COMP 431 — INTERNET SERVICES & PROTOCOLS

Kevin Jeffay

Fall 2021

Homework 2, August 30

Due: 11:15 AM, September 15

More Baby-steps Towards the Construction of an SMTP Server

In this assignment you will extend the program you developed for Homework 1 to parse more SMTP commands. In addition to the "MAIL FROM" command, your parser will now recognize the "RCPT TO" and "DATA" commands. You will also turn your parser into the basic engine of an SMTP server by having it process the DATA command to receive and store the contents of a mail message in a file.

To begin, you will extend your parser to recognize two additional commands:

```
RCPT TO: <jeffay@cs.unc.edu>
DATA
```

The RCPT TO command is used to specify the recipient of an email message and the DATA command is used to communicate the body of the email message to be sent to the SMTP server.

The RCPT TO command

The grammar for the RCPT TO command is:

```
 <rcpt-to-cmd> ::= "RCPT" <whitespace> "TO:" <nullspace> <forward-path>
                   <nullspace> <CRLF>
<forward-path> ::= <path>
```

where the `path`, `whitespace`, and `nullspace` non-terminals are the same as in the grammar for the MAIL FROM command in Homework 1.

One change from Homework 1 is that for well-formed MAIL FROM commands your program reads, your program should now reply with a line of the form:

```
250 OK
```

This reply should also be emitted for well-formed RCPT TO commands your program reads. This response is required by the SMTP protocol and its meaning will be explained later in the course.

The DATA command

The DATA command is the command used in the SMTP protocol to transfer the data (*e.g.*, the text) of an email message from an SMTP client to an SMTP server. The grammar for the DATA command is simply:

```
<data-cmd> ::= "DATA" <nullspace> <CRLF>
```

When your program reads a well-formed DATA command, your program should reply with a text message prompting the sender for the message data (*e.g*., the text of the email) with a line of the form:

```
354 Start mail input; end with <CRLF>.<CRLF>
```

where "`<CRLF>`" in this case refers the literal character string "`<`", "`C`", "`R`", "`L`", "`F`", "`>`", and not to the carriage return and line-feed characters.

After a DATA command is input, all subsequent lines of input are treated as message data (*i.e*., the body of an email message). Each line of input should be read in and stored in a file as described below.

Processing of the DATA command ends when the user enters a line containing only a single period (*i.e*., when the sequence <newline> "." <newline> is read). This last line is *not* considered to be part of the mail message and hence is not stored with the contents of the mail message.

If the DATA command and the subsequent email message text has been successfully entered, your program should again reply with a line of the form:

```
250 OK
```

As an example, consider the following text of an email message:

```
Hey Bob, do you really think we should use SMTP as a class
project in COMP 431 this year? The smart students are going to
figure out automated ways to use this project to send
anonymous SPAM to the world!
```

If we wanted to send this text as an email message to two users, say *bob@cs.unc.edu* and *alice@cs.unc.edu*, then a complete, valid interactive exchange with your SMTP server program to transfer the text of this message might look like the following:[1]

```
Input:  MAIL FROM: <jeffay@cs.unc.edu>
Output: MAIL FROM: <jeffay@cs.unc.edu>
Output: 250 OK
Input:  RCPT TO: <bob@cs.unc.edu>
Output: RCPT TO: <bob@cs.unc.edu>
Output: 250 OK
Input:  RCPT TO: <alice@cs.unc.edu>
Output: RCPT TO: <alice@cs.unc.edu>
Output: 250 OK
Input:  DATA
Output: DATA
Output: 354 Start mail input; end with <CRLF>.<CRLF>
Input:  Hey Bob, do you really think we should use SMTP as a class
Output: Hey Bob, do you really think we should use SMTP as a class
Input:  project in COMP 431 this year? The smart students are going to
```

