Jonathan Reynolds

CS 447

Project 1

09/27/2018

In order to implement the SMTP server I decided to use four files. A client and server to handle the e-mail creation process as well as a client and server to handle the retrieval of the e-mail once it has been stored.

The SMTP server should be able to send the following response codes:

* 500 – syntax error, command unrecognized
* 501 – syntax error in parameters or arguments
* 503 – bad sequence of commands
* 250 – requested action completed OK
* 354 – start mail input, end with <CRLF> . <CRLF>
* 221 - <domain> service closing transmission channel

Response codes 500, 501, and 503 are included as they allow for more precise error reporting. Response code 250 is included as it allows the client to be updated on the success of their request. Code 354 is included as it is the response to the DATA command and as such notifies the user that they can begin writing the email. Code 221 is included as it notifies the client when their connection with the client ends, following the QUIT command.

In order to support the HTTP Get functionality the following response codes were selected.

* 200 – OK
* 400 – bad request
* 404 – file not found

Code 200 was included to notify the user of the successful completion of the Get request. Code 400 was included in the event the user makes an error in there request, such as mistyping the path of the Get. Code 404 was included to handle the event wherein the server is unable locate the requested file.

The SMTP ‘sender’ client and server were designed so that once the client connects to the server over a TCP connection the client will be in a read/write loop until the termination of the upload process. The server on the other hand will transition through states, represented as integers. (state 0 = HELO, state 1 = MAIL FROM …) This structure was achieved using an if-else tree.

When the server is first started it checks to see if its ‘db’ folder is created, if not the server creates it. After this the server creates, binds, and listens on a socket using the TCP port number. Once the client attempts a connection with the client the server will accept the connection, but before this the server forks the process to allow for multiple user to connect to over the same socket. The server\_socket that is used to send data to the client is created when the connection is accepted.

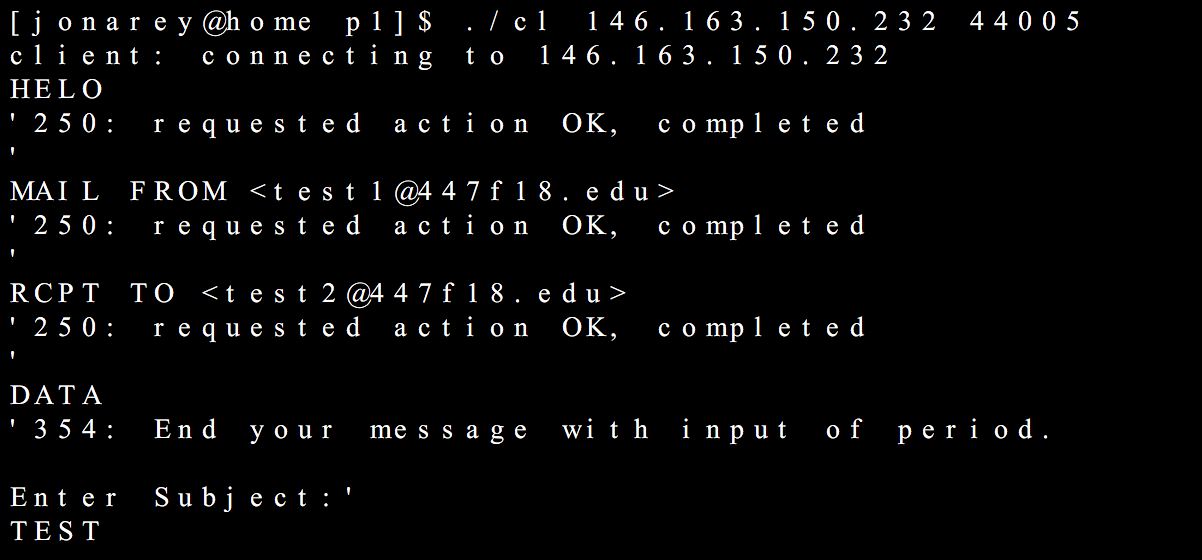
The server’s behavior to the client’s input was determined by the current state that the server was in. While moving through the states the server store’s information relevant to the SMTP protocol, such as sender and receiver e-mails, in global variables so that they are accessible anywhere throughout the program. When the server is given potential e-mails in the MAIL FROM and RCPT TO commands it verifies that they are of the 447f18.edu domain or it will send a response code of 501 to the client.

When a new receiver is entered as an argument for the RCPT TO command the server checks if the receiver already has a folder inside of ‘db’, creating one if it does not exist. Additionally at this point the server creates a config file inside of the receiver’s directory. This is file purely stores the count of how many e-mails are currently in the receiver’s folder although I would like to attempt storing additional data in the file containing specifics about the receiver. The config file is read when the DATA command is entered as incrementing the count of the config file by one provides the name for the next e-mail, and writing this value back to the file allows for the e-mail count to be easily tracked for each user. Additionally it needs to be cited that the getLastLine function used to retrieve the value from the config file was written by user sea otter in the following thread. (//http://www.webhostingtalk.com/showthread.php?t=327009)

I was unable to ever get the DATA command fully working as the buffer between the client and server kept breaking when I attempted to loop and read data until a period was entered to conclude the e-mail. I was unable to get this error remedied and as such the SMTP sender server is unable to fully write e-mails although it does create them sequentially and is able to write the sender, receiver and subject to the e-mail before breaking.

The SMTP receiver client and server follow a similar design pattern to that of the sender although their protocol implementation is not complete. The client and server as they are now just share a working UDP connection and are able to send data back in forth in a loop. I found the UDP to be much simpler to setup than the TCP connection as there were fewer system calls to establish the server socket.

The receiver client and server were planned to follow an execution format very similar to that of the sender client and server. The receiver server would transition through states as it collected and verified the information necessary to complete the Get request. The receiver’s config file would allow the server to easily verify if the count of emails to be transferred was higher than the number of e-mails in a user’s folder, thus allowing for an easy way to detect invalid commands.



Output of the sender client.cpp during “successful” execution. The client stops working in the DATA command state after the subject has been entered.

In conclusion, the sender SMTP client and server are able to successfully handle the HELO, MAIL FROM, and RCPTO TO commands, although they are unable to complete the DATA and QUIT commands. Additionally the server is able to detect invalid input in both the command and parameters for the HELO, MAIL FROM and RCPT TO commands. The sender SMTP client and server have a working TCP connection and the receiver SMTP client and server have a working UDP connection.

The main issue I ran into during development was the buffer’s failure whenever I attempted to write to the .email file. This delayed my implementation of the receiver client and server and as such I was unable to complete the fully implemented Get protocol.