syntax: HELO yourhostname	
	HELO clienthostname
250 netid	
	mkdir
502 5.5.2 Error: command not recognized	
	MAIL FROM: student @CS4410
504 5.5.2 <student @CS4410>: Sender address rejected	
	MAIL FROM: student@CS4410
250 2.1.0 OK	
	MAIL FROM: professor@CS4410
503 5.5.1 Error: nested MAIL command	
	RCPT TO: TA @CS4410
504 5.5.2 <TA @CS4410>: Recipient address invalid	
	RCPT TO: TA@CS4410
250 2.1.5 OK	
	DATA
354 End data with <CR><LF>.<CR><LF>	
	I'm going to send and wait
421 4.4.2 netid Error: timeout exceeded	

Had the client sent a trailing "." before the timeout, this message would have been successfully delivered despite encountering every error response your basic SMTP server must generate.

Timeouts

The timeout should not start when the connection is established, but from the time the server starts expecting the next valid line (transaction) in the protocol. So the client should get 10 seconds to provide each part (HELO, MAIL FROM, RCPT TO, DATA) of the protocol, for a maximum possible connection time of (10 * 4 =) 40 seconds, 10 seconds per line.

Note that the protocol guarantees 10 seconds to the client before it times out: *timing out earlier is not acceptable!* But timing out much later than 10 seconds isn't great either lest you waste precious server resources.

Email Inbox

Email should be saved on disk in a file named 'mailbox' (not mailbox.txt). When we launch your SMTP server, it should create this file in the current directory, erasing all current contents. Each email should be appended to this file in a way that preserves the integrity and order of the emails. For example, if the emails from the previous two examples were both delivered, the resulting mailbox file should be:

```
Received: from client by netid (CS4410MP3)
Number: 1
From: student@CS4410
To: TA@CS4410

I made an awesome submission!
Network programming is fun.

Received: from clienthostname by netid (CS4410MP3)
Number: 2
From: student@CS4410
To: TA@CS4410

I'm going to send and wait
```

You can test your server by connecting to it with a telnet client. To do so, run your server on a linux machine, and on the same machine run "telnet localhost 8765" (or whatever port your server is currently listening on).

Multiclient

Once you have a functioning client and server, you should modify our provided client to thoroughly test your server program. As part of this modification, you should create a version called 'multiclient.py' which opens 32 connections at once, and psuedorandomly generates SMTP commands. An implementation which is eligible for full credit will randomly vary the client-provided information (hostnames, emails, and data) and will follow the

core protocol in the normal case. To test error conditions, 'multiclient.py' should randomly decide to deviate from the protocol and issue a command that results in an error. It should be able to detect that your server sends the appropriate response. Please see the included 'multiclient.py' skeleton for more information.

Inbox Backup (Required for 4410, Extra Credit for 4411 Students)

Create a separate "backup" thread whose job is to create a backup of the mailbox. Creating a backup involves copying the current mailbox file to a separate location, and emptying mailbox. Messages should be backed up in groups of 32. For instance, upon delivering message number 64, the backup thread will create file mailbox.33-64, and then truncate mailbox. After this thread completes, the current directory will contain the files mailbox (not mailbox.txt), mailbox.1-32, and mailbox.33-64.

The backup process should not be affected if another thread tries to deliver a message. The thread that tries to deliver a new message will wait for the backup process to complete before writing its message.

For CS 4411 students, please note in your README.txt file if you have decided to implement this feature.

Considerations

- In all cases above where it says "netid", replace that with your netid.
- Your thread pool should contain 32 threads.
- Each thread should finish one connection before moving on to the next.
- The consumer (which calls accept on the listening socket) should not accept connections at a faster rate than the thread pool is processing them. It is better to let them become backlogged in the OS than to accept them and let them sit on a queue. Your implementation must behave in this fashion.
- Emails should be assigned a unique number, indicating their order of delivery. This number should reflect the order of the emails in the mailbox. For consistency, the first message is always 1, and every subsequent message is <number of previously delivered message> + 1.
- You'll need to build at least one monitor to enable your multi-threaded behavior to adhere to our spec. Servers which do not implement proper monitors will not receive full credit.
- We will test on the Linux networking stack using Python 2.7. Please be sure to test your implementation on the VM if you are not developing on it.
- It is perfectly acceptable, and encouraged, to cross-test your SMTP server with other students. The procedure for doing so is to both log into the CSUGLab machines and spawn your server on a port that is known to both you and others. At no time should you be in possession of, or have access to, the source code of another student.

Deliverables

Follow the same GitHub release procedure as in previous projects.

- **server.py**

This should be your server implementation. If run with no arguments, server.py should bind to '127.0.0.1' on port 8765. Messages should be saved to the mailbox file in the directory in which server.py is launched.

- **multiclient.py**

If run with no arguments, multiclient.py should connect to '127.0.0.1' on port 8765 and begin the stress test. The stress test should run through every error case in the protocol, and possibly other error cases that you yourself conceive. We will run multiclient.py for 1 minute and expect at least 1000 operations, requiring it to sustain about 17 ops/sec.

- **QUESTIONS.txt**

A plain-text, ASCII file containing your netid and answers to every question contained in the distributed QUESTIONS.txt.

- **README.txt**

A plain-text, ASCII file containing your netid and a short description of your implementation and any potential problems.

Grading Breakdown:

Programming Components: 80%
Written Questions: 20%