[1] This example assumes you are manually typing your inputs to your program. If you type your commands into your program the above is what you can expect to see in your terminal window. The "Input" lines represent what Linux will echo to your terminal/ssh window as you type in commands and the "Output" lines represent the outputs that your program will generate in response. (However, note that the "Input:" and "Output:" prefixes above are included to demonstrate the operation of the program and are not part of the actual input or output.) If you use I/O redirection to read commands from a file (or write the output to a file) you will not see all these lines in your terminal window.

```
Output: project in COMP 431 this year? The smart students are going to
Input:  figure out automated ways to use this project to send
Output: figure out automated ways to use this project to send
Input:  anonymous SPAM to the world!
Output: anonymous SPAM to the world!
Input:  .
Output: .
Output: 250 OK
```

Note that the DATA command isn't fully "accepted" (the 250 acknowledgement message isn't emitted) until after the body of the mail message is read in. As described above, after the DATA command is entered, the intermediate response

```
354 Start mail input; end with <CRLF>.<CRLF>
```

is generated. The "250 OK" acceptance response isn't generated until the end of input sequence (<newline>.<newline>) is read in.

Error Processing

For all ill-formed or unrecognized SMTP commands, your program should now reply with one of the following error messages:

```
500 Syntax error: command unrecognized
501 Syntax error in parameters or arguments
```

Ill-formed MAIL FROM, RCPT TO, and DATA commands can generate either of these errors. Whenever the first parsed tokens on an input line do not match the literal string(s) in the production rule for any command in the grammar, a type 500 error message is generated. Operationally, a 500 error means that your parsing can't recognize which SMTP command it should be parsing.

If the command token(s) are recognized (*i.e.*, your parser "knows" what command it's parsing), but some other error occurs on the line, a type 501 error message is generated. The following would be examples of 500 and 501 errors:

```
MAILFROM: <jeffay@cs.unc.edu>
500 Syntax error: command unrecognized
MAIL FROM <jeffay@cs.unc.edu>
500 Syntax error: command unrecognized
RCPT TO : < bob@cs.unc.edu>
500 Syntax error: command unrecognized
MAIL   FROM: <jeffay @cs.unc.edu>
501 Syntax error in parameters or arguments
RCPT TO: < bob@cs.unc.edu
501 Syntax error in parameters or arguments
MAILFROM: <jeffay @cs.unc.edu>
500 Syntax error: command unrecognized
```

Note that this last command appears to have two errors in it. It has a syntax error in the command name and, it appears to also have a parameter error ("jeffay @"). We report a syntax error and not a parameter error because since a parser would not recognize "MAILFROM," the parser does not "know" what SMTP command it is parsing. And since the parser does not know what SMTP command it is parsing it can't know if what follows (the parameters/arguments) is correct or not. Said another way, syntax errors in the SMTP command name (500 errors) take precedence for reporting over parameter/argument errors (501 errors). That is, in order to have a 501 error, you have to first recognize which SMTP command you are parsing the parameters/arguments for.

Beyond syntax and parameter/argument errors, you now need to also detect sequencing errors. SMTP requires that MAIL FROM, RCPT TO, and DATA commands be received in a particular order. Specifically, SMTP commands must be received in the order:

- a MAIL FROM command must be received first, followed by
- 1 or more RCPT TO commands, followed by
- a DATA command.

You can view these ordering constraints as either a higher-level grammar for a "SMTP session" or you can view them as states in a state-machine. We'll take the latter approach. Thus, a new processing requirement for this assignment includes building a simple protocol state machine to ensure SMTP commands are generated in the correct order.

Your program should start in the state of waiting to receive a MAIL FROM command (or end-of-file) and upon receiving this command, transition to the state of waiting for 1 or more RCPT TO commands. Once one or more RCPT TO commands have been received, upon receipt of a DATA command, your program should transition to the state of processing message data (described below).

Whenever a command is received out of order, a type 503 error is generated:

```
503 Bad sequence of commands
```

Syntax errors in the command name (type 500 errors) should take precedence over all other errors. That is, before a command can be considered to be out-of-order, the command being parsed must be recognized. Thus, for example, if a RCPT TO: command is expected but the input DATAX is seen next (*i.e.*, an ill-formed DATA command is seen), this would be considered a type 501 syntax error and not an out-of-order error.

Out-of-order errors (type 503 errors) should take precedence over parameter/argument errors (type 501 errors). That is, if a RCPT TO: command is expected but the input "MAIL FROM: bob" is seen, this would be considered an out-of-order error and not a parameter/argument error even though there is an error in the parameters for the MAIL FROM: command. Said another way, parameter/argument errors can only occur on commands that (1) have a syntactically correct command name, and (2) are received in the correct order.

It's up to you to decide how your state machine will recover from errors. Give some thought as to how to deal with syntax or out-of-order errors in a robust manner. Two obvious choices are to reset the state machine (*i.e.*, to return to the state of waiting from a MAIL FROM command) or to remain in the current state waiting for a correct version of the awaited command. However, in all cases, the parsing of a single SMTP command line generates either a 250 "OK" message, a 354 "Start" message, or one (and only one) of the error messages described above.

Processing Multiple Messages

The input of a MAIL FROM command, followed by one or more RCPT TO commands, followed by a DATA command, followed by the text of an email message, corresponds to the sending of a single email message. Once a DATA command has been received and processed (as explained below), another MAIL FROM command can be received. This would correspond to the scenario of an SMTP client sending multiple email messages to a server in a single SMTP session.

Thus, in terms of the SMTP state machine, upon receipt and processing of a DATA command, your program should transition back to the state of waiting to receive either a MAIL FROM command or an end-of-file indication.

In all cases, the SMTP processing continues until end-of-file is reached on the input stream.

Homework Requirements

Write a Python program to read in lines of input from standard input and process them as SMTP commands according to the SMTP grammar required for Homework 1 and as extended herein. You should check for syntax errors as well as ordering errors in the order in which commands are input and emit error messages as appropriate. All acknowledgement and error messages should be formatted *exactly* as shown above.[2] As in Homework 1, your program should read from standard input and echo all input lines and acknowledgement/error messages to standard output. As before, your program must not output any user prompts, debugging information, status messages, *etc*.

For all valid "mail messages" you process (*i.e*., all valid sequences of a MAIL FROM command, one or more RCPT TO commands, a DATA command, and an arbitrary number of lines of message body terminated by a period on an otherwise blank line), you should append the following information into *n* Linux files, where *n* is the number of (valid) RCPT TO command lines received.

```
From: <reverse-path>
To: <forward-path-1>
           :
To: <forward-path-n>
```
Text entered after the data command

This text should be appended to the files *forward/forward-path-1*, *forward/forward-path-2*, through *forward/forward-path-n* in the current working directory (the directory in which you executed your program) where *forward-path-k* is the name of the kth forward path in the message. Thus, for the message in the example above, the text:

```
From: <jeffay@cs.unc.edu>
To: <bob@cs.unc.edu>
To: <alice@cs.unc.edu>
Hey Bob, do you really think we should use SMTP as a class
project in COMP 431 this year? The smart students are going to
figure out automated ways to use this project to send
anonymous SPAM to the world!
```

would be appended to the files *forward/bob@cs.unc.edu* and *forward/alice@cs.unc.edu* in the current working directory. You may assume that the directory *forward* already exists in the current working directory. (Thus, when you are testing your program be sure to (manually) create this directory before you execute your program!) However, if the *forward-path* file does not already exist it will have to be created by your program as part of the process of writing to the file.

You should keep processing lines of input until end-of-file is reached on the input stream. After the DATA command and its accompanying message data are read in, if end-of-file is not reached on the input stream, your program should expect another MAIL FROM command from standard input. If errors are encountered on input lines you should simply emit the appropriate error message and begin the parse of

[2] This is done for ease of grading. The actual SMTP specification only specifies the response numbers and allows arbitrary text to follow the error code/acknowledgement number.

the next line of input. If end-of-file is reached in the middle of processing an email message then no message data for this message should be written to the forward-path file. That is, only complete, well-formed messages should be written to the *forward-file*.

Grading

As per the instructions distributed with Homework 1, create a subdirectory named HW2 in your home directory, and place the final version of your program in this directory before the due date. Please give your final program the name "SMTP1.py."

When your program is complete, send an email to the TAs (*noahj98@live.unc.edu*, *crisingc@live.unc.edu*) saying your program is ready for grading. As before, please be sure to include your Linux login id in your submission message.

As before, your program should be neatly formatted (*i.e.*, easy to read) and well documented